Reliable Scalable Cluster Technology
Version 3.1.5.0

Programming RMC for RSCT

Reliable Scalable Cluster Technology
Version 3.1.5.0

Programming RMC for RSCT

Note
Before using this information and the product it supports, read the information in "Notices" on page 289.

This edition applies to Reliable Scalable Cluster Technology Version 3.1.5.0 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

About this document

This publication describes the microsensor application programming interface (API) and the resource monitoring and control (RMC) API.

The RMC API is a library of subroutines and supporting datatypes, written in C, that enable cluster applications to connect to the RMC subsystem to perform the following tasks:

- list the resources of a resource class
- monitor changes in attribute values for events of interest
- query dynamic or persistent attributes of resources or resource classes
- change the persistent attributes of resources or resource classes
- define and undefine resources
- bring resources online and take them offline

Before using the information about the RMC API in this publication, you must first understand the RMC subsystem and resource managers. See the **Managing and monitoring resources using RMC and resource managers** chapter in the *Administering RSCT* guide for an overview.

Highlighting

The following highlighting conventions are used in this document:

Table 1. Conventions

Convention	Usage
bold	**Bold** words or characters represent system elements that you must use literally, such as commands, flags, path names, directories, file names, values, PE component names (**poe**, for example), and selected menu options.
bold underlined	**Bold underlined** keywords are defaults. These take effect if you do not specify a different keyword.
`constant width`	Examples and information that the system displays appear in `constant-width` typeface.
italic	*Italic* words or characters represent variable values that you must supply. *Italics* are also used for information unit titles, for the first use of a glossary term, and for general emphasis in text.
<*key*>	Angle brackets (less-than and greater-than) enclose the name of a key on the keyboard. For example, <**Enter**> refers to the key on your terminal or workstation that is labeled with the word *Enter*.

\	In command examples, a backslash indicates that the command or coding example continues on the next line. For example:
	`mkcondition -r IBM.FileSystem -e "PercentTotUsed > 90" \`
	`-E "PercentTotUsed < 85" -m d "FileSystem space used"`

{*item*}	Braces enclose a list from which you must choose an item in format and syntax descriptions.
[*item*]	Brackets enclose optional items in format and syntax descriptions.
<**Ctrl-***x*>	The notation <**Ctrl-***x*> indicates a control character sequence. For example, <**Ctrl-c**> means that you hold down the control key while pressing <**c**>.
item...	Ellipses indicate that you can repeat the preceding item one or more times.
|	• In *synopsis* or *syntax* statements, vertical lines separate a list of choices. In other words, a vertical line means *Or*. • In the left margin of the document, vertical lines indicate technical changes to the information.

Entering commands

When you work with the operating system, you typically enter commands following the shell prompt on the command line. The shell prompt can vary. In the following examples, $ is the prompt.

To display a list of the contents of your current directory, you would type `ls` and press the **Enter** key:

```
$ ls
```

When you enter a command and it is running, the operating system does not display the shell prompt. When the command completes its action, the system displays the prompt again. This indicates that you can enter another command.

The general format for entering operating system commands is:

Command Flag(s) Parameter

The flag alters the way a command works. Many commands have several flags. For example, if you type the **-l** (long) flag following the **ls** command, the system provides additional information about the contents of the current directory. The following example shows how to use the **-l** flag with the **ls** command:

```
$ ls -l
```

A parameter consists of a string of characters that follows a command or a flag. It specifies data, such as the name of a file or directory, or values. In the following example, the directory named `/usr/bin` is a parameter:

```
$ ls -l /usr/bin
```

When entering commands in, it is important to remember the following items:

- Commands are usually entered in lowercase.
- Flags are usually prefixed with a - (minus sign).
- More than one command can be typed on the command line if the commands are separated by a ; (semicolon).
- Long sequences of commands can be continued on the next line by using the \ (backslash). The backslash is placed at the end of the first line. The following example shows the placement of the backslash:

  ```
  $ cat /usr/ust/mydir/mydata > \
  /usr/usts/yourdir/yourdata
  ```

When certain commands are entered, the shell prompt changes. Because some commands are actually programs (such as the telnet command), the prompt changes when you are operating within the command. Any command that you issue within a program is known as a subcommand. When you exit the program, the prompt returns to your shell prompt.

The operating system can operate with different shells (for example, Bourne, C, or Korn) and the commands that you enter are interpreted by the shell. Therefore, you must know what shell you are using so that you can enter the commands in the correct format.

Case sensitivity in AIX

Everything in the AIX® operating system is case sensitive, which means that it distinguishes between uppercase and lowercase letters. For example, you can use the **ls** command to list files. If you type `LS`, the system responds that the command `is not found`. Likewise, **FILEA**, **FiLea**, and **filea** are three distinct file names, even if they reside in the same directory. To avoid causing undesirable actions to be performed, always ensure that you use the correct case.

ISO 9000

ISO 9000 registered quality systems were used in the development and manufacturing of this product.

RSCT versions

This edition applies to RSCT version, release, modification, and fix number 3.1.5.0.

You can use the **ctversion** command to find out which version of RSCT is running on a particular AIX, Linux, Solaris, or Windows node. For example:

```
/usr/sbin/rsct/install/bin/ctversion
```

An example of the output follows:

```
# /usr/sbin/rsct/install/bin/ctversion
rlis1313a 3.1.5.0
```

where, `rlis1313a` is the RSCT build level.

On the AIX operating system, you can also use the **lslpp** command to find out which version of RSCT is running on a particular AIX node. For example:

```
lslpp -L rsct.core.utils
```

An example of the output follows:

```
  Fileset                      Level    State  Type  Description (Uninstaller)
  ----------------------------------------------------------------------------
  rsct.core.utils              3.1.5.0    C     F    RSCT Utilities

State codes:
 A -- Applied.
 B -- Broken.
 C -- Committed.
 E -- EFIX Locked.
 O -- Obsolete.  (partially migrated to newer version)
 ? -- Inconsistent State...Run lppchk -v.

Type codes:
 F -- Installp Fileset
 P -- Product
 C -- Component
 T -- Feature
 R -- RPM Package
```

On the Linux operating system, you can also use the **rpm** command to find out which version of RSCT is running on a particular Linux or Solaris node. For example:

```
rpm -qa | grep rsct.basic
```

On the Windows operating system, you can also perform the following steps to find out which version of RSCT is running on a particular Windows node:

1. Click the Windows **start** button.
2. Select **All Programs**.
3. Select **Tivoli SA MP Base**.
4. Click **SA MP Version**.

Related information

The following PDF documents that contain RSCT information can be found on the Reliable Scalable Cluster Technology (RSCT) PDFs page:

- *Administering RSCT*
- *Messages for RSCT*
- *Programming Group Services for RSCT*
- *Technical Reference: RSCT for AIX*
- *Technical Reference: RSCT for Multiplatforms*
- *Troubleshooting RSCT*

Programming RMC for RSCT

This publication is intended for programmers who want to create applications that use the RMC API to connect to the RMC subsystem to leverage its resource management and monitoring capabilities.

This publication is also intended for programmers who want to create microsensors.

The programmer should be familiar with UNIX operating systems, networked systems, and the C programming language.

What's new in Programming RMC for RSCT

Read about new or significantly changed information for the Programming RMC for RSCT topic collection.

How to see what's new or changed

In this PDF file, you might see revision bars (|) in the left margin that identify new and changed information.

November 2013

The following information is a summary of the updates made to this topic collection:

- Updated information about RMC commands and client applications in the "RMC subsystem session" on page 2 topic.
- Updated information about the **MC_SESSION_OPTS_SR_SCOPE** session scope in the "mc_start_session" on page 206 and "mc_timed_start_session" on page 212 topics.

RMC API concepts

Before you use calls to the RMC API in an application, you should understand basic concepts. For example, you should understand what an RMC session is and how to establish one. You should also have an understanding of the RMC API's base programming model including how commands are sent to the RMC subsystem, how responses are returned to the application, and how the application can register for event notifications. You should also have a high-level understanding of the RMC API subroutines that are available.

Understanding the RMC API

The RMC API is a library of subroutines and supporting data types, written in C.

They enable applications (in particular, cluster applications) to establish a connection with the RMC subsystem to:

- list the resources of a resource class
- monitor changes in attribute values for events of interest
- query dynamic or persistent attributes of resources or resource classes
- change the persistent attributes of resources or resource classes
- define and undefine resources
- bring resources online and take them offline

Using the RMC API, an application can:

- establish one or more sessions with the RMC subsystem. The scope of a session determines whether subsequent commands within the session can effect nodes of an RSCT peer domain, nodes and subdomains of a CSM management domain, or a standalone node only.
- send commands to the RMC subsystem using blocking or nonblocking subroutines. The RMC API can return responses to the application using callback routines or by returning a pointer to the response. Callbacks are also used to notify the application when events for which it has registered occur.
- when calling a subroutine to issue a command to the RMC subsystem, target the command at one or more resources of a resource class, or one resource class.
- obtain error information returned by the RMC API subroutine, an RMC subsystem daemon, or a resource manager.

RMC subsystem session

A resource monitoring and control (RMC) subsystem session is a connection with the RMC subsystem that the application establishes through an RMC daemon that runs on a particular node.

Before calling any other RMC API subroutines, the application must establish a session by calling either the **mc_start_session** or the **mc_timed_start_session** subroutine. Both subroutines identify one or more nodes that the RMC API can contact in an attempt to connect to the node's RMC subsystem daemon, and so establish a session. If a session is successfully established, the subroutine returns a session handle that must be used in subsequent subroutine calls to identify the session. The only difference between the **mc_start_session** and **mc_timed_start_session** subroutines is that the **mc_timed_start_session** subroutine also specifies time limits for establishing the session and, after the session is established, for completion of blocking operations.

Since the RMC subsystem operates within the bounds of either an RSCT peer domain, a CSM management domain, or a stand-alone system, the scope of a particular session is similarly limited. Subsequent calls to the RMC API issue commands that can affect the following items:

- Nodes in an RSCT peer domain. It is called a peer domain or shared resource (SR) session scope.
- Nodes in a CSM management domain, which might include one or more RSCT peer domains. It is called a management domain or distributed management (DM) session scope.
- An individual node that runs the RMC subsystem. It is called a local session scope.

Note: If your operating system has an installation of RSCT 3.1.5.0, or later, the RMC subsystem is not recycled on nodes when the node is brought online or taken offline from a peer domain. The RMC clients that established the session on the nodes do not lose the connection to RMC when the node is brought back online. Similarly, the RMC clients that specify a scope other than SR do not lose connection to RMC if the peer domain or peer node is taken offline.

Although a session scope is limited to a particular RSCT peer domain, CSM management domain, or individual node, an application can establish multiple sessions with the RMC subsystem in separate domains or on separate stand-alone nodes. This capability enables the application to call subroutines to issue commands in different domains or on different stand-alone nodes. The session handle that is returned by the **mc_start_session** or the **mc_timed_start_session** subroutine, and required as input to other subroutines, enables the application to direct a particular subroutine's command at the appropriate domain or stand-alone node.

The scope of a session that is established with the RMC subsystem depends upon the following items:

- A session scope option that is specified as a parameter to the **mc_start_session** or the **mc_timed_start_session** subroutine.
- The execution environment of the RMC subsystem daemon with which the session is established.

If the session scope that is specified when calling the **mc_start_session** or **mc_timed_start_session** subroutine is not supported on the node that the RMC API contacts to connect to the RMC subsystem daemon, a session cannot be established. Instead, the subroutine returns an error. For example:

- When calling the **mc_start_session** or **mc_timed_start_session** subroutine, the application specifies a required session scope of peer domain (SR). However, when specifying the nodes for the RMC API to contact in an attempt to establish a session, the application specifies a node that is not in a peer domain. The RMC API tries to establish a session with the RMC subsystem daemon on this node and fails, returning an error to the application.
- To establish a management domain scope, the session must be established with the RMC subsystem daemon on the management server of the management domain (and not one of the managed nodes). When calling the **mc_start_session** or **mc_timed_start_session** subroutine, the application specifies a management domain scope, but includes only managed nodes in the list of nodes the RMC API must attempt to contact to establish a session. The RMC API tries to establish a session with the RMC subsystem daemon on a managed node and fails, returning an error to the application.

Related reference:

"mc_start_session" on page 206
This subroutine establishes a session with the RMC subsystem.

"mc_timed_start_session" on page 212
This subroutine establishes a session with the RMC subsystem.

RMC API base programming model

The base programming model of the RMC API is *command-response* with *notification of events*.

These terms have the following meanings:

- *command-response* means that the application calls subroutines which issue commands to the RMC subsystem, and the RMC subsystem returns responses to the application.
- *notification of events* means that, if the application has registered events with the RMC subsystem (in other words, asked the RMC subsystem to monitor changes in attribute values for events of interest), the RMC subsystem will notify the application when such events occur.

Sending commands to the RMC subsystem:

A command is sent to the RMC subsystem by invoking an RMC API subroutine.

The command issued to the RMC subsystem could be:

- a request for information (such as a list of resources of a resource class, or the dynamic or persistent attribute values of a resource or resource class),
- an operation (such as defining a new resource instance, setting a persistent resource attribute, bringing a resource online, taking a resource offline, or invoking a resource action).
- an event registration (a request to monitor changes in attribute values for events of interest)

Most of the RMC API command interfaces consist of four related subroutines that issue the same command action to the RMC subsystem, but differ in how the command is sent to the RMC subsystem and how responses are returned to the application. When sending a command to the RMC subsystem, the application can use either a *blocking* or a *nonblocking* subroutine.

- *blocking* subroutines do not return until the command is completely processed by the RMC subsystem.
- *nonblocking* subroutines return immediately after adding the command to a *command group*.

  A *command group* is an area of memory that the application can allocate by calling the **mc_start_cmd_grp** subroutine. If the memory is successfully allocated, the **mc_start_cmd_grp** subroutine returns a command group handle to the application. The application can use this handle in subsequent subroutine calls to identify the command group to which a command should be added.

  It is important to understand that a nonblocking subroutine does not send a command to the RMC subsystem, but merely adds it to the command group. The command group is later sent as a single unit to the RMC subsystem. To send all the commands in a command group to the RMC subsystem, the application calls either the **mc_send_cmd_grp** or **mc_send_cmd_grp_wait** subroutine.

– the **mc_send_cmd_grp** subroutine is nonblocking. It sends the command group to the RMC subsystem and returns immediately.
– the **mc_send_cmd_grp_wait** subroutine is blocking. It sends the command group to the RMC subsystem and blocks execution until all of the command group's commands complete.

The following two figures illustrate the difference between the blocking and nonblocking subroutines.

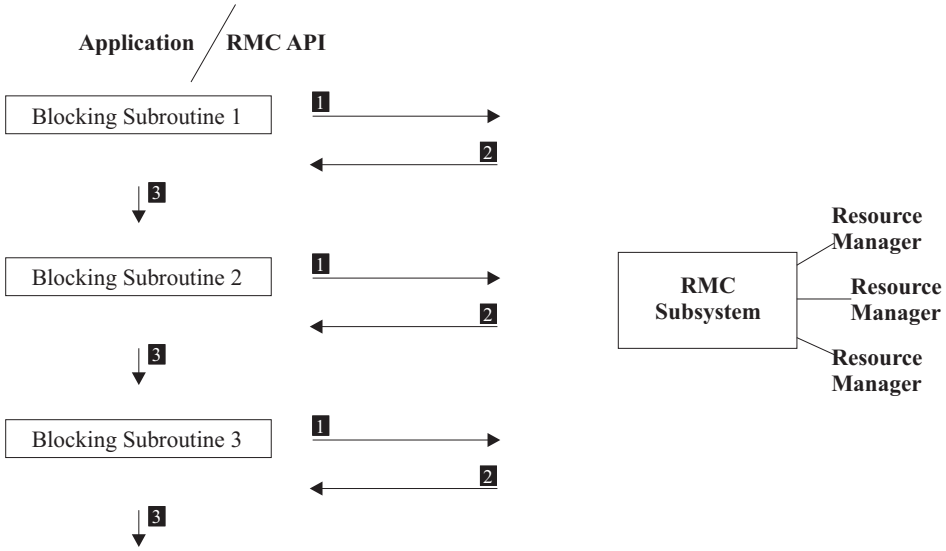


Figure 1. Application using blocking subroutine calls. The application calls a blocking RMC API subroutine which (**1**) issues a command to the RMC subsystem. The RMC subsystem performs the operation and (**2**) sends responses back to the RMC API which forwards them to the application. Only then does execution of the application resume (**3**).

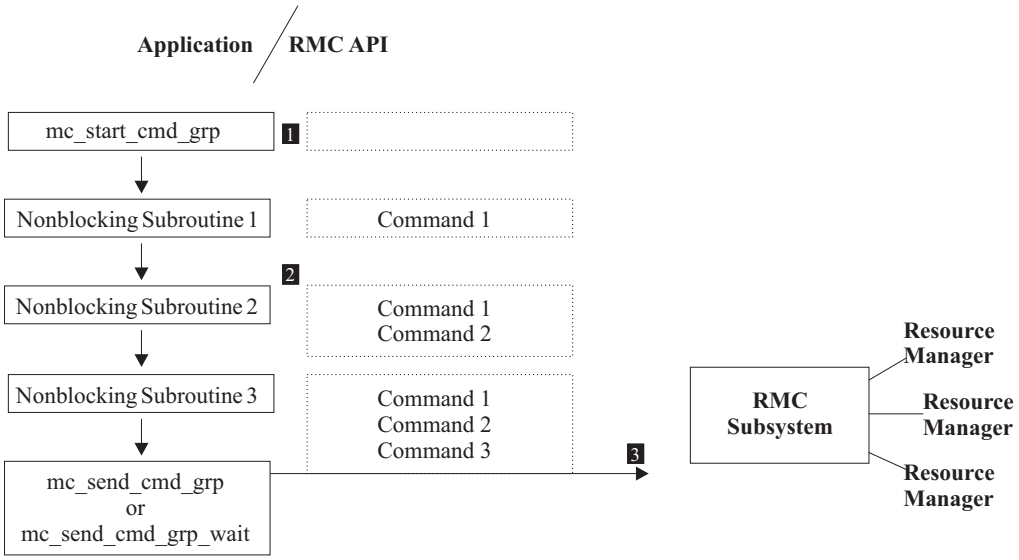


Figure 2. Application using nonblocking subroutine calls. The application calls the **mc_start_cmd_grp** subroutine (**1**) to allocate memory for a command group. Subsequent calls to nonblocking subroutines (**2**) add commands to the command group. When the application is ready, it calls either the **mc_send_cmd_grp** or **mc_send_cmd_grp_wait** subroutine (**3**) to send the commands in the command group as a single unit to the RMC subsystem.

Ordered command group:

An *ordered command group* is a command group whose commands will be processed by the RMC subsystem in the order in which they were added to the group.

In addition, the application can specify whether or not processing of the ordered command group should stop with any command that results in an error. Each command in the ordered command group must specify the same resource or set of resources, and may not contain commands directed to a resource class. If more than one resource is specified then the ordering applies to the processing of the group against each individual resource. Only commands that are processed directly by a resource manager may be placed into an ordered command group. Each command description indicates whether or not the command may be placed into an ordered command group.

An ordered command group may also be *non-interleaved*. This means that no command, not in the command group, can be processed while the command group is being processed.

Related information:

"RMC API subroutines" on page 40
The RMC API is a library of subroutines and supporting data types, written in C.

Returning responses to the application:

When the application invokes an RMC API subroutine to send a command to the RMC subsystem, a successful return value from the subroutine indicates only that the command was successfully sent to the RMC subsystem or added to a command group.

The actual response to the command is returned by the RMC API in the form of one or more response structures.

Table 2. Response structures returned by the RMC API

If the command sent to the RMC API is:	Such as:	The Response Structure will contain:
a request for information	a request for: • a list of resources of a resource class • the dynamic or persistent attribute values of a resource or resource class	• the requested information • a return value indicating whether the command was successful or if some or all of the requested information could not be returned
an operation on a resource	• setting a resource persistent attribute value • bringing a node online • taking a node offline • invoking a resource action	• the unique resource handle that identifies the resource that was the target of the command • a return value indicating the command was successful or an error indicating a problem in performing the command operation
an operation on a resource class	• setting a resource class persistent attribute value • invoking a resource class action • defining a new resource instance of the resource class • undefining a resource instance of the resource class	• the name of the resource class that was the target of the command • a return value indicating the command was successful or an error indicating a problem in performing the command operation
a request to monitor (or to stop monitoring) for an event	A request to: • register a resource event • register a resource class event • unregister a resource event or a resource class event	• a registration ID that uniquely identifies the event • a return value indicating that the event was successfully registered or unregistered or an error

The preceding table lists only the general information returned by each type of command, and does not necessarily list all the information returned by each individual command.

A single command can result in multiple response structures being returned. For example, a command can be targeted to multiple resources, in which case a separate response is returned for each resource.

Most of the RMC API command interfaces consist of four related subroutines that issue the same command action to the RMC subsystem, but differ on how the command is sent to the RMC subsystem and how the response is returned to the application. Depending on the particular subroutine used, the application can determine if the response structures are returned using *pointer response* or *callback response*. When using:

- *pointer response*, the application specifies, using a parameter of the subroutine, a location where a response pointer should be returned. When the command is complete, the location specified will contain a pointer to the response, or (if the command results in multiple responses) a pointer to an array of responses.
- *callback response*, the application specifies a callback routine using a parameter of the subroutine. When a response structure is received by the RMC API, the RMC API invokes the callback and passes it the response structure. If the command results in multiple responses from the RMC subsystem, the RMC API invokes the callback for each response.

Once a response or response array has been given to the application, the application may hold the data as long as necessary, even subsequent to the return of any callback that received the data. When the response or response array is no longer needed, the application must free it by calling the **mc_free_response** subroutine. Note that a response array may be returned even under certain error conditions.

The callback response approach can be advantageous when multiple responses are expected in reply to a command. When using callback response, the application does not have to wait for the entire command to complete (as it does using the pointer response approach). Instead, as the responses arrive, the callback is invoked to process the data.

The advantage of the pointer response approach is that it supports a simpler programming style, since the logic to issue a command and process the response can be in the same function.

The nonblocking subroutines return immediately after adding a command to a command group which the application later sends as a unit to the RMC subsystem by calling the **mc_send_cmd_grp** or **mc_send_cmd_grp_wait** subroutines. Responses are returned for commands in the command group the same way they are for commands issued by calling blocking subroutines.

If a command group is sent to the RMC subsystem using the nonblocking subroutine **mc_send_cmd_grp**, the RMC API will invoke an application-specified completion callback routine once all commands in the command group have been fully processed. Be aware that the application must provide one or more threads to the RMC API so in can invoke the completion callback routine for the command group as well as any callback routines for commands in the command group. The application provides a thread to the RMC API using the **mc_dispatch** subroutine.

For blocking subroutines (or nonblocking subroutines that were sent to the RMC subsystem using the blocking subroutine **mc_send_cmd_grp_wait**) it is not necessary to provide threads to the RMC API for processing response callbacks in a separate thread. For blocking subroutines, the RMC API uses the blocked thread to process the response callback. However, the application could still use the **mc_dispatch** subroutine to provide additional threads to the RMC API (which will attempt to parallelize callback exection if there are threads available).

Related information:

"RMC API subroutines" on page 40
The RMC API is a library of subroutines and supporting data types, written in C.

Illustrations of pointer response:

The following figures illustrate how responses are returned to an application using pointer response.

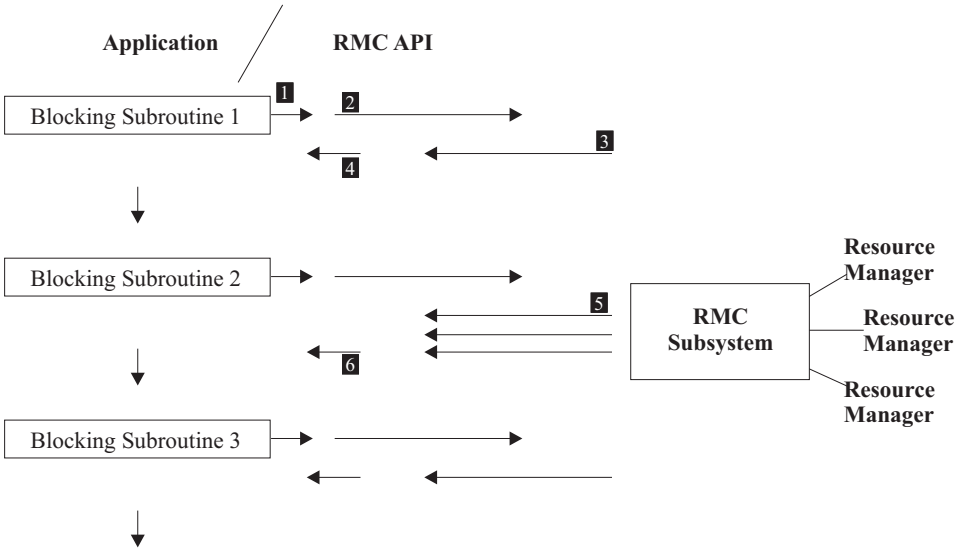


Figure 3. Application using blocking subroutine calls and pointer response. The application (**1**) calls a blocking subroutine that uses pointer response. A parameter of the subroutine is a pointer to a location in which the RMC API will return a pointer to the response. The RMC API (**2**) issues the command to the RMC subsystem. The RMC subsystem processes the command and (**3**) returns a Response Structure to the RMC API which (**4**) returns a pointer to the response at the location specified in the subroutine call.

If the command sent to the RMC subsystem results in (**5**) multiple Response Structures being returned to the RMC API, note that the RMC API waits for all responses to be returned from the RMC subsystem before (**6**) returning a pointer to an array of responses at the location specified in the subroutine call.

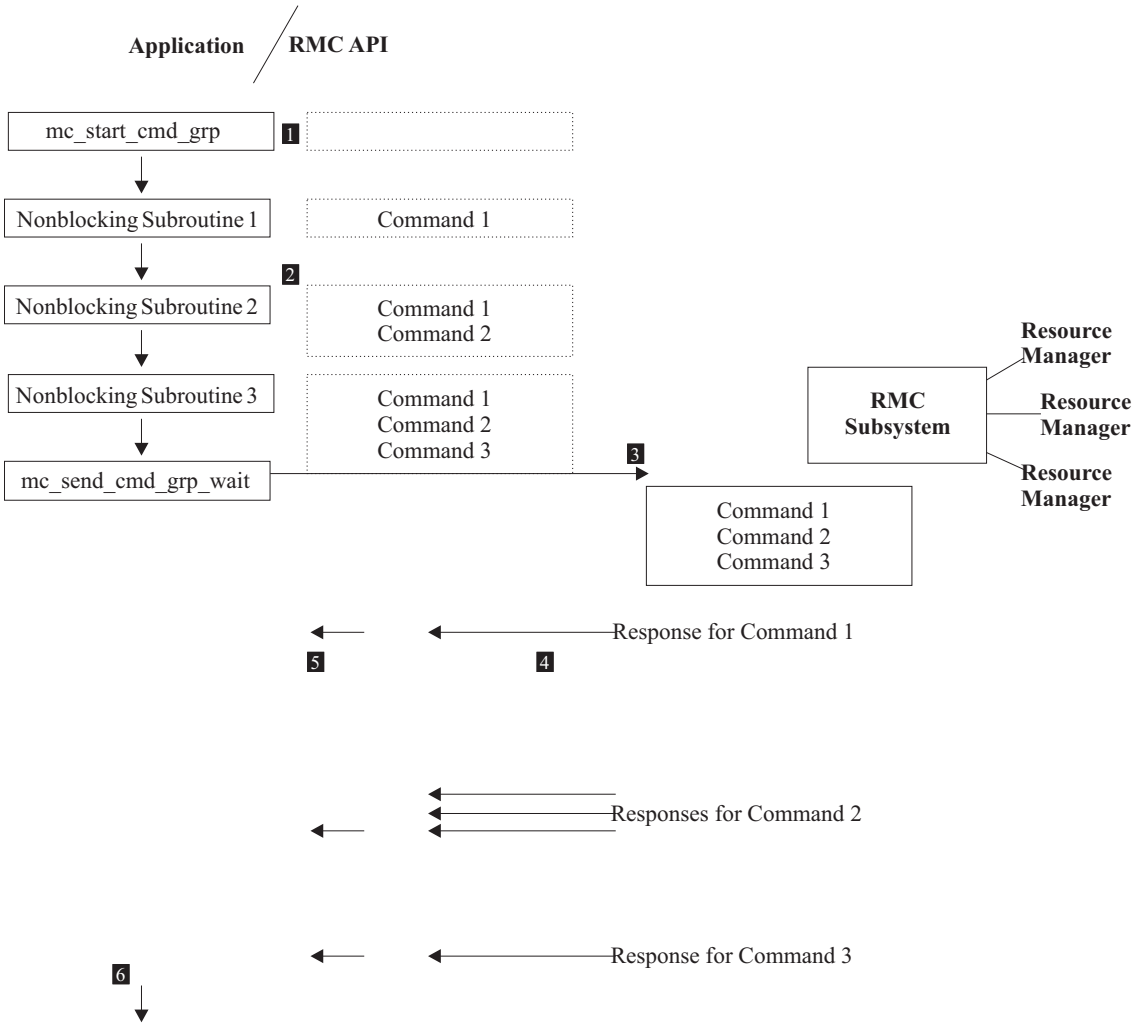


Figure 4. Application using nonblocking subroutine calls and pointer response (command group sent using blocking subroutine **mc_send_cmd_grp_wait***)..* The application (**1**) calls the **mc_start_cmd_grp** subroutine to create a command group. The application calls a number of subroutines (**2**) to add commands to the command group. A parameter on each of these subroutines is a pointer to a location in which the RMC API will return a pointer to the response. The application (**3**) sends the commands in the command group as a single unit to the RMC subsystem. The RMC subsystem processes the commands in the command group and (**4**) returns response structures to the RMC API. As the response structures are returned, the RMC API (**5**) returns pointers to the responses at the locations specified in the subroutine calls. If a command in the command group results in multiple Response Structures being returned to the RMC API, the RMC API waits until all responses are returned before returning a pointer to an array of responses at the location specified by the subroutine call. Since the **mc_send_cmd_grp_wait** subroutine is blocking, it does not return control to the application until (**6**) responses have been received and processed for all commands in the command group.

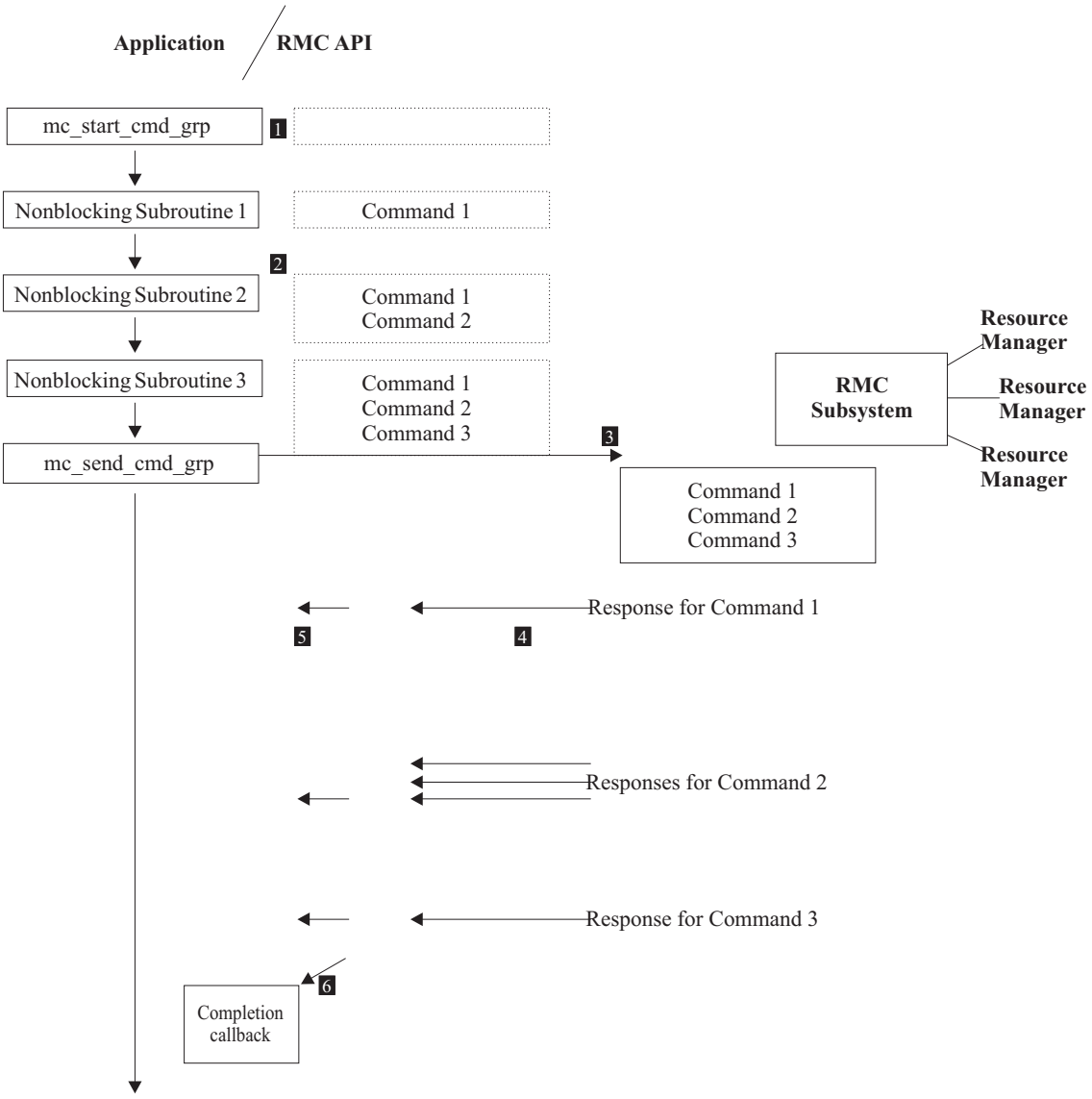


Figure 5. Application using nonblocking subroutine calls and pointer response (command group sent using nonblocking subroutine **mc_send_cmd_grp***).*. The application (**1**) calls the **mc_start_cmd_grp** subroutine to create a command group. The application calls a number of subroutines to (**2**) add commands to the command group. The application (**3**) sends the commands in the command group as a single unit to the RMC subsystem. The RMC subsystem processes the commands in the command group and (**4**) returns Response Structures to the RMC API. As the Response Structures are returned, the RMC API returns pointers to the responses at the locations specified in the subroutine calls. If a command in the command group results in multiple Response Structures being returned to the RMC API, the RMC API waits until all responses are returned before returning a pointer to an array of responses at the location specified by the subroutine call. Since the **mc_send_cmd_group** subroutine is nonblocking, the application is able to perform other work while the responses are being received and processed by the RMC API. When responses have been received and processed for all commands in the command group, the RMC API informs the application by invoking the completion callback routine that was specified by the **mc_send_cmd_group** subroutine call. The completion callback runs in a separate thread that the application provided by calling the **mc_dispatch** subroutine.

Illustrations of callback response:

The following figures illustrate how responses are returned to the application using callback response.

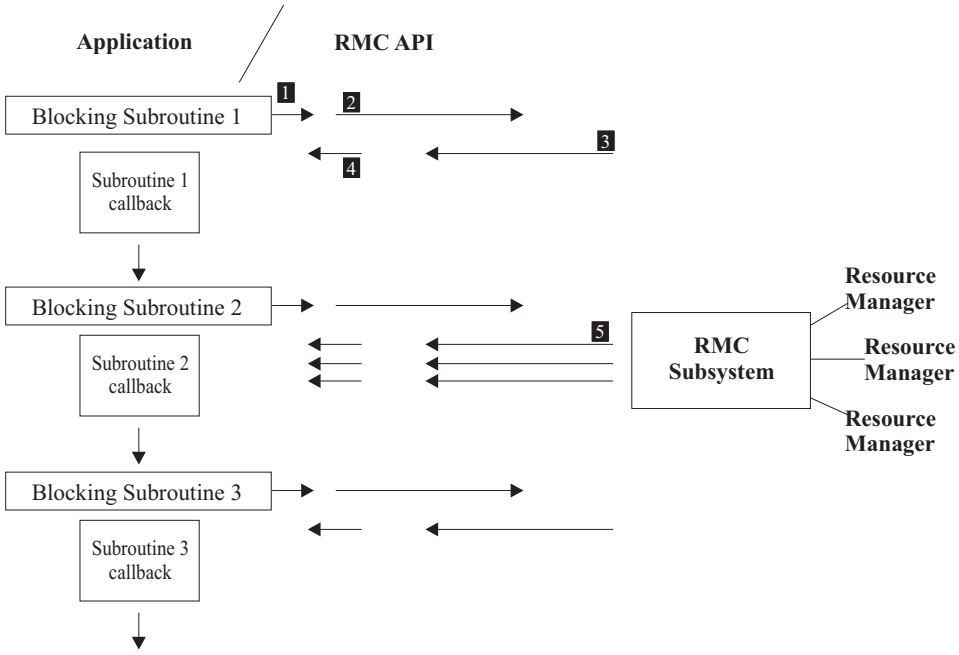


Figure 6. Application using blocking subroutines and callback response. In this illustration, we are assuming that no additional threads have been supplied to the RMC API. In this case, the RMC API will use the blocked thread to invoke callback routines. The application (**1**) calls a blocking subroutine that uses callback response. A parameter of the subroutine identifies the callback routine that the RMC API should invoke to return responses to the application. The RMC API (**2**) issues the command to the RMC subsystem. The RMC subsystem processes the command and (**3**) returns a Response Structure to the RMC API which (**4**) invokes the callback routine specified in the subroutine call for processing the response. The callback is processed in the application thread.

If the command sent to the RMC subsystem results in (**5**) multiple response structures being returned by the RMC API, note that the RMC API invokes the callback routine to process each response as it arrives.

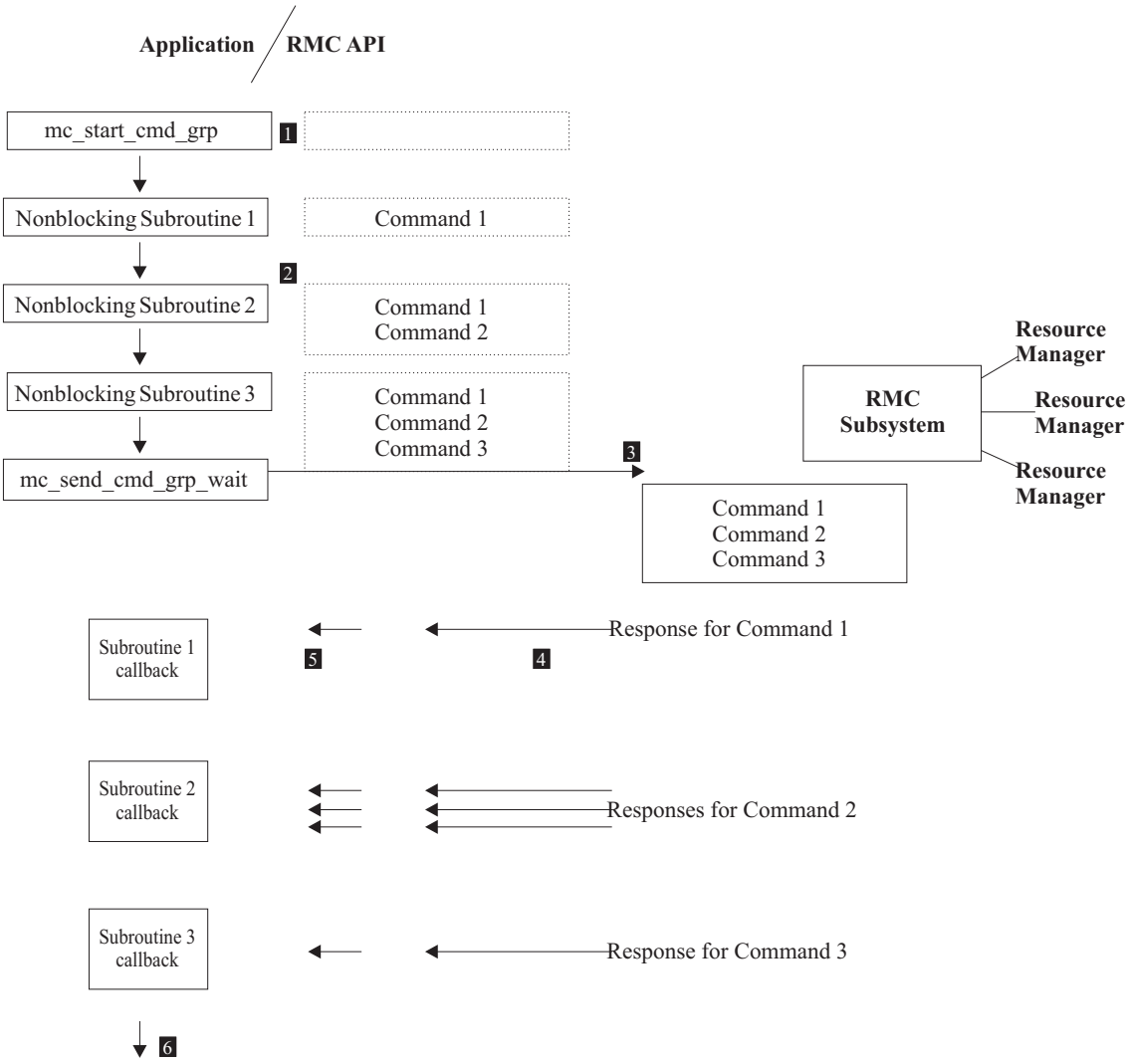


Figure 7. Cluster applicaiton using nonblocking subroutine calls and callback response (command group sent using blocking subroutine **mc_send_cmd_grp_wait***).* In this illustration, we are assuming that no additional threads have been supplied to the RMC API. In this case, the RMC API will use the thread blocked by the **mc_send_cdm_grp_wait** subroutine to invoke callback routines. The application (**1**) calls the **mc_start_cmd_grp** subroutine to create a command group. The application calls a number of subroutines to (**2**) add commands to the command group. A parameter on each of these subroutines identifies the callback routine that the RMC API should invoke to return responses to the application. The application (**3**) sends the commands in the command group as a single unit to the RMC subsystem. The RMC subsystem processes the commands in the command group and (**4**) returns response structures to the RMC API. As each response structure arrives, the RMC API (**5**) invokes the appropriate response callback routine and passes it the response structure. If the command sent to the RMC subsystem results in multiple response structures being returned to the RMC API, note that the RMC API invokes the callback routine to process each response as it arrives. Since the **mc_send_cmd_grp_wait** subroutine is blocking, all callback routines must complete before (**6**) control is returned to the application thread.

In the preceding two figures, the callback routines are all processed in the blocked application thread. To parallelize execution of the callbacks, however, the application can supply one or more additional threads that the RMC API can use to invoke callback routines. The application can supply these additional threads by calling the **mc_dispatch** routine.

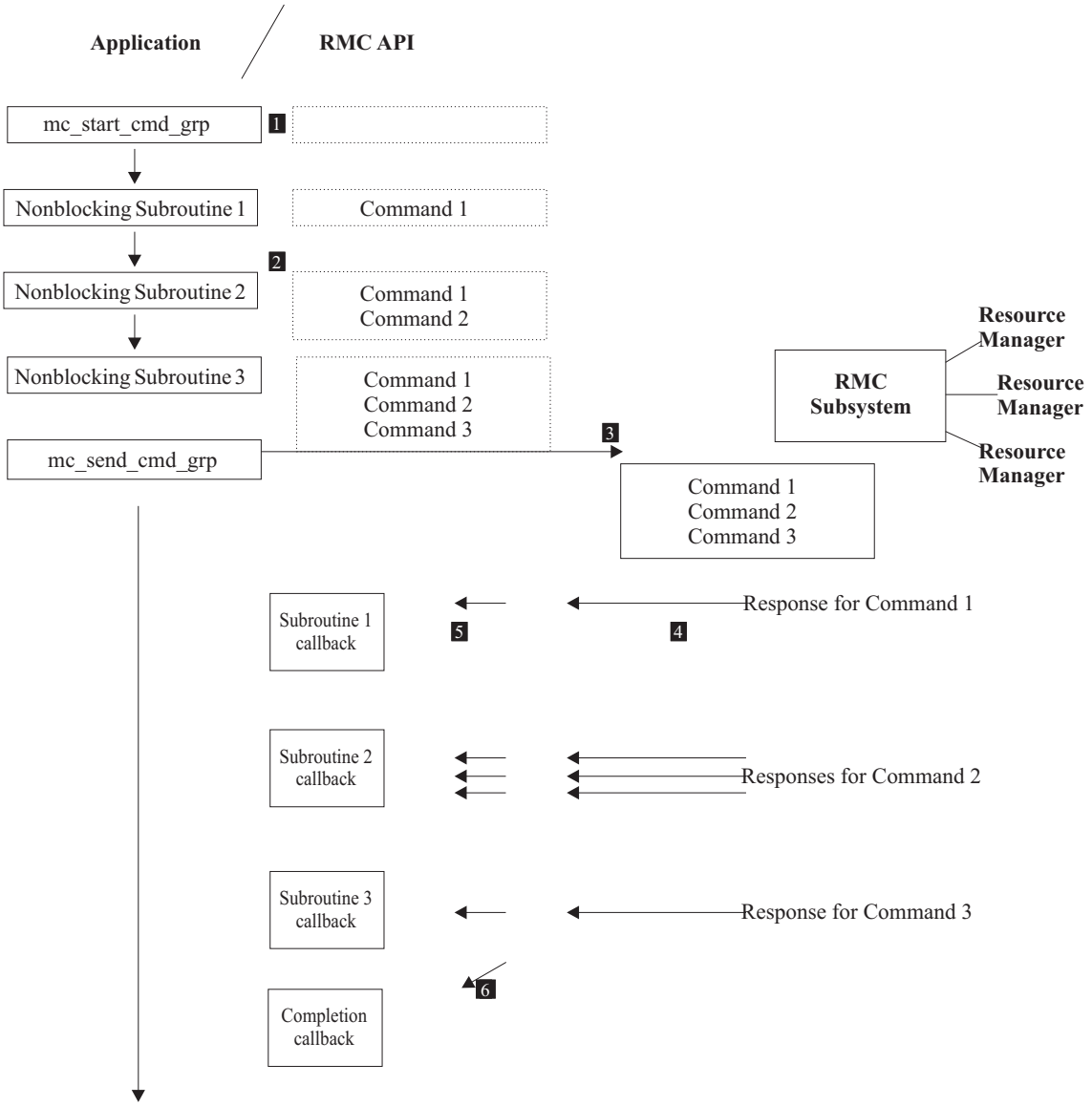


Figure 8. Application using nonblocking subroutine calls and callback response (command group sent using nonblocking subroutine **mc_send_cmd_grp**)*.*. In this illustration, we are assuming that the application has created an additional thread and assigned it to the RMC API using the mc_dispatch subroutine. The application (**1**) calls the **mc_start_cmd_grp** subroutine to create a command group. The application calls a number of subroutines to (**2**) add commands to the command group. A parameter on each of these subroutines identifies the callback routine that the RMC API should invoke to return responses to the application. The application (**3**) sends the commands in the command group as a single unit to the RMC subsystem, using the **mc_send_cmd_grp** subroutine. A parameter of the **mc_send_cmd_grp** subroutine identifies a callback routine that the RMC API will invoke after all commands in the command group are processed. The RMC subsystem processes the commands in the command group and (**4**) returns response structures to the RMC API. As each response structure arrives, the RMC API (**5**) invokes the appropriate response callback routine and passes it the response structure. If the command sent to the RMC subsystem results in multiple response structures being returned to the RMC API, note that the RMC API invokes the callback routine to process each response as it arrives. Since the **mc_send_cmd_grp** subroutine is nonblocking, the application is able to perform other work while the response callbacks run in a separate thread. When all response callbacks have returned, the RMC API (**6**) invokes the completion callback (specified by the **mc_send_cmd_grp** subroutine) to notify the application that all commands in the command group have been processed.

Related concepts:

"Providing the RMC API with one or more threads" on page 15
The application must provide the RMC API with a thread for processing responses and event notifications under two conditions.

Registering the application for event notifications:

A set of RMC subroutines enables an application to register with the RMC subsystem for event notifications.

An event is a change in an attribute that is of interest to the application.

To register for event notifications, the application calls a particular RMC subroutine and provides it with an *event expression*. In most cases, an event expression consists of one or more attribute names (usually dynamic attribute names), one or more mathematical comparison symbols, and one or more constants that together describe the event of interest. The RMC subsystem will evaluate the event expression when the attribute values are observed. The RMC subsystem observes, or obtains, the values of attributes specified in event expressions either at periodic intervals or whenever a new value is supplied by the associated resource manager, depending on the variable type of the attribute.

When the RMC subsystem evaluates an event expression, it will trigger an event if the expression evaluates to True. For example, resources of the **IBM.FileSystem** resource class use the dynamic attribute **PercentTotUsed** to represent the percentage of space used in a file system. The following event expression, once registered with the RMC subsystem for a resource, would cause the RMC subsystem to notify the application if the particular file system resource is over 90 percent full.

```
PercentTotUsed > 90
```

As already stated, each event expression refers to a particular attribute value. This is usually a dynamic attribute, since such attributes represent changing characteristics of a resource. Once the application registers with the RMC subsystem for event notifications based on a particular event expression, the RMC subsystem will observe the value of the attribute and evaluate the event expression. If the event expression evaluates to True, the RMC subsystem will notify the application.

When observing attribute values to evaluate event expressions, RMC remembers the previously observed value of the attribute. If the event expression suffixes the attribute name with **@P**, this represents the previously observed value of the attribute. For example, resources of the **IBM.Host** resource class have the dynamic attribute **ProcRunQueue**, which indicates the average number of processes that are waiting for the processor. The following event expression, once registered with the RMC subsystem for an **IBM.Host** resource, would cause the RMC subsystem to notify the application if the average number of processes in the run queue has increased by 50 percent or more between observations.

```
(ProcRunQueue - ProcRunQueue@P) >= (ProcRunQueue@P * 0.5)
```

Although most event expressions consist of attribute names, one or more mathematical comparison symbols, and one or more constants, this is not True if the attribute type is of variable type **Quantum**. An attribute of variable type **Quantum** signifies a change, but has no value associated with it. To register for events involving a single attribute of type **Quantum**, the event expression must consist of the attribute name only.

For example, the **IBM.FileSystem** resource class has the dynamic attribute **ResourceDefined** that is of type **Quantum**. Although it has no value, it is asserted whenever a resource of the class is created. The following event expression, once registered with the RMC subsystem for the **IBM.FileSystem** resource class, would cause the RMC subsystem to notify the application when a file system is created.

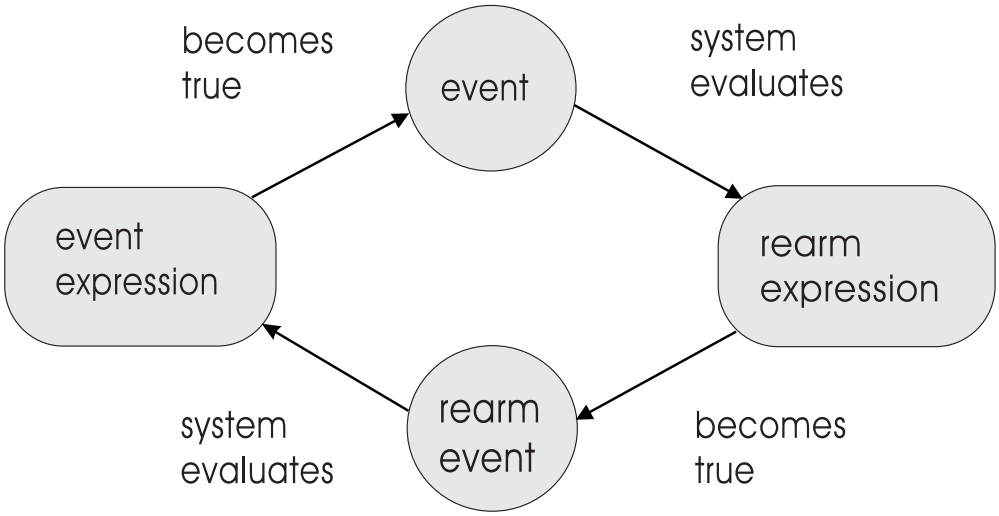
```
ResourceDefined
```

To register for events involving more than one attribute name, where at least one attribute name is an attribute of type **Quantum**, the attributes of type **Quantum** can only be used as operands of boolean operators. For example, the **IBM.FileSystem** resource class has the dynamic resource attributes **ConfigChanged** and **PercentTotused**. The attribute **ConfigChanged** is of type **Quantum**, while **PercentTotused** is not. The following event expression would result in an event notification if the file system is more than 90% full or the persistent attributes of the resource are modified:

```
PercentTotUsed > 90 || ConfigChanged
```

When registering for event notifications, a cluster applicaiton can optionally specify a *rearm event expression*. Like most event expressions, a rearm event expression consists of one or more attribute names, one or more mathematical comparison symbol, and one or more constants. The attributes in the rearm event expression do not need to be the same as the attributes that are used in the event expression. If a rearm event expression is specified, then the RMC will stop evaluating the event expression once it tests True, and instead will evaluate the rearm event expression until it tests True. Once the rearm event expression tests True, the RMC subsystem will once again evaluate the event expression.

The following diagram illustrates the cycle of event expression / rearm event expression evaluation.



To understand why the application might want to specify a rearm event expression when registering for event notifications, consider the event registration that specifies the following event expression on an IBM.FileSystem resource.

```
PercentTotUsed > 90
```

For this event expression, the RMC subsystem will periodically observe the value of the PercentTotUsed attribute for the resource and will evaluate the event expression. If the event expression evaluates to True, the RMC subsystem will notify the application. If there is no rearm event expression, and the event expression still tests True the next time the RMC subsystem evaluates it, the RMC subsystem will again notify the application. The RMC subsystem will continue to notify the application each time it evaluates the event expression until it tests false. In some cases, the application may want these repeated notifications. In other situations, however, it might be preferable to be notified once until the problem has been solved and only then "rearm" the event expression. The rearm event expression is a way of telling the RMC subsystem when the event expression should be evaluated again. In our example, say the application also provides the following rearm event expression when registering for the event notifications.

```
PercentTotUsed < 75
```

If this rearm event expression is specified, then once the event expression `PercentTotUsed > 90` evaluates to True, the RMC subsystem will stop evaluating it, and will instead evaluate the rearm event expression until it tests True. When calling the RMC subroutine to register for event notifications, the application can specify whether or not it also wants to be notified when the rearm event expression evaluates to True. Whether the application chooses to be notified or not, once the rearm event expression tests True, the RMC subsystem will toggle to once again evaluate the event expression.

Related concepts:

"Targeting resources for a command" on page 18
Commands issued to the RMC subsystem using the RMC API are usually targeted to one or more resources of a resource class, or one resource class.

"Data types and operators supported in expressions" on page 19
An expression in RMC is similar to a C language statement or the WHERE clause of an SQL query. It is composed of variables, operators and constants. The C and SQL syntax styles may be intermixed within a single expression.

"Data types and operators supported in expressions" on page 19
An expression in RMC is similar to a C language statement or the WHERE clause of an SQL query. It is composed of variables, operators and constants. The C and SQL syntax styles may be intermixed within a single expression.

Related information:

"RMC API subroutines" on page 40
The RMC API is a library of subroutines and supporting data types, written in C.

Notifying the application of events:

A set of RMC subroutines enables an application to register with the RMC subsystem for event notifications.

These subroutines are summarized in "Monitoring command interfaces" on page 32 and described fully in "RMC API subroutines" on page 40. When calling any of these subroutines, the application identifies a callback routine. When the event expression (and optionally, the rearm event expression) evaluates to True, the RMC API will invoke the specified callback, passing it an event notification structure. The event notification structure will contain information about the event, including:

- A timestamp that indicates when the event occurred.
- For a resource event, the resource handle of the resource whose state change resulted in the generation of the event. For a resource class event, the name of the resource class whose state change resulted in the generation of the event.
- The values of the attributes that are used in the event expression.

When calling the particular RMC subroutine to register an event, the application can also specify that the RMC subsystem should return additional persistent attribute information in the event notification structure.

Once an event notification has been given to the application, the application may hold the data as long as necessary, even subsequent to the return of any callback that received the data. When the event notification is no longer needed, the application must free it by calling the **mc_free_response** subroutine. Note that an event notification may be returned even under certain error conditions.

Once the application has registered an event with the RMC subsystem, event notification will continue until the application unregisters the event using one of the **mc_unreg_event_*** subroutines.

The RMC API invokes the event callback routine using a thread that the application must provide by calling the **mc_dispatch** subroutine. For more information on the **mc_dispatch** subroutine and supplying threads to the RMC API, see "Providing the RMC API with one or more threads."

Providing the RMC API with one or more threads:

The application must provide the RMC API with a thread for processing responses and event notifications under two conditions.

The application must provide the RMC API with a thread if:

- it sends a command group to the RMC subsystem using the nonblocking subroutine **mc_send_cmd_grp**

- it registers for one or more event notifications

In each of these cases, the RMC API needs a thread to invoke callback routines. In the case of a command group sent to the RMC subsystem using the nonblocking subroutine **mc_send_cmd_grp**, the RMC API needs a thread in order to invoke the completion callback routine for the command group as well as any callback routines for commands in the command group. In the case of one or more registered events, the RMC API needs a thread in order to invoke the event notification callback routines.

To provide a thread to the RMC API, the application calls the **mc_dispatch** subroutine (as described in "mc_dispatch" on page 62). How the application uses the **mc_dispatch** subroutine depends on whether it is a single-threaded application or a multi-threaded application.

Related concepts:

"Illustrations of callback response" on page 9
The following figures illustrate how responses are returned to the application using callback response.

Related reference:

"Command management interfaces" on page 32
The following table summarizes the RMC API command management interfaces.

A single-threaded application surrendering its thread to the RMC API:

If an application is single threaded (in other words, does not create any threads) and either sends a command group using the nonblocking **mc_send_cmd_grp** subroutine or else registers for event notifications, it will need to surrender its thread to the RMC API one or more times so that the RMC API can process the necessary response or event notification callback(s).

To provide the single application thread to the RMC API, the application calls the **mc_dispatch** subroutine, specifying the subroutine option MC_DISPATCH_OPTS_WAIT. Calling the **mc_dispatch** subroutine with this option causes the application thread to block until the RMC API has used it to process one response or event notification callback routine. Once one response or event notification callback routine is processed, the subroutine returns.

Since the **mc_dispatch** subroutine with the option MC_DISPATCH_OPTS_WAIT returns after processing one response or event notification callback routine, the application will need to call the **mc_dispatch** subroutine for each response or event notification. A difficulty in doing this, however, is it is often unclear how many responses or event notifications to expect.

- In the case of a command group sent using the nonblocking subroutine **mc_send_cmd_grp**, the RMC API will need the thread to process any response callbacks for commands in the command group (some of which could be receiving multiple responses), as well as the command group's completion callback. The invocation of the completion callback, however, indicates that all commands in the command group have been processed.
- In the case of event notification callbacks, the RMC API will need a thread to process these whenever the event expression evaluates to TRUE; there is no way for the application to determine how often this will happen. Only when it unregisters the event can the application be certain there will be no further event notifications.

Another difficulty in using the **mc_dispatch** subroutine with the option MC_DISPATCH_OPTS_WAIT, is that it blocks the application thread until a response or event notification callback has been processed. If called too soon, before there is a response or event notification to process, the application thread is blocked unnecessarily.

To prevent the **mc_dispatch** subroutine from blocking unnecessarily, the application can call the **mc_get_descriptor** subroutine to obtain a descriptor that is made ready to read whenever the RMC API has received a response or event notification from the RMC subsystem and needs the application thread to process the appropriate callback routine. The application can use the descriptor in a **select** or **poll**

system call to determine when it needs to call the **mc_dispatch** subroutine. When the **mc_dispatch** subroutine returns, and another response or event notification is expected, the application can again use the descriptor in a **select** or **poll** system call.

A multi-threaded application providing threads to the RMC API:

When the application either sends a command group using the nonblocking **mc_send_cmd_grp** subroutine or else registers for event notifications, it will need to make at least one thread available to the RMC API to process the necessary response or event notification callback(s).

An application can do this by creating a thread and having it call the **mc_dispatch** subroutine, specifying the subroutine option MC_DISPATCH_OPTS_ASSIGN. When the **mc_dispatch** subroutine is called with this option, it does not return. Instead the thread is kept by the RMC API to process response and event notification callbacks. Multiple threads can be provided to the RMC API in this way. When the RMC API receives multiple responses or event notifications form the RMC subsystem, it will invoke as many callbacks in parallel as it has threads available, subject to the following rules:

- Multiple responses to the commands in the command group from the same resource or resource class are processed one at a time. Responses to the commands in the command group from different resources or resource classes can be processed in parallel. Responses not generated by a resource or resource class can be processed in parallel. The following subroutines have responses generated by resources of resource classes:
  - **mc_enumerate_resources_***
  - **mc_enumerate_permitted_rsrcs_***
  - **mc_query_p_select_***
  - **mc_query_p_handle_***
  - **mc_class_query_p_***
  - **mc_define_resource_***
  - **mc_undefine_resource_***
  - **mc_refresh_config_***
  - **mc_set_select_***
  - **mc_set_handle_***
  - **mc_class_set_***
  - **mc_invoke_action_***
  - **mc_invoke_class_action_***
  - **mc_online_***
  - **mc_offline_***
  - **mc_reset_***

  The **mc_validate_rsrc_hndl_*** subroutines also have responses generated from a resource class. However, the response serialization rules do no apply to them.
- Event notifications for the same event registration and from the same resource or resource class are processed one at a time.
- Event notifications for the same event registration, but from different resources or resource classes, can be processed in parallel
- Event notifications for different event registrations may be processed in parallel, regardless of the resource class from which they come.

Cancelling threads executing RMC API subroutines:

The RMC API does the necessary internal cleanup when threads are cancelled. A thread executing an RMC API subroutine can be safely cancelled if the thread's cancelability state type is deferred.

When a thread with a deferred cancelability state type is the target of a cancellation request, the request is acted upon when a thread cancellation point is reached. The blocking subroutines all contain thread cancellation points as do the **mc_start_session**, **mc_timed_start_session**, **mc_send_cmd_grp_wait**, and **mc_dispatch** subroutines.

- When a thread executing a blocking subroutine is cancelled, the command will have been sent to the RMC subsystem. None, some, or all of the command group responses will have been delivered to the application. If the subroutine was an event registration command, the RMC API unregisters the event. Some event notifications may be delivered to the application before the event is unregistered.
- When a thread executing the **mc_start_session** or **mc_timed_start_session** subroutine is cancelled, the session with the RMC subsystem is not established.
- When a thread executing the **mc_send_cmd_grp_wait** subroutine is cancelled, the commands in the command group will have been sent to the RMC subsystem. None, some, or all of the command group responses will have been delivered to the application. If the command group includes event registration commands, the RMC API unregisters the events. Some event notifications may be delivered to the application before the events are unregistered.

An event notification callback, command response callback, or command group completion callback may include cancellation points. When developing such callback routines for an environment in which threads are cancelled, it may be necessary to develop thread cancellation cleanup handlers. A thread cancellation cleanup handler might call the **mc_free_response** subroutine which frees the storage used by a response or event notification structure.

Targeting resources for a command

Commands issued to the RMC subsystem using the RMC API are usually targeted to one or more resources of a resource class, or one resource class.

While a resource class is always identified by its name, there are, depending on the particular subroutine, two ways to identify target resources for a command.

- When calling some subroutines, an application can identify a single resource by its *resource handle*. A resource handle is returned in the response structure for many RMC API subroutines (including the **mc_define_resource_*** subroutines for defining a resource, and the **mc_enumerate_resources_*** and **mc_enumerate_permitted_rsrcs_*** subroutines for listing resources of a resource class). A resource handle is also provided in the resource event notification structures.
- When calling some subroutines, an application can identify one or more resources of a particular resource class using *attribute selection*. To identify one or more resources using attribute selection, the application specifies a resource class name and a *selection string* that identifies a set of resources of the resource class. A selection string specifies a set of persistent attributes and associated attribute values. Resources of the resource class whose persistent attribute values match the values specified in the selection string are identified as the target resources for the command.

For example, when an application registers for event notifications, it can monitor an attribute of a resource class or an attribute of one or more resources of a resource class. There are three RMC API interfaces (each including four subroutine variations to accommodate the blocking/nonblocking, callback response/pointer response variations). These three interfaces are:

- the **mc_reg_class_event_*** subroutines. All four variations of this interface register a resource class event with the RMC subsystem. The application identifies the resource class by its name.
- the **mc_reg_event_handle_*** subroutines. All four variations of this interface register a resource event with the RMC subsystem. The application identifies a single resource using a resource handle.
- the **mc_reg_event_select_*** subroutines. All four variations of this interface register a resource event with the RMC subsystem. The application identifies one ore more resources of a resource class by:
  - specifying the resource class name
  - providing a selection string to specify the target resource(s). For example, say the application is registering for events based on a dynamic attribute of the IBM.Host resource class (which represents a host machine that is running a single copy of an operating system). However, the application only

wants to register the event for host machines that are running the AIX operating system. The following selection string uses the OSName persistent attribute (which indicates the operating system running on the host machine) to target just those resources of interest.

```
OSName == 'AIX'
```

Even if the selection string does not match any resources, the event registration will still succeed. In such a case, the RMC subsystem will monitor the specified resource class and if, at some future time, resources then match the selection string, those resources will automatically be added to the target list. In general, when a selection string is specified for an event registration, the RMC subsystem continually monitors the resource class to see if additional resources match the selection string or if resources no longer match the selection string. Such resources are then added to the target list or, optionally, removed from the target list.

Related concepts:

"Registering the application for event notifications" on page 13
A set of RMC subroutines enables an application to register with the RMC subsystem for event notifications.

Data types and operators supported in expressions

An expression in RMC is similar to a C language statement or the WHERE clause of an SQL query. It is composed of variables, operators and constants. The C and SQL syntax styles may be intermixed within a single expression.

There are two types of expressions in RMC. One type is the event expression or rearm event expressions. The other type is the selection string expressions.

Related concepts:

"Registering the application for event notifications" on page 13
A set of RMC subroutines enables an application to register with the RMC subsystem for event notifications.

"Qualifiers" on page 27
A mechanism is needed that permits an RMC client to qualify an event. In other words, an event may be of some interest, but only if some other condition is also met.

SQL restrictions:

SQL syntax is supported for selection strings.

The following table relates the RMC terminology to SQL terminology.

Table 3. Relationship of RMC terminology to SQL terminology

RMC terminology	SQL terminology
attribute name	column name
selection string	WHERE clause
operators	predicates, logical connectives
resource class	table

Although SQL syntax is generally supported in selection strings, the following restrictions apply.

- Only a single table may be referenced in an expression.
- Queries may not be nested.
- The IS NULL predicate is not supported because there is no concept of a NULL value.
- The period (.) operator is not a table separator (for example, table.column). Rather, in this context, the period (.) operator is used to separate a field name from its containing structure name.
- The pound sign (#) is hard-coded as the escape character within SQL pattern strings.

- All column names are case sensitive.
- All literal strings must be enclosed in either single or double quotation marks. Bare literal strings are not supported because they cannot be distinguished from column and attribute names.

Supported base data types:

The term *variable* is used in this context to mean the column name or attribute name in an expression.

Variables and constants in an expression may be one of the following data types that are supported by the RMC subsystem:

Table 4. Supported Base Data Types

Symbolic name	Description
CT_INT32	Signed 32–bit integer
CT_UINT32	Unsigned 32–bit integer
CT_INT64	Signed 64–bit integer
CT_UINT64	Unsigned 64–bit integer
CT_FLOAT32	32–bit floating point
CT_FLOAT64	64–bit floating point
CT_CHAR_PTR	Null-terminated string
CT_BINARY_PTR	Binary data – arbitrary-length block of data
CT_RSRC_HANDLE_PTR	Resource handle – an identifier for a resource that is unique over space and time (20 bytes)

Aggregate data types:

In addition to the base data types, aggregates of the base data types may be used as well.

The first aggregate data type is similar to a structure in C in that it can contain multiple fields of different data types. This aggregate data type is referred to as *structured data* (SD). The individual fields in the structured data are referred to as *structured data elements*, or simply *elements*. Each element of a structured data type may have a different data type which can be one of the base types in the preceding table or any of the array types discussed in the next paragraph, except for the structured data array.

The second aggregate data type is an array. An array contains zero or more values of the same data type, such as an array of CT_INT32 values. Each of the array types has an associated enumeration value (CT_INT32_ARRAY, CT_UINT32_ARRAY). Structured data may also be defined as an array but is restricted to have the same elements in every entry of the array.

Data types that can be used for literal values:

Several literal values can be specified for each of the base data types.

Literal values can be specified for each of the base data types as follows:

Array An array or list of values may be specified by enclosing variables or literal values, or both, within braces {} or parentheses () and separating each element of the list with a comma. For example: { 1, 2, 3, 4, 5 } or ("abc", "def", "ghi").

Entries of an array can be accessed by specifying a subscript as in the C programming language. The index corresponding to the first element of the array is always zero; for example, List [2] references the third element of the array named List. Only one subscript is allowed. It may be a variable, a constant, or an expression that produces an integer result. For example, if List is an integer array, then List[2]+4 produces the sum of 4 and the current value of the third entry of the array.

Binary Data

A binary constant is defined by a sequence of hexadecimal values, separated by white space. All hexadecimal values comprising the binary data constant are enclosed in double quotation marks. Each hexadecimal value includes an even number of hexadecimal digits, and each pair of hexadecimal digits represents a byte within the binary value. For example:

```
"0xabcd 0x01020304050607090a0b0c0d0e0f1011121314"
```

Character Strings

A string is specified by a sequence of characters surrounded by single or double quotation marks (you can have any number of characters, including none). Any character may be used within the string except the null '\0' character. Double quotation marks and backslashes may be included in strings by preceding them with the backslash character.

Floating Types

These types can be specified by the following syntax:

- A leading plus (+) or minus (-) sign
- One or more decimal digits
- A radix character, which at this time is the period (.) character
- An optional exponent specified by the following:
  - A plus (+) or minus (-) sign
  - The letter 'E' or 'e'
  - A sequence of decimal digits (0–9)

Integer Types

These types can be specified in decimal, octal, or hexadecimal format. Any value that begins with the digits 1-9 and is followed by zero or more decimal digits (0-9) is interpreted as a decimal value. A decimal value is negated by preceding it with the character '-'. Octal constants are specified by the digit 0 followed by 1 or more digits in the range 0-7. Hexadecimal constants are specified by a leading 0 followed by the letter x (uppercase or lowercase) and then followed by a sequence of one or more digits in the range 0–9 or characters in the range a–f (uppercase or lowercase).

Resource Handle

A fixed-size entity that consists of two 16-bit and four 32-bit words of data. A literal resource handle is specified by a group of six hexadecimal integers. The first two values represent 16-bit integers and the remaining four each represent a 32-bit word. Each of the six integers is separated by white space. The group is surrounded by double quotation marks. The following is an example of a resource handle:

```
"0x4018 0x0001 0x00000000 0x0069684c 0x00519686 0xaf7060fc"
```

Structured Data

Structured data values can be referenced only through variables. Nevertheless, the RMC command-line interface displays structured data (SD) values and accepts them as input when a resource is defined or changed. A literal SD is a sequence of literal values, that are separated by commas and enclosed in square brackets. For example, ['abc',1,{3,4,5}] specifies an SD that consists of three elements: (a) the string 'abc', (b) the integer value 1, and (c) the three-element array {3,4,5}.

Variable names refer to values that are not part of the expression but are accessed while evaluating the expression. For example, when RMC processes an expression, the variable names are replaced by the corresponding persistent or dynamic attributes of each resource.

Entries of an array may be accessed by specifying a subscript as in 'C'. The index corresponding to the first element of the array is always 0 (for example, List[2] refers to the third element of the array named List). Only one subscript is allowed. It may be a variable, a constant, or an expression that produces an integer result. A subscripted value may be used wherever the base

data type of the array is used. For example, if List is an integer array, then "List[2]+4" produces the sum of 4 and the current value of the third entry of the array.

The elements of a structured data value can be accessed by using the following syntax:

```
<variable name>.<element name>
```

For example, `a.b`

The variable name is the name of the table column or resource attribute, and the element name is the name of the element within the structured data value. Either or both names may be followed by a subscript if the name is an array. For example, `a[10].b` refers to the element named b of the 11th entry of the structured data array called a. Similarly, `a[10].b[3]` refers to the fourth element of the array that is an element called b within the same structured data array entry `a[10]`.

Handling variable names:

Variable names refer to values that are not part of an expression but are accessed while evaluating the expression. When used to select a resource, the variable name is a persistent attribute.

When used to generate an event, the variable name is usually a dynamic attribute (but may be a persistent attribute). When used to select audit records, the variable name is the name of a field within the audit record.

A variable name is restricted to include only 7-bit ASCII characters that are alphanumeric (a-z, A-Z, 0-9) or the underscore character (_). The name must begin with an alphabetic character.

When the expression is used by the RMC subsystem for an event or a rearm event, the name can have a suffix that is the '@' character followed by 'P', which refers to RMC's previous observation of the attribute value. Because RMC observes attribute values periodically and keeps track of the previously observed value, you can use this syntax to compare the currently observed value with the previously observed value.

Operators that can be used in expressions:

Constants and variables may be combined by an operator to produce a result that in turn may be used with another operator.

The resulting data type of the expression must be a scalar integer or floating-point value. If the result is zero, the expression is considered to be FALSE; otherwise, it is TRUE.

Note: Blanks are optional around operators and operands unless their omission causes an ambiguity. An ambiguity typically occurs only with the word form of operator (that is, AND, OR, IN, LIKE, etc.). With these operators, a blank or separator, such as a parenthesis or bracket, is required to distinguish the word operator from an operand. For example, aANDb is ambiguous. It is unclear if this is intended to be the variable name aANDb or the variable names a, b combined with the operator AND. It is actually interpreted by the application as a single variable name aANDb. With non-word operators (for example, +, -, =, &&, etc.) this ambiguity does not exist, and therefore blanks are optional.

The set of operators that can be used in expressions is summarized in the following table:

Table 5. Operators That Can Be Used in Expressions

Operator	Description	Left Data Types	Right Data Types	Example	Notes
+	Addition	Integer,float	Integer,float	"1+2" results in 3	None
-	Subtraction	Integer,float	Integer,float	"1.0-2.0" results in -1.0	None
*	Multiplication	Integer,float	Integer,float	"2*3" results in 6	None
/	Division	Integer,float	Integer,float	"2/3" results in 1	None
-	Unary minus	None	Integer,float	"-abc"	None
+	Unary plus	None	Integer,float	"+abc"	None
..	Range	Integers	Integers	"1..3" results in 1,2,3	Shorthand for all integers between and including the two values
%	Modulo	Integers	Integers	"10%2" results in 0	None
|	Bitwise OR	Integers	Integers	"2|4" results in 6	None
&	Bitwise AND	Integers	Integers	"3&2" results in 2	None
~	Bitwise complement	None	Integers	~0x0000ffff results in 0xffff0000	None
^	Exclusive OR	Integers	Integers	0x0000aaaa^0x0000ffff results in 0x00005555	None
>>	Right shift	Integers	Integers	0x0fff>>4 results in 0x00ff	None
<<	Left shift	Integers	Integers	"0x0ffff<<4" results in 0xffff0	None
== =	Equality	All but SDs	All but SDs	"2==2" results in 1 "2=2" results in 1	Result is true (1) or false (0)
!= <>	Inequality	All but SDs	All but SDs	"2!=2" results in 0 "2<>2" results in 0	Result is true (1) or false (0)
>	Greater than	Integer,float	Integer,float	"2>3" results in 0	Result is true (1) or false (0)
>=	Greater than or equal	Integer,float	Integer,float	"4>=3" results in 1	Result is true (1) or false (0)
<	Less than	Integer,float	Integer,float	"4<3" results in 0	Result is true (1) or false (0)
<=	Less than or equal	Integer,float	Integer,float	"2<=3" results in 1	Result is true (1) or false (0)
=~	Pattern match	Strings	Strings	"abc"=~"a.*" results in 1	Right operand is interpreted as an extended regular expression. To use this operator in an expression, the locale(s) of the node(s) running the RMC daemon must be using either Unicode Transfer Format-8 (UTF-8) encoding (or a codeset that matches UTF-8), or else C locale encoding. If multiple nodes are involved, the encoding must be consistent across all nodes.

Table 5. Operators That Can Be Used in Expressions (continued)

Operator	Description	Left Data Types	Right Data Types	Example	Notes
!~	Not pattern match	Strings	Strings	"abc"!~"a.*" results in 0	Right operand is interpreted as an extended regular expression. To use this operator in an expression, the locale(s) of the node(s) running the RMC daemon must be using either Unicode Transfer Format-8 (UTF-8) encoding (or a codeset that matches UTF-8), or else C locale encoding. If multiple nodes are involved, the encoding must be consistent across all nodes.
=? LIKE like	SQL pattern match	Strings	Strings	"abc"=? "a%" results in 1	Right operand is interpreted as a SQL pattern
!? NOT LIKE not like	Not SQL pattern match	Strings	Strings	"abc"!? "a%" results in 0	Right operand is interpreted as a SQL pattern
|< IN in	Contains any	All but SDs	All but SDs	"{1..5}|<{2,10}" results in 1	Result is true (1) if left operand contains any value from right operand
>> NOT IN not in	Contains none	All but SDs	All but SDs	"{1..5}><{2,10}" results in 1	Result is true (1) if left operand contains no value from right operand
&<	Contains all	All but SDs	All but SDs	"{1..5}&<{2,10}" results in 0	Result is true (1) if left operand contains all values from right operand
|| OR or	Logical OR	Integers	Integers	"(1<2)||(2>4)" results in 1	Result is true (1) or false (0)
&& AND and	Logical AND	Integers	Integers	"(1<2)&&(2>4)" results in 0	Result is true (1) or false (0)

Table 5. Operators That Can Be Used in Expressions (continued)

Operator	Description	Left Data Types	Right Data Types	Example	Notes
! NOT not	Logical NOT	None	Integers	"!(2==4)" results in 1	Result is true (1) or false (0)

When integers of different signs or size are operands of an operator, standard C style casting is implicitly performed. When an expression with multiple operators is evaluated, the operations are performed in the order defined by the precedence of the operator. The default precedence can be overridden by enclosing the portion or portions of the expression to be evaluated first in parentheses (). For example, in the expression "1+2*3", multiplication is normally performed before addition to produce a result of 7. To evaluate the addition operator first, use parentheses as follows: "(1+2)*3". This produces a result of 9. The default precedence rules are shown in the following table. All operators in the same table cell have the same or equal precedence.

Table 6. Operator Precedence

Operators	Description
.	Structured data element separator
~	Bitwise complement
!	Logical not
NOT	
not	
-	Unary minus
+	Unary plus
*	Multiplication
/	Division
%	Modulo
+	Addition
-	Subtraction
<<	Left shift
>>	Right shift
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal

Table 6. Operator Precedence (continued)

Operators	Description
== != =? LIKE like !? =~ !~ ?= | IN in NOT IN not in	Equality Inequality SQL match SQL not match Reg expr match Reg expr not match Reg expr match (compat) Contains any Contains none Contains all
&	Bitwise AND
^	Bitwise exclusive OR
|	Bitwise inclusive OR
&&	Logical AND
||	Logical OR
,	List separator

Pattern matching:

Two types of pattern matching are supported; extended regular expressions and that which is compatible with the standard SQL LIKE predicate.

This type of pattern may include the following special characters:

- The percentage sign (%) matches zero or more characters.
- The underscore (_) matches exactly one character.
- All other characters are directly matched.
- The special meaning for the percentage sign and the underscore character in the pattern may be overridden by preceding these characters with an escape character, which is the pound sign (#) in this implementation.

With respect to the extended regular expression matching syntax (=~), anomalous behavior may result if the locale of the RMC daemon (for event expression evaluation) or the Resource Manager daemons (for select string evaluation) is not a UTF-8 locale. Expressions and select strings passed into the RMC

subsystem are converted to UTF-8. The =~ operator is implemented through the use of **regcomp** and **regexec**, which are effected by the locale. Therefore, if the locale is not UTF-8, unexpected results may occur.

Qualifiers:

A mechanism is needed that permits an RMC client to qualify an event. In other words, an event may be of some interest, but only if some other condition is also met.

Normally, an expression that evaluates to True results in the generation of an event. But, it might be the case that a single event is not of much interest. Consider the "file system close to full" example:

```
PercentTotUsed > 90
```

While it is interesting that a file system is almost full, to a system administrator responsible for managing file systems, what might be of more interest is that a file system is close to full and remains in that condition for a period of time. A temporary spike in usage can be ignored.

A *qualifier* is an extension to the expression syntax that specifies this other condition. A qualifier consists of a double underscore (__), the string **QUAL**, a single underscore (_), and the qualifier name (*xxxx*). It has the following general form:

```
expression __QUAL_xxxx(arg1, arg2, ...)
```

A qualifier is appended to the end of an expression, separated by one or more blanks. A qualifier can be used with a primary expression or a re-arm expression.

The **__QUAL_COUNT(***arg1***, *arg2***)** qualifier counts the number of True expression evaluations. Once the *arg1* number of True evaluations have occurred, the event notification is generated. However, this count is maintained within a moving window of the last *arg2* consecutive evaluations. Once *arg2* consecutive evaluations have occurred, prior to performing the next evaluation, the count of True evaluations is reduced by one if the oldest evaluation in the window was True. When an event notification is generated, or would have been generated if the **MC_REG_OPTS_REARM_EVENT** option was set, the count of True evaluations and consecutive evaluations is set to **0**.

The value for *arg1* must be less than or equal to the value for *arg2*. Continuing with the file system full example, consider the following primary and re-arm expressions, respectively:

```
PercentTotUsed > 90 __QUAL_COUNT(7, 10)
PercentTotUsed < 60
```

If seven out of the last 10 primary expression evaluations are True, an event notification is generated and the re-arm expression is evaluated until it is True. In simpler terms, if seven out of the last 10 samples of file system usage were greater than 90%, an event is generated. Another event will not be generated until the file system usage drops below 60%.

If all the attributes in the primary expression have a regular period, *arg2* can be considered a duration over which the count of True evaluations is maintained. The actual duration is a function of the shortest and longest reporting interval associated with the attributes specified in the expression, as given by:

```
min_interval * arg2 <= duration <= max_interval * arg2
```

For this example, the duration is 10 minutes.

The **__QUAL_RATE(***arg1***, *arg2***)** qualifier specifies a rate of True expression evaluations that must be achieved before an event notification is generated. *arg1* is a count of True evaluations and *arg2* is a number of seconds. An event notification is generated when the last *arg1* True evaluations have occurred within *arg2* seconds. Once *arg1* True evaluations have occurred, prior to performing the next evaluation, the count of True evaluations is reduced by one. False evaluations are ignored and not counted. When an event notification is generated, or would have been generated if the **MC_REG_OPTS_REARM_EVENT**

option was set, the count of True evaluations is set to **0**. Note that the rate calculation is not performed until *arg1* True evaluations have occurred. The time at which this occurs is a function of the time a new value is reported for any of the attributes in the expression.

The rate qualifier is probably more appropriate for expressions containing attributes of variable type **State**, that is, the periodic evaluation is irregular. If the operational state of a resource is expected to change, but not change rapidly, this expression is useful:

```
OpState != OpState@P __QUAL_RATE(5, 60)
```

If five True evaluations occur within one minute, an event is generated. Note that this qualifier is not an instantaneous rate calculation, that is, an event is not generated if the express is True every (*arg1 / arg2*) seconds, or every 0.0833 seconds in the preceding example.

Qualifiers can be used with re-arm expressions as well.

Related concepts:

"Data types and operators supported in expressions" on page 19
An expression in RMC is similar to a C language statement or the WHERE clause of an SQL query. It is composed of variables, operators and constants. The C and SQL syntax styles may be intermixed within a single expression.

Custom dynamic attributes:

The RMC subsystem supports the definition of custom dynamic attributes within a resource class and their use in event expressions.

Custom dynamic attributes are run-time extensions to the resource dynamic attributes that are returned by the RMC API's **mc_qdef_d_attribute** subroutine. Custom dynamic attributes are defined for each resource. The manner in which custom dynamic attributes are defined is a function of the resource class implementation. Custom dynamic attributes cannot be defined as class dynamic attributes.

Custom dynamic attributes are supported if the resource class defines the **CustomDynamicAttributes** persistent resource attribute. This persistent attribute is of type SD Array and contains one array element for each custom dynamic attribute that may be defined for the resource. For any given resource, if this persistent attribute has an array value with zero elements, custom dynamic attribute are not defined for the resource. This SD contains the following fields:

- **Name**, which is the name of the custom dynamic attribute.
- **ID**, which is the ID of the custom dynamic attribute.
- **DataType**, which is the datatype of the custom dynamic attribute.

  Custom dynamic attributes cannot be a Structured Data type.
- **VariableType**, which is the variable type of the custom dynamic attribute: **Counter**, **Quantity**, **Quantum**, or **State**. For more information about variable types, see the *Administering RSCT* guide.
- **ReportingInterval**, which is the reporting interval of the custom dynamic attribute.
- **Properties**, which are the properties of the custom dynamic attribute.

  One property is supported. If Bit 0 of the property value is set, the attribute must be monitored using an event expression in order to query its value using the **mc_query_d_handle** subroutine or the **mc_query_d_select** subroutine.

Each resource does not need to define the same custom dynamic attributes. However, when using the **mc_reg_event_select** subroutine, all selected resource must have matching values for their **CustomDynamicAttributes** persistent resource attributes if any custom dynamic attributes are used in the event expressions. If the RMC subsystem cannot obtain the values of the **CustomDynamicAttributes** persistent resource attributes for the selected resources, the **mc_reg_event_handle** subroutine or the **mc_reg_event_select** subroutine returns an error in its response, indicating that the custom dynamic attributes in the expressions could not be validated.

Examples of expressions:

This topic contains examples of the types of expressions that can be constructed.

Some examples of the types of expressions that can be constructed follow:

1. The following expressions match all rows or resources that have a name which begins with '*tr*' and ends with '*0*', where "*Name*" indicates the column or attribute that is to be used in the evaluation:

   ```
   Name =~'tr.*0'
   Name LIKE 'tr%0'
   ```

2. The following expressions evaluate to *TRUE* for all rows or resources that contain *1*, *3*, *5*, *6*, or *7* in the column or attribute that is called *IntList*, which is an array:

   ```
   IntList|<{1,3,5..7}
   IntList in (1,3,5..7)
   ```

3. The following expression combines the previous two so that all rows and resources that have a name beginning with '*tr*' and ending with '*0*' and have *1*, *3*, *5*, *6*, or *7* in the *IntList* column or attribute will match:

   ```
   (Name LIKE "tr&(IntList|<(1,3,5..7))
   (Name=~'tr.*0') AND (IntList IN {1,3,5..7})
   ```

Notifying the application of errors

Errors can be detected in various ways when using the RMC API.

When using the RMC API, errors can be detected by:

- an RMC API subroutine. When an RMC API subroutine detects an error, the subroutine returns immediately with an error code. If the subroutine is one that usually issues a command to the RMC subsystem, the command is not issued and therefore, no response will be generated by the RMC subsystem. The application can use a set of cluster common utilities to get additional information about an error returned by an RMC API subroutine.
- an RMC subsystem daemon or a resource manager. When an RMC subsystem daemon or a resource manager detects an error, the response structure or event notification structure will contain an error structure.

Related information:

"RMC API error codes and return values" on page 243
Errors can be detected by an RMC API subroutine, an RMC subsystem daemon, or a resource manager.

"Cluster utilities: error-related subroutines" on page 264
The cluster utilities component of RSCT includes several subroutines that an application can use to get additional information about errors that are returned by RMC API subroutines.

Obtaining error information returned by the RMC API subroutines:

All RMC API subroutines return a value of type **ct_int32_t**. A return value of 0 indicates that the subroutine completed successfully. Any non-zero value is an error value.

The possible errors are defined by macros in the header file **ct_mc.h**.

In many cases, the returned error value is sufficient information for the application to determine the appropriate recovery action. However, detailed error information can be obtained by calling the **cu_get_error** subroutine (described in "cu_get_error" on page 265). This subroutine has one argument, which is the address in which the function returns a pointer to an error structure. The detailed error information that can be returned in this structure is stored by the RMC API subroutine in a common, per-thread data area. Therefore, to obtain additional error information, the application must call the **cu_get_error** subroutine using the same thread that invoked the failing subroutine, before calling any other subroutine on that thread.

In addition to the error structure, the application can obtain a message that corresponds to the error by calling the “cu_get_errmsg” on page 264 subroutine. This subroutine accepts a pointer to an error structure and returns, at a location specified by the application, the error message.

The memory that is returned by the **cu_get_error** and **cu_get_errmsg** subroutines is not reused by the subroutines, so the application can hold the memory as long as necessary. The memory returned by these subroutines, however, must not be modified by the application.

When an error structure that is obtained by the **cu_get_error** subroutine is no longer needed, the application can free it by calling the **cu_rel_error** subroutine (described in “cu_rel_error” on page 270). Similarly, when a message obtained by the **cu_get_errmsg** subroutine is no longer needed, the application can free it by calling the **cu_rel_errmsg** subroutine (described in “cu_rel_errmsg” on page 269).

To package error information into a cluster error structure or to create cluster error structures that are returned by microsensor API subroutines, use the **cu_pkg_error** subroutine or the **cu_vpkg_error** subroutine (described in “cu_pkg_error, cu_vpkg_error” on page 267).

Related information:

“Cluster utilities: error-related subroutines” on page 264
The cluster utilities component of RSCT includes several subroutines that an application can use to get additional information about errors that are returned by RMC API subroutines.

“RMC API error codes and return values” on page 243
Errors can be detected by an RMC API subroutine, an RMC subsystem daemon, or a resource manager.

Obtaining error information returned in response structures or event notification structures:

All response structures and event notification structures contain error information.

The error information is in a structure of type **mc_errnum_t**.

```
typedef struct mc_errnum                mc_errnum_t;
struct mc_errnum {
     ct_uint32_t                        mc_errnum;
     ct_char_t                          *mc_ffdc_id;
     ct_char_t                          *mc_error_msg;
     cu_error_arg_t                     *mc_args;
     ct_uint32_t                        mc_arg_count;
};
```

The fields of this structure contain the following:

mc_errnum
: An error code from the RMC subsystem. If this field is zero, there is no error. The error codes can be returned in response structures and event notification structures.

mc_ffdc_id
: A pointer to a string that is a failure identifier. This failure identifier specifies additional error information that may have been logged by the RMC subsystem. If this field contains a NULL pointer, then no additional error information has been logged. If the application is logging errors that it has detected, then this failure identifier should be included in the information being logged.

mc_error_msg
: A pointer to an error message.

mc_args
: If the *mc_arg_count* field of this structure is non-zero, a pointer to an array of *mc_arg_count* elements. Each element of the array is an error argument specific to the error.

mc_arg_count
: The number of elements in the *mc_args* array.

Error codes that may be returned in the *mc_errnum* field are grouped in ranges of 64K, starting with the lowest values in each group. The high-order sixteen bits of the error code identify the error group or general class of error, and the lower sixteen bits indicate a specific error within the group. The application can use the macros MC_GET_GENERR and MC_GET_SPECERR to obtain these values. The macro definitions are:

```
#define MC_GET_GENERR(e) ((e >> 16) & 0xffff)
#define MC_GET_SPECERR(e) (e & 0xffff)
```

Related reference:

"Response and event structure error codes" on page 247
All response structures and event notification structures contain error information in a structure of type **mc_errnum_t**.

Related information:

"RMC API error codes and return values" on page 243
Errors can be detected by an RMC API subroutine, an RMC subsystem daemon, or a resource manager.

RMC API subroutine overview

The RMC API subroutines are grouped into several categories.

The RMC API subroutines are grouped as:

- Session interfaces for establishing and ending sessions with the RMC subsystem.
- Command management interfaces creating and issuing command groups, assigning threads to the RMC API, and managing responses and event notifications.
- Monitoring command interfaces for registering events.
- Configuration command interfaces for querying or modifying the configuration of resources.
- Control command interfaces for bringing resources online, and taking them offline.

Session interfaces

The following table summarizes the RMC API session interfaces.

These subroutines enable an application to start and end one or more sessions with the RMC subsystem, and to obtain a file descriptor to detect when a session has received response or event notifications.

Please note that this table merely summarizes the interfaces, and is not meant to be complete. For complete information on these subroutines, see "RMC API subroutines" on page 40.

Table 7. Overview of session interfaces

Subroutine:	Description:	When calling this subroutine, the application specifies:
"mc_start_session" on page 206	Establishes a session with the RMC subsystem.	• One or more nodes the RMC API may contact to start a session with the RMC subsystem. • Session scope options • An address for the session handle.
"mc_timed_start_session" on page 212	Establishes a session with the RMC subsystem. This subroutine is identical to the **mc_start_session** subroutine except that it also enables the calling application to specify time limits for establishing a session and, once the session is established, for completion of blocking operations.	• One or more nodes the RMC API may contact to start a session with the RMC subsystem. • Session scope options • An address for the session handle • A time limit for establishing a session • A time limit for blocking operations
"mc_end_session" on page 64	Ends a session with the RMC subsystem.	The session handle.

Table 7. Overview of session interfaces (continued)

Subroutine:	Description:	When calling this subroutine, the application specifies:
"mc_get_descriptor" on page 76	Returns a descriptor that can be used in a poll or select system call. The descriptor is made ready for read whenever the RMC API has received a response or event notification from the RMC subsystem, but needs a thread to process an associated callback routine. Using a descriptor in a select or poll system call enables the application to prevent the **mc_dispatch** subroutine from blocking as described in "A single-threaded application surrendering its thread to the RMC API" on page 16.	• The session handle • An address for the descriptor.
"mc_free_descriptor" on page 74	Frees a descriptor previously obtained by a call to the **mc_get_descriptor** subroutine. .	• The session handle • The descriptor to be freed

Command management interfaces

The following table summarizes the RMC API command management interfaces.

These subroutines enable an application to create and issue command groups, assign threads to the RMC API, and manage responses and event notifications.

Please note that this table merely summarizes the interfaces, and is not meant to be complete.

Table 8. Overview of command management interfaces

Subroutine:	Description:	When calling this subroutine, the application specifies:
"mc_dispatch" on page 62	Provides a thread to the RMC API to enable it to invoke a callback routine to process a response or event notification.	• The session handle • Either the MC_DISPATCH_OPTS_WAIT or the MC_DISPATCH_OPTS_ASSIGN option.
"mc_start_cmd_grp" on page 204	Alllocates a command group.	• The session handle • Options for the command group (whether it is an ordered command group, and, if so, whether it is non-interleaved) • An address for the returned command group handle.
"mc_cancel_cmd_grp" on page 41	Cancels a command group.	The command group handle.
"mc_send_cmd_grp" on page 188	Sends a command group to the RMC subsystem.	• The command group handle • The completion callback routine to be invoked by the RMC API after all responses to commands in the command group have been processed.
"mc_send_cmd_grp_wait" on page 191	Sends a command group to the RMC subsystem and waits for completion.	The command group handle.
"mc_free_response" on page 75	Frees the storage used by a response or event notification structure.	A pointer to the response array or event notification structure.

Related concepts:

"Providing the RMC API with one or more threads" on page 15
The application must provide the RMC API with a thread for processing responses and event notifications under two conditions.

Monitoring command interfaces

The following table summarizes the RMC API monitoring command interfaces.

These subroutines enable an application to register for event notifications from the RMC subsystem. Each of the monitoring command interfaces have four related subroutines that issue the same command to the RMC subsystem, but are differentiated by how the command is sent to the RMC subsystem (using a blocking subroutine, or added to a command group to be sent later), and how the command response is made available to the application (using callback response or pointer response). When there are four related subroutines that all provide essentially the same command interface they are suffixed by _* in documentation. The _* suffix represents the four individual suffixes of the subroutines, which are **_bp**

(blocking/pointer response), **_ap** (added to command group/pointer response), **_bc** (blocking/callback response), and **_ac** (added to command group/callback response).

Please note that this table merely summarizes the interfaces, and is not meant to be complete.

Table 9. Overview of monitoring command interfaces

Subroutine:	Description:	When calling this subroutine, the application specifies:	If successful, a response structure contains:	Number of response structures returned:
"mc_reg_event_select_*" on page 176	Registers a resource event with the RMC subsystem. The event is registered for one or more resources of the resource class using attribute selection.	The application specifies: • The session handle • A pointer to resource class name • Selection string to identify one or more resources of the resource class • A pointer to an event expression • Optionally, a pointer to a rearm event expression • The event callback routine to be invoked to process event notifications • Optionally, an array of pointers to persistent attribute names. This array identifies additional attribute values to be returned in event notifications	The event registration ID	1
"mc_reg_event_handle_*" on page 168	Registers a resource event with the RMC subsystem using a resource handle.	• The session handle • The resource handle • a pointer to an event expression • Optionally, a pointer to a rearm event expression • The event callback routine to be invoked to process event notifications • Optionally, an array of pointers to persistent attribute names. This array identifies additional attribute values to be returned in event notifications	The event registration ID	1
"mc_reg_class_event_*" on page 160	Registers a resource class event with the RMC subsystem.	• The session handle • A pointer to a resource class name • If the management style of the resource class is globalized and the session scope is DM, a pointer to an array of peer domain names. This identifies the peer domain(s) where the resource class event should be registered. • A pointer to an event expression • Optionally, a pointer to a rearm event expression • The event callback routine to be invoked to process event notifications • Optionally, an array of pointers to persistent attribute names. This array identifies additional attribute values to be returned in event notifications	The event registration ID	1
"mc_query_event_*" on page 143	Queries the RMC subsystem to obtain an event's current state.	• The session handle • The event registration ID	• The event registration ID • The number of events that were generated as a result of issuing this command.	1
"mc_unreg_event_*" on page 223	Unregisters an event with the RMC subsystem.	• The session handle • The event registration ID	The event registration ID of the event that was unregistered	1

Configuration command interfaces

The following tables summarize the RMC API configuration command interfaces.

These subroutines enable an application to:

- query resources and resource classes
- query the definition of resource classes, attributes, Structured Data, valid values, or actions
- define or modify a resource
- invoke resource or resource class actions

Please note that the following tables merely summarize the interfaces, and are not meant to be complete.

Each of the configuration command interfaces have four related subroutines that issue the same command to the RMC subsystem, but are differentiated by how the command is sent to the RMC subsystem (using a blocking subroutine, or added to a command group to be sent later), and how the command response is made available to the application (using callback response or pointer response). When there are four related subroutines that all provide essentially the same command interface they are suffixed by _* in documentation. The _* suffix represents the four individual suffixes of the subroutines, which are **_bp** (blocking/pointer response), **_ap** (added to command group/pointer response), **_bc** (blocking/callback response), and **_ac** (added to command group/callback response).

The following table summarizes the configuration command interfaces for querying resources and resource classes

Table 10. Overview of configuration command interfaces – query commands

Subroutine:	Description:	When calling this subroutine, the application specifies:	If successful, a response structure contains:	Number of response structures returned:
"mc_enumerate_resources_*" on page 69	Enumerates the resources of a resource class using attribute selection.	• the session handle • A pointer to a resource class name • A selection string	• The resource class name • A pointer to an array of resource handles	1 or more
"mc_enumerate_permitted_rsrcs_*" on page 65	Enumerates the resources of a resource class using attribute selection. Enumerates only those resources for which the calling application has specified permissions.	• The session handle • a pointer to a resource class name • A selection string • The required permissions	• The resource class name • A pointer to an array of resource handles	1 or more
"mc_query_p_select_*" on page 151	Queries the RMC subsystem to obtain the persistent attribute values of one or more resources of a resource class. The resources are identified using attribute selection.	• The session handle • A pointer to the resource class name • A selection string to specify one or more resources of the resource class • An array of pointers to persistent attribute names or, to specify all persistent attributes, a NULL pointer	• The resource handle of a resource that was queried • A pointer to an array of the requested attributes	1 for each resource identified by the selection string
"mc_query_d_select_*" on page 138	Queries the RMC subsystem to obtain the dynamic attribute values of one or more resources of a resource class. The resources are identified using attribute selection.	• The session handle • A pointer to the resource class name • A selection string to specify one or more resources of the resource class • An array of pointers to dynamic attribute names or, to specify all dynamic attributes, a NULL pointer	• The resource handle of a resource that was queried • A pointer to an array of the requested attributes	1 for each resource identified by the selection string

Table 10. Overview of configuration command interfaces – query commands (continued)

Subroutine:	Description:	When calling this subroutine, the application specifies:	If successful, a response structure contains:	Number of response structures returned:
"mc_query_p_handle_*" on page 147	Queries the RMC subsystem to obtain the persistent attribute values of a resource. The resource is identified using a resource handle.	• The session handle • The resource handle • An array of pointers to persistent attribute names or, to specify all persistent attributes, a NULL pointer	• The resource handle of the resource that was queried • A pointer to an array of the requested attributes	1
"mc_query_d_handle_*" on page 133	Queries the RMC subsystem to obtain the dynamic attribute values of a resource. The resource is identified using attribute selection.	• The session handle • The resource handle • An array of pointers to dynamic attribute names or, to specify all dynamic attributes, a NULL pointer	• The resource handle of the resource that was queried • A pointer to an array of the requested attributes	1
"mc_class_query_p_*" on page 47	Queries the RMC subsystem to obtain the persistent attribute values of a resource class.	• The session handle • A pointer to the resource class name • An array of pointers to persistent attribute names or, to specify all persistent attributes, a NULL pointer • If the management style of the resource class is globalized and the session scope is DM, a pointer to an array of peer domain names	• The name of the resource class that was queried • A pointer to an array of the requested attributes • If the persistent attribute values were queried for a peer domain, the name of the peer domain	1 for each peer domain identified, or, if no peer domains were identified, 1
"mc_class_query_d_*" on page 42	Queries the RMC subsystem to obtain the dynamic attribute values of a resource class.	• The session handle • A pointer to the resource class name • An array of pointers to dynamic attribute names or, to specify all dynamic attributes, a NULL pointer • If the management style of the resource class is globalized and the session scope is DM, a pointer to an array of peer domain names	• The name of the resource class that was queried • A pointer to an array of the requested attributes • If the dynamic attribute values were queried for a peer domain, the name of the peer domain	1 for each peer domain identified, or, if no peer domains were identified, 1
"mc_validate_rsrc_hndl_*" on page 227	Validate one or more resource handles.	• The session handle • The pointer to an array of resource handles	A resource handle for a resource that was the target of this command. The error field of the return structure will indicate whether or not the resource is valid.	1 for each resource handle specified by the application

The following table summarizes the configuration command interfaces for querying the definition of resource classes, attributes, Structured Data, valid values, or actions

Table 11. Overview of configuration command interfaces – query definition commands

Subroutine:	Description:	When calling this subroutine, the application specifies:	If successful, a response structure contains:	Number of response structures returned:
"mc_qdef_resource_class_*" on page 114	Queries the RMC subsystem to obtain the definition of a resource class, or all definitions of all resource classes.	• The session handle • A pointer to a resource class name, or, to return information for all resource classes, a NULL pointer • Whether or not detailed descriptions or display names should be returned	• The resource class name • The resource class ID • If requested, a display name (suitable to display as the name of the resource class in a GUI) • If requested, a pointer to a detailed description of the resource class • The name of a persistent attribute of a resource of the resource class that implies the location of the resource • The number of persistent attributes defined for the resource class itself • The number of dynamic attributes defined for the resource class itself • The number of persistent attributes for a resource of the resource class • The number of dynamic attributes for a resource of the resource class • The number of different actions that can be invoked against a resource of the resource class • The number of different actions that can be invoked against the resource class itself • Information on the resource manager (or resource managers) that implements this resource class	1 for each class queried
"mc_qdef_p_attribute_*" on page 108	Queries the RMC subsystem to obtain persistent attribute definitions for a resource or resource class.	• The session handle • A pointer to the resource class name • A pointer to an array of persistent attribute names • Whether or not detailed descriptions or display names should be returned • Whether information for persistent class attributes or persistent resource attributes should be returned	• Properties of the persistent attribute • The programmatic name of the attribute • If requested, a display name (suitable to display as the name of the attribute in a GUI) • A pointer to the name of a group to which the attribute belongs • If requested, a pointer to a detailed description of the attribute • The attribute ID • The attribute's group ID (used to group related attributes) • The data type of the attribute • The default value of the attribute	1 for each persistent attribute queried

Table 11. Overview of configuration command interfaces – query definition commands (continued)

Subroutine:	Description:	When calling this subroutine, the application specifies:	If successful, a response structure contains:	Number of response structures returned:
"mc_qdef_d_attribute_*" on page 102	Queries the RMC subsystem to obtain dynamic attribute definitions for a resource or resource class.	• The session handle • A pointer to the resource class name • A pointer to an array of dynamic attribute names • Whether or not detailed descriptions or display names should be returned • Whether information for dynamic class attributes or dynamic resource attributes should be returned	• Properties of the dynamic attribute • The programmatic name of the attribute • If requested, a display name (suitable to display as the name of the attribute in a GUI) • A pointer to the name of a group to which the attribute belongs • If requested, a pointer to a detailed description of the attribute • The attribute ID • The attribute's group ID (used to group related attributes) • The data type of the attribute • The variable type of the attribute • The initial value of the attribute • The minimum value of the attribute • The maximum value of the attribute • An example event expression for the variable • A pointer to a description of the example event expression • An example rearm event expression • A pointer to a description of the rearm event expression • If applicable, a pointer to the PTX path name for this variable.	1 for each dynamic attribute queried
"mc_qdef_sd_*" on page 120	Queries the RMC subsystem to obtain the definition of Structured Data.	• The session handle • A pointer to the name of the resource class name for which Structured Data information should be returned • The type of Structured Data information to be returned • If Structured Data information is being returned for attributes or actions, the specific attribute or actions for which information will be returned • Whether or not detailed descriptions or display names should be returned	A Structured Data definition	1 for each Structured Data definition requested
"mc_qdef_valid_values_*" on page 126	Queries the RMC subsystem to obtain the definition of valid values.	• The session handle • A pointer to the name of the resource class for which valid value information is to be returned • The type of valid value information to be returned • The attributes or actions for which valid value information is to be returned • Whether or not detailed descriptions or display names should be returned	The requested valid value information for one persistent attribute, one dynamic attribute that has a variable type of RMC_STATE, one action input, or one command argument. If the valid values are Structured Data, the response contains valid values for each element of Structured Data	1 for each attribute or action for which valid value information was requested

Table 11. Overview of configuration command interfaces – query definition commands (continued)

Subroutine:	Description:	When calling this subroutine, the application specifies:	If successful, a response structure contains:	Number of response structures returned:
"mc_qdef_actions_*" on page 97	Queries the RMC subsystem to obtain the definitions of actions of a resource class.	• The session handle • A pointer to the resource class name • A pointer to an array of action names • Whether or not detailed descriptions and display names should be returned. • Whether information on resource class actions or resource actions should be returned	• A pointer to an array containing information about each action queried. This information includes: – Properties of the action – If requested, a display name (suitable to display as the name of the action in a GUI) – If requested, a pointer to a detailed description of the action – A pointer to a string that can be used in a GUI to prompt a user for confirmation to perform an action – The action ID – The permissions required to execute the action • The ID of the resource class	1 or more

The following table summarizes the configuration command interfaces for defining or modifying resources

Table 12. Overview of configuration command interfaces – modify configuration commands

Subroutine:	Description:	When calling this subroutine, the application specifies:	If successful, a response structure contains:	Number of response structures returned:
"mc_define_resource_*" on page 58	Defines a new resource.	• The session handle • The resource class of the new resource • The persistent attribute values for the new resource • any resource-class specific options for defining a resource	• The name of the resource class • The resource handle of the new resource instance	1
"mc_undefine_resource_*" on page 219	Removes a resource from the RMC subsystem.	• The session handle • The resource handle • optionally, a pointer to Structured Data containing resource-class specific options for undefining a resource	• The name of the resource class in which the resource instance is deleted. • The resource handle	1
"mc_refresh_config_*" on page 156	Refreshes the configuration of resources within a resource class.	• The session handle • A pointer to a resource class name	The resource class name	1 or more
"mc_set_select_*" on page 199	Sets persistent attribute values of one or more resources of a particular resource class.	• The session handle • A pointer to the resource class name • A selection string that identifies one or more resources of the resource class • The persistent attributes to be set and their new values	The resource handle of a resource whose attributes were set.	1 for each resource identified by the selection string
"mc_set_handle_*" on page 194	Sets persistent attribute values of a resource identified by a resource handle.	• The session handle • The resource handle • The persistent attribute values to be set and their new values	The resource handle	1

Table 12. Overview of configuration command interfaces – modify configuration commands (continued)

Subroutine:	Description:	When calling this subroutine, the application specifies:	If successful, a response structure contains:	Number of response structures returned:
"mc_class_set_*" on page 53	Sets one ore more persistent attributes of a resource class.	• The session handle • The resource class name • The persistent class attributes to be set and their new values • If the management style of the resource class is globalized and the session scope is DM, a pointer to an array of peer domain names.	• A pointer to the name of the resource class whose attributes were set • If the attributes were set in a peer domain, the name of the peer domain	1 for each peer domain identified, or, if no peer domains were identified, 1

The following table summarizes the configuration command interfaces for invoking actions.

Table 13. Overview of configuration command interfaces – invoke action commands

Subroutine:	Description:	When calling this subroutine, the application specifies:	If successful, a response structure contains:	Number of response structures returned:
"mc_invoke_action_*" on page 78	Invokes an action on a resource.	• The session handle • The resource handle • A pointer to the action name • If the action accepts input, a pointer to Structured Data that may be used as input to the action.	• The resource handle of the resource that was the target of the command • Optionally, a block of response data from the action	1 or more
"mc_invoke_class_action_*" on page 82	Invokes an action on a resource class.	• The session handle • A pointer to the resource class name • A pointer to the action name • If the action accepts input, a pointer to Structured Data that may be used as input to the action • If the management style of the resource class is subdivided, a pointer to an array of node names (identifying where the class action should be invoked) • If the management style of the resource class is globalized, and the session scope is DM, a pointer to an array of peer domain names (identifying the peer domains where the class action should be invoked)	• The name of the resource class that was the target of the command • Optionally, a block of response data from the action • The primary node name of the node where the cluster action was invoked • If applicable, the name of the peer domain where the class action was invoked.	1 or more

Control command interfaces

The following table summarizes the RMC API control command interfaces.

These subroutines enable an application to bring resources online, and take them offline. Each of the control command interfaces have four related subroutines that issue the same command to the RMC subsystem, but are differentiated by how the command is sent to the RMC subsystem (using a blocking subroutine, or added to a command group to be sent later), and how the command response is made available to the application (using callback response or pointer response). When there are four related subroutines that all provide essentially the same command interface they are suffixed by _* in documentation. The _* suffix represents the four individual suffixes of the subroutines, which are **_bp** (blocking/pointer response), **_ap** (added to command group/pointer response), **_bc** (blocking/callback response), and **_ac** (added to command group/callback response).

Please note that this table merely summarizes the interfaces, and is not meant to be complete.

Table 14. Overview of control command interfaces

Subroutine:	Description:	When calling this subroutine, the application specifies:	If successful, a response structure contains:	Number of response structures returned:
"mc_offline_*" on page 88	Sends a request to the RMC subsystem to bring a resource online.	• The session handle • The resource handle • A pointer to an array of node names identifying the node(s) on which the resource should be brought online • Optionally, a pointer to Structured Data containing resource class specific options bringing the resource online	The resource handle that identifies the resource that was brought online	1
"mc_online_*" on page 92	Sends a request to the RMC subsystem to take a resource offline.	• The session handle • The resource handle • Optionally, a pointer to Structured Data containing resource class specific options to taking the resource offline	The resource handle that identifies the resource that was taken offline	1
"mc_reset_*" on page 184	Sends a request of the RMC subsystem to force a resource offline.	• The session handle • The resource handle • Optionally, a pointer to Structured Data containing resource class specific options to taking the resource offline	The resource handle that identifies the resource that was taken offline	1

RMC API reference

The RMC API defines a number of macros and typedefs in order to simplify the task of programming the API, provide more complete error checking during compilation, and hide implementation details. Reference information is provided for these RMC data definitions as well as the RMC API subroutines.

RMC API subroutines

The RMC API is a library of subroutines and supporting data types, written in C.

These enable applications (in particular, cluster applications) to establish a connection with the RMC subsystem to:

- list the resources of a resource class
- monitor changes in attribute values for events of interest
- query dynamic or persistent attributes of resources or resource classes
- change the persistent attributes of resources or resource classes
- define and undefine resources
- bring resources online and take them offline

For many of the RMC command interfaces, there are four separate subroutines that issue the same command action, but vary on how the command is sent to the RMC subsystem, and how the command response is made available to the application. When a subroutine name appears in documentation suffixed by _*, the _* suffix represents the four individual suffixes of the subroutines. The four individual suffixes are described in the following table:

Table 15. Four individual suffixes of the subroutines

This suffix:	Indicates that the subroutine:
_bp	Is blocking and uses pointer response
_ap	Adds the command to a command group, and uses pointer response
_bc	Is blocking and uses callback response
_ac	Adds the command to a command group and uses callback response

Related concepts:

"Registering the application for event notifications" on page 13
A set of RMC subroutines enables an application to register with the RMC subsystem for event notifications.

"Returning responses to the application" on page 5
When the application invokes an RMC API subroutine to send a command to the RMC subsystem, a successful return value from the subroutine indicates only that the command was successfully sent to the RMC subsystem or added to a command group.

"Ordered command group" on page 4
An *ordered command group* is a command group whose commands will be processed by the RMC subsystem in the order in which they were added to the group.

mc_cancel_cmd_grp

This subroutine cancels a command group.

Purpose

Cancels a command group.

Library

RMC Library (**libct_mc.a**)

Syntax

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_cancel_cmd_grp(
          mc_cmdgrp_hndl_t   cmd_hndl)
```

Parameters

INPUT

cmd_hndl
: The command group handle that identifies the command group to cancel. A command group handle is returned by the **mc_start_cmd_group** subroutine when the application allocates a command group.

Description

The **mc_cancel_cmd_grp** subroutine can be used by the application to cancel a command group that it previously allocated using the **mc_start_cmd_grp** subroutine. The application will need to call the **mc_cancel_cmd_grp** subroutine to free a command group that it no longer needs to send to the RMC subsystem.

The application can call the **mc_cancel_cmd_grp** subroutine any time after the **mc_start_cdm_grp** subroutine returns the command group handle. However, the application can only call the

mc_cancel_cmd_grp subroutine if it has not already called the **mc_send_cmd_grp** or **mc_send_cmd_grp_wait** subroutine to send the command group to the RMC subsystem.

Return values

A return value of 0 (zero) indicates that the command group has been cancelled. Any other return value is an error and indicates that the command group has not been cancelled.

The following errors can be returned by this subroutine. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_send_cmd_grp" on page 188
This subroutine sends a command group to the RMC subsystem.

"mc_send_cmd_grp_wait" on page 191
This subroutine sends a command group to the RMC subsystem and waits for completion.

"mc_start_cmd_grp" on page 204
This subroutine allocates a command group.

mc_class_query_d_*

This subroutine queries the RMC subsystem to obtain the dynamic attribute values of a resource class.

Purpose

Queries the RMC subsystem to obtain the dynamic attribute values of a resource class.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_class_query_d_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_class_query_d_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_class_query_d_bp(
          mc_sess_hndl_t            sess_hndl,
          mc_class_query_rsp_t      **rsp_array,
          ct_uint32_t               *array_cnt,
          ct_char_t                 *rsrc_class_name,
```

```
        ct_char_t                   **pd_names,
        ct_uint32_t                 name_count,
        ct_char_t                   **return_attrs,
        ct_uint32_t                 attr_count)
```

- The **mc_class_query_d_ap** subroutine adds the command to a command group. Note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
mc_class_query_d_ap(
     mc_cmdgrp_hndl_t           cmdgrp_hndl,
        mc_class_query_rsp_t    **rsp_array,
        ct_uint32_t             *array_cnt,
        ct_char_t               *rsrc_class_name,
        ct_char_t               **pd_names,
        ct_uint32_t             name_count,
        ct_char_t               **return_attrs,
        ct_uint32_t             attr_count)
```

- The **mc_class_query_d_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_class_query_d_bc(
        mc_sess_hndl_t          sess_hndl,
        mc_class_query_cb_t     *query_cb,
        void                    *query_cb_arg,
        ct_char_t               *rsrc_class_name,
        ct_char_t               **pd_names,
        ct_uint32_t             name_count,
        ct_char_t               **return_attrs,
        ct_uint32_t             attr_count)
```

  The definition for the response callback is:

```
typedef void (mc_class_query_cb_t)(mc_sess_hndl_t,
mc_class_query_rsp_t *,
void *);
```

- The **mc_class_query_d_ac** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_class_query_d_ac(
        mc_cmdgrp_hndl_t        cmdgrp_hndl,
        mc_class_query_cb_t     *query_cb,
        void                    *query_cb_arg,
        ct_char_t               *rsrc_class_name,
        ct_char_t               **pd_names,
        ct_uint32_t             name_count,
        ct_char_t               **return_attrs,
        ct_uint32_t             attr_count)
```

  The definition for the response callback is:

```
typedef void (mc_class_query_cb_t)(mc_sess_hndl_t,
mc_class_query_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to the variations of this subroutine that add the command to the command group.

query_cb
Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

query_cb_arg
Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

rsrc_class_name
Identifies the resource class whose dynamic attribute values are being requested by the application.

pd_names

Table 16. mc_class_query_d_* subroutine conditional pd_names parameter functions

If:	Then:
The management style of the resource class is globalized and the session scope is DM	The parameter should be a pointer to an array of *name_count* peer domain names. Since a session scope of DM refers to a CSM management domain, and, since such a domain can contain multiple peer domains, this enables the application to specify the peer domain(s) where the class dynamic attributes should be queried. To specify all peer domains within the management domain, this parameter should be a NULL pointer, and the *name_count* parameter should be 0 (zero).
The management style of the resource class is subdivided or the session scope is not DM	This parameter should be a NULL pointer, and the *name_count* parameter should be 0 (zero)

name_count
Identifies the number of pointers in the *pd_names* array. If *pd_names* is a NULL pointer, this parameter must be 0 (zero)

return_attrs
An array of *attr_count* pointers to dynamic class attribute names. This parameter, in conjunction with the *attr_count* parameter enables the application to specify dynamic class attributes to be included in the response structure. If any of the specified attributes are not supported by the resource class, those attributes will not be included in the response structure.

Dynamic attributes that are of variable type Quantum cannot be queried because Quantum attributes have no value.

If *attr_count* is 0 (zero), *return_attrs* should be a NULL pointer. In this case, the response includes only the name of the resource class being queried.

attr_count
Indicates the number of pointers to dynamic class attribute names in the *return_attrs* array. This parameter, in conjunction with the *return_attrs* parameter, enables the application to specify

dynamic attributes to be included in the query response. If set to 0 (zero), the response includes only the name of the resource class being queried.

OUTPUT

rsp_array
A pointer to a location in which the RMC API will return a pointer to a response array of *array_cnt* elements. This parameter applies only to variations of this subroutine that use the pointer response method.

array_cnt
Identifies a location in which the RMC API will return the number of elements in the response array *rsp_array*. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_class_query_d_*** subroutines can be used by the application to obtain the dynamic attribute values of a resource class from the RMC subsystem.

The response for these subroutines is a structure of type **mc_class_query_rsp_t**, and is described in Response structure.

This command cannot be used in an ordered command group.

Security

To obtain dynamic class attribute information, the user of the calling application must have either **q** or **r** permission specified in an ACL entry for this resource class.

Return values

For the **mc_class_query_d_bp** and **mc_class_query_d_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and one or more responses have been received and processed.

For the **mc_class_query_d_ap** and **mc_class_query_d_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_EORDERGROUP
An attempt was made to add the command to an ordered command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before all responses could be received. If subroutine uses pointer response, check the appropriate array count for any responses. If the subroutine uses callback response, the callback will have been invoked for any responses received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_class_query_rsp_t**. If any of the query arguments are invalid, then only one response is returned, indicating the error.

If the management style of the resource class being queried is globalized, and the session scope is DM, one response will be returned for each peer domain identified by the *pd_names* and *name_count* parameters. If *pd_names* and *name_count* are, respectively, a NULL pointer and 0 (zero), a response will be returned for each peer domain within the management domain.

If the management style of the resource class is subdivided or the session scope is not DM, only one response will be returned.

The response structure definition is:

```
typedef struct mc_class_query_rsp       mc_class_query_rsp_t;
struct mc_class_query_rsp {
     mc_errnum_t                        mc_error;
     ct_char_t                          *mc_class_name;
```

```
    ct_char_t                       *mc_peer_domain_name;
    mc_attribute_t                  *mc_attrs;
    ct_uint32_t                     mc_attr_count;
};
```

The fields of this structure contain the following:

mc_error
: If 0 (zero), this field indicates that the query was successful. Any other value is an error and indicates that the RMC subsystem or a resource manager could not provide some or all of the requested information. The error may also indicate that the command arguments were in error. The error codes imply which of the remaining fields in the structure are defined.

mc_class_name
: Specifies the name of the resource class that was queried and whose attributes are contained in this response.

mc_peer_domain_name
: Specifies the name of the peer domain from which the attributes were obtained. This is a NULL string if the response is from a node not in a peer domain.

mc_attrs
: A pointer to an array of the requested attributes.

mc_attr_count
: Indicates the number of entries in the *mc_attrs* array.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_qdef_d_attribute_*" on page 102
This subroutine queries the RMC subsystem to obtain dynamic attribute definitions for a resource or resource class.

"mc_class_query_p_*"
This subroutine queries the RMC subsystem to obtain the persistent attribute values of a resource class.

"mc_query_d_handle_*" on page 133
This subroutine queries the RMC subsystem to obtain the dynamic attribute values of a resource.

"mc_query_d_select_*" on page 138
This subroutine queries the RMC subsystem to obtain the dynamic attribute values of one or more resources of a resource class.

"mc_query_p_handle_*" on page 147
This subroutine queries the RMC subsystem to obtain the persistent attribute values of a resource.

"mc_query_p_select_*" on page 151
This subroutine queries the RMC subsystem to obtain the persistent attribute values of one or more resources of a resource class.

"mc_free_response" on page 75
This subroutine frees a response or event notification structure.

mc_class_query_p_*

This subroutine queries the RMC subsystem to obtain the persistent attribute values of a resource class.

Purpose

Queries the RMC subsystem to obtain the persistent attribute values of a resource class.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_class_query_p_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_class_query_p_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_class_query_p_bp(
          mc_sess_hndl_t            sess_hndl,
          mc_class_query_rsp_t      **rsp_array,
          ct_uint32_t               *array_cnt,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 **pd_names,
          ct_uint32_t               name_count,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count)
```

- The **mc_class_query_p_ap** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, the subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_class_query_p_ap(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_class_query_rsp_t      **rsp_array,
          ct_uint32_t               *array_cnt,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 **pd_names,
          ct_uint32_t               name_count,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count)
```

- The **mc_class_query_p_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_class_query_p_bc(
          mc_sess_hndl_t            sess_hndl,
          mc_class_query_cb_t       *query_cb,
          void                      *query_cb_arg,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 **pd_names,
          ct_uint32_t               name_count,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count)
```

  The definition for the response callback is:

```
typedef void (mc_class_query_cb_t)(mc_sess_hndl_t,
mc_class_query_rsp_t *,
void *);
```

- The **mc_class_query_p_ac** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, the subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_class_query_p_ac(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_class_query_cb_t       *query_cb,
          void                      *query_cb_arg,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 **pd_names,
          ct_uint32_t               name_count,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count)
```

The definition for the response callback is:

```
typedef void (mc_class_query_cb_t)(mc_sess_hndl_t,
mc_class_query_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl

The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl

The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

query_cb

Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

query_cb_arg

Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

rsrc_class_name

Identifies the resource class whose persistent attribute values are being requested by the application.

pd_names

Table 17. mc_class_query_p_* subroutine conditional pd_names parameter functions

If:	Then:
The management style of the resource class is globalized and the session scope is DM	The parameter should be a pointer to an array of *name_count* peer domain names. Since a session scope of DM refers to a CSM management domain, and, since such a domain can contain multiple peer domains, this enables the application to specify the peer domain(s) where the class persistent attributes should be queried. To specify all peer domains within the management domain, this parameter should be a NULL pointer, and the *name_count* parameter should be 0 (zero).
The management style of the resource class is subdivided or the session scope is not DM	This parameter should be a NULL pointer, and the *name_count* parameter should be 0 (zero)

name_count
Identifies the number of pointers in the *pd_names* array. If *pd_names* is a NULL pointer, this parameter must be 0 (zero)

return_attrs
An array of *attr_count* pointers to persistent class attribute names. This parameter, in conjunction with the *attr_count* parameter enables the application to specify persistent class attributes to be included in the Query Response. If any of the specified attributes are not supported by the resource class, those attributes will not be included in the Query Response.

Persistent attributes that are of variable type Quantum cannot be queried because Quantum attributes have no value.

If *attr_count* is 0 (zero), *return_attrs* should be a NULL pointer. In this case, the response includes only the name of the resource class being queried.

attr_count
Indicates the number of pointers to persistent class attribute names in the *return_attrs* array. This parameter, in conjunction with the *return_attrs* parameter, enables the application to specify persistent attributes to be included in the query response. If set to 0 (zero), the response includes only the name of the resource class being queried.

OUTPUT

rsp_array
A pointer to a location in which the RMC API will return a pointer to a response array of *array_cnt* elements. This parameter applies only to variations of this subroutine that use the pointer response method.

array_cnt
Identifies a location in which the RMC API will return the number of elements in the response array *rsp_array*. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_class_query_p_*** subroutines can be used by the application to obtain the persistent attribute values of a resource class from the RMC subsystem.

The response for these subroutines is a structure of type **mc_class_query_rsp_t**, and is described in Response structure.

This command cannot be used in an ordered command group.

Security

To obtain persistent class attribute information, the user of the calling application must have either **q** or **r** permission specified in an ACL entry for this resource class.

Return values

For the **mc_class_query_p_bp** and **mc_class_query_p_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and one or more responses have been received and processed.

For the **mc_class_query_p_ap** and **mc_class_query_p_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_EORDERGROUP
An attempt was made to add the command to an ordered command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before all responses could be received. If subroutine uses pointer response, check the appropriate array count for any responses. If the subroutine uses callback response, the callback will have been invoked for any responses received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_class_query_rsp_t**. If any of the query arguments are invalid, then only one response is returned, indicating the error.

If the management style of the resource class being queried is globalized, and the session scope is DM, one response will be returned for each peer domain identified by the *pd_names* and *name_count* parameters. If *pd_names* and *name_count* are, respectively, a NULL pointer and 0 (zero), a response will be returned for each peer domain within the management domain.

If the management style of the resource class is subdivided or the session scope is not DM, only one response will be returned.

The response structure definition is:

```
typedef struct mc_class_query_rsp       mc_class_query_rsp_t;
struct mc_class_query_rsp {
     mc_errnum_t                        mc_error;
     ct_char_t                          *mc_class_name;
     ct_char_t                          *mc_peer_domain_name;
     mc_attribute_t                     *mc_attrs;
     ct_uint32_t                        mc_attr_count;
};
```

The fields of this structure contain the following:

mc_error
: If 0 (zero), this field indicates that the query was successful. Any other value is an error and indicates that the resource monitoring and control (RMC) subsystem or a resource manager could not provide some or all of the requested information. The error may also indicate that the command arguments were in error. The error codes imply which of the remaining fields in the structure are defined.

mc_class_name
: Specifies the name of the resource class that was queried and whose attributes are contained in this response.

mc_peer_domain_name
: Specifies the name of the peer domain from which the attributes were obtained. This is a NULL string if the response is from a node not in a peer domain.

mc_attrs
: A pointer to an array of the requested attributes.

mc_attr_count
: Indicates the number of entries in the *mc_attrs* array.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_class_query_d_*" on page 42
This subroutine queries the RMC subsystem to obtain the dynamic attribute values of a resource class.

"mc_class_set_*" on page 53
This subroutine sets persistent attribute values of a resource class.

"mc_query_p_select_*" on page 151
This subroutine queries the RMC subsystem to obtain the persistent attribute values of one or more resources of a resource class.

“mc_query_d_select_*” on page 138
This subroutine queries the RMC subsystem to obtain the dynamic attribute values of one or more resources of a resource class.

mc_class_set_*

This subroutine sets persistent attribute values of a resource class.

Purpose

Sets persistent attribute values of a resource class.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_class_set_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_class_set_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_class_set_bp(
          mc_sess_hndl_t            sess_hndl,
          mc_class_set_rsp_t        **rsp_array,
          ct_uint32_t               *array_cnt,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 **pd_names,
          ct_uint32_t               name_count,
          mc_attribute_t            *attrs,
          ct_uint32_t               count)
```

- The **mc_class_set_ap** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, the subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_class_set_ap(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_class_set_rsp_t        **rsp_array,
          ct_uint32_t               *array_cnt,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 **pd_names,
          ct_uint32_t               name_count,
          mc_attribute_t            *attrs,
          ct_uint32_t               count)
```

- The **mc_class_set_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_class_set_bc(
          mc_sess_hndl_t            sess_hndl,
          mc_class_set_cb_t         *set_cb,
          void                      *set_cb_arg,
          ct_char_t                 *rsrc_class_name,
```

```
        ct_char_t                   **pd_names,
        ct_uint32_t                 name_count,
        mc_attribute_t              *attrs,
        ct_uint32_t                 count)
```

The definition for the response callback is:

```
typedef void (mc_class_set_cb_t)(mc_sess_hndl_t,
mc_class_set_rsp_t *,
void *);
```

- The **mc_class_set_ac** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, the subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_class_set_ac(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_class_set_cb_t         *set_cb,
          void                      *set_cb_arg,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 **pd_names,
          ct_uint32_t               name_count,
          mc_attribute_t            *attrs,
          ct_uint32_t               count)
```

The definition for the response callback is:

```
typedef void (mc_class_set_cb_t)(mc_sess_hndl_t,
mc_class_set_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
: The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
: The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

set_cb
: Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

set_cb_arg
: Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

rsrc_class_name
: Identifies the resource class whose persistent attribute values are to be set.

pd_names

Table 18. mc_class_set_* subroutine conditional pd_names parameter functions

If:	Then:
The management style of the resource class is globalized and the session scope is DM	The parameter should be a pointer to an array of *name_count* peer domain names. Since a session scope of DM refers to a CSM management domain, and, since such a domain can contain multiple peer domains, this enables the application to specify the peer domain(s) where the class persistent attributes should be set. To specify all peer domains within the management domain, this parameter should be a NULL pointer, and the *name_count* parameter should be 0 (zero).
The management style of the resource class is subdivided or the session scope is not DM	This parameter should be a NULL pointer, and the *name_count* parameter should be 0 (zero).

name_count
: Identifies the number of pointers in the *pd_names* array. If *pd_names* is a NULL pointer, this parameter must be 0 (zero).

attrs Specifies the persistent class attributes to be set and their new values using a pointer to an array of *count* elements of type **mc_attribute_t**. Each element in the array specifies a persistent attribute of the resource class and a value.

count Specifies the number of elements in the *attrs* array.

OUTPUT

rsp_array
: A pointer to a location in which the RMC API will return a pointer to a response array of *array_cnt* elements. This parameter applies only to variations of this subroutine that use the pointer response method.

array_cnt
: Identifies a location in which the RMC API will return the number of elements in the response array *rsp_array*. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_class_set_*** subroutines can be used by the application to set the persistent attribute values of a resource class.

The response for these subroutines is a structure of type **mc_class_set_rsp_t**, and is described in Response structure.

This command cannot be used in an ordered command group.

Return values

For the **mc_class_set_bp** and **mc_class_set_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and one response has been received and processed.

For the **mc_class_set_ap** and **mc_class_set_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_EORDERGROUP
An attempt was made to add the command to an ordered command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_EINVALIDDATATYPE
Invalid attribute data type specified.

MC_EINVALIDVALUEPTR
Invalid attribute value pointer specified.

MC_EINVALIDSDTYPE
Invalid structured data subtype specified.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before all

responses could be received. If subroutine uses pointer response, check the appropriate array count for any responses. If the subroutine uses callback response, the callback will have been invoked for any responses received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_class_set_rsp_t**.

If the management style of the resource class is globalized and the session scope is DM, one response will be returned for each peer domain identified by the *pd_names* and *name_count* parameters. If *pd_names* and *name_count* are, respectively, a NULL pointer and 0, a response will be returned for each peer domain within the management domain.

If the management style of the resource class is subdivided and the session scope is not DM, only one response will be returned.

If any of the set attribute arguments are invalid. then only one response is returned, indicating the error.

The response structure definition is:

```
typedef struct mc_class_set_rsp         mc_class_set_rsp_t;
struct mc_class_set_rsp {
     mc_errnum_t                        mc_error;
     ct_char_t                          *mc_class_name;
     ct_char_t                          *mc_peer_domain_name;
     mc_error_attr_t                    *mc_error_attrs;
     ct_uint32_t                        mc_attr_count;
};
```

The fields of this structure contain the following:

mc_error
If 0 (zero), this field indicates that the command was successful, and, if the *mc_attr_count* is 0 (zero), all the specified attributes were set. If the *mc_attr_count* field is not 0 (zero), the *mc_error_attrs* field contains a pointer to an array of attributes that could not be set. Attributes not included in the array have been set. If the *mc_error* field is not 0 (zero), the value indicates the error.

mc_class_name

If the command is successful, this field is a pointer to the name of the resource class where attribute were set. If an error is indicated by the *mc_error* field, this field is a pointer to the name of the resource class specified on the command.

mc_peer_domain_name
Specifies the name of the peer domain where the attributes were set. This is a NULL string if the response is from a node not in a peer domain.

mc_error_attrs
If no error is indicated by the *mc_error* field, and only some of the attributes could be set, this field is an array of *mc_attr_count* elements of type **mc_error_attr_t**.

```
typedef struct mc_error_attr        mc_error_attr_t;
struct mc_error_attr {
     mc_errnum_t                    mc_error;
     ct_char_t                      *mc_at_name;
};
```

Each element in the array identifies an attribute that could not be set. The fields of this structure contain the following:

mc_error
Indicates the reason the attribute could not be set.

mc_at_name
Indicates the name of the attribute that could not be set.

mc_attr_count
Indicates the number of entries in the *mc_error_attrs* array.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_class_query_p_*" on page 47
This subroutine queries the RMC subsystem to obtain the persistent attribute values of a resource class.

"mc_set_handle_*" on page 194
This subroutine sets persistent attribute values of a resource identified by a resource handle.

"mc_set_select_*" on page 199
This subroutine sets persistent attribute values of one or more resources of a particular resource class. The resources are identified by attribute selection.

"mc_free_response" on page 75
This subroutine frees a response or event notification structure.

mc_define_resource_*

This subroutine defines a new resource.

Purpose

Defines a new resource.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_define_resource_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_define_resource_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_define_resource_bp(
          mc_sess_hndl_t            sess_hndl,
          mc_define_rsrc_rsp_t      **response,
          ct_char_t                 *rsrc_class_name,
          mc_attribute_t            *attrs,
          ct_uint32_t               count,
          ct_structured_data_t      *data)
```

- The **mc_define_resource_ap** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_define_resource_ap(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_define_rsrc_rsp_t      **response,
          ct_char_t                 *rsrc_class_name,
          mc_attribute_t            *attrs,
          ct_uint32_t               count,
          ct_structured_data_t      *data)
```

- The **mc_define_resource_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_define_resource_bc(
          mc_sess_hndl_t            sess_hndl,
          mc_def_rsrc_cb_t          *def_rsrc_cb,
          void                      *def_rsrc_cb_arg,
          ct_char_t                 *rsrc_class_name,
          mc_attribute_t            *attrs,
          ct_uint32_t               count,
          ct_structured_data_t      *data)
```

  The definition for the response callback is:

```
typedef void (mc_def_rsrc_cb_t)(mc_sess_hndl_t,
mc_define_rsrc_rsp_t *,
void *);
```

- The **mc_define_resource_ac** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_define_resource_ac(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_def_rsrc_cb_t          *def_rsrc_cb,
          void                      *def_rsrc_cb_arg,
          ct_char_t                 *rsrc_class_name,
          mc_attribute_t            *attrs,
          ct_uint32_t               count,
          ct_structured_data_t      *data)
```

  The definition for the response callback is:

```
typedef void (mc_def_rsrc_cb_t)(mc_sess_hndl_t,
mc_define_rsrc_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
: The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
: The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

def_rsrc_cb
Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

def_rsrc_cb_arg
Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

rsrc_class_name
Identifies the resource class of the resource to be defined.

attrs Specifies persistent attribute values for the resource using a pointer to an array of *count* elements of type **mc_attribtue_t**. Each element in the array specifies a persistent attribute of the resource and a value.

count Specifies the number of attributes in the **attrs** array.

data A pointer to structured data containing resource-class specific options for defining a resource. To accept the default values (or if the resource class does not define options) for defining a resource, the *data* parameter should be a NULL pointer.

To obtain the syntax and semantics for the structured data required by the resource class for specifying define resource options, the application can use the **mc_qdef_sd_*** subroutines.

OUTPUT

response
A pointer to a location in which the RMC API will return a pointer to the response. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_define_resource_*** subroutines can be used by the application to define a new resource. The resource manager associated with the proposed resource will create the actual resource instance. To define a new resource, the application identifies the resource class (using the *rsrc_class_name* parameter) and one or more persistent attribute values (using the *attrs* parameter). If the resource manager accepts structured data as options for defining a resource, the application can provide this using the *data* parameter.

The response for these subroutines is a structure of type **mc_define_rsrc_rsp_t**, and is described in Response structure.

This command cannot be used in an ordered command group.

Security

To define a new resource, the user of the calling application must have either the **d** or **w** permission specified in an ACL entry for the associated resource class.

Return values

For the **mc_define_resource_bp** and **mc_define_resource_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and a response has been received and processed.

For the **mc_define_resource_ap** and **mc_define_resource_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_EORDERGROUP
An attempt was made to add the command to an ordered command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before the response could be received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before the response could be received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_EINVALIDDATATYPE
Invalid attribute data type specified.

MC_EINVALIDVALUEPTR
Invalid attribute value pointer specified.

MC_EINVALIDSDTYPE
Invalid structured data subtype specified.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before the response could be received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_define_rsrc_rsp_t**. If any of the define resource arguments are invalid, then the *mc_error* field in the response indicates an error. This command results in only one response.

The response structure definition is:

```
typedef struct mc_define_rsrc_rsp       mc_define_rsrc_rsp_t;
struct mc_define_rsrc_rsp {
     mc_errnum_t                        mc_error;
     ct_char_t                          *mc_class_name;
     ct_resource_handle_t               mc_rsrc_hndl;
};
```

The fields of this structure contain the following:

mc_error
: If 0 (zero), this field indicates that the command was successful. Any other value is an error, and indicates that the RMC subsystem or a resource manager could not complete the command. The error may also indicate that the command arguments were in error. The error codes imply which of the remaining fields in the structure are defined.

mc_class_name
: The name of the resource class in which a new resource instance is defined.

mc_rsrc_hndl
: The resource handle of the new resource instance.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_undefine_resource_*" on page 219
This subroutine removes a resource from the RMC subsystem.

"mc_qdef_sd_*" on page 120
This subroutine queries the RMC subsystem to obtain the definition of structured data.

"mc_qdef_p_attribute_*" on page 108
This subroutine queries the RMC subsystem to obtain the persistent attribute definitions for a resource or resource class.

mc_dispatch

This subroutine provides a thread to the RMC API to enable it to invoke a callback to process a response or event notification.

Purpose

Provides a thread to the RMC API to enable it to invoke a callback to process a response or event notification.

Library

RMC Library (**libct_mc.a**)

Syntax

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_dispatch(
          mc_sess_hndl_t               session_hndl,
          mc_dispatch_opts_t           options)
```

Parameters

INPUT

session_hndl
The session handle that identifies the RMC subsystem session for which this thread is being provided. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

options Specifies thread behavior using one of the following options:

MC_DISPATCH_OPTS_WAIT
If no response or event notification needs to be processed, the thread blocks execution until one does need to be processed. The subroutine returns after a response or event notification is processed.

MC_DISPATCH_OPTS_ASSIGN
The subroutine does not return. The thread is kept by the RMC API to process future response or event notifications.

When either of these options are used, this subroutine returns if the session is interrupted or ended by a call to the **mc_end_session** subroutine.

Description

The **mc_dispatch** subroutine can be used by the application to provide a thread to the RMC API. This thread is used by the RMC API to invoke a callback to process a response or event notification received from the RMC subsystem for the session specified by the *session_hndl* parameter.

A multithreaded application can call the subroutine a number of times to provide sufficient threads to the RMC API to process expected responses or event notifications. Such an approach will be especially needed parallelize callback execution.

To prevent the **mc_dispatch** subroutine from blocking when using the MC_DISPATCH_OPTS_WAIT option, the application can obtain a descriptor by calling the **mc_get_descriptor** subroutine. The descriptor is made ready for read whenever the RMC API has received a response or event notification for the session and there is no application thread available to the API to invoke the necessary callback. By using this in a **select** or **poll** system call, the application can detect when it should call the **mc_dispatch** subroutine.

Return values

A return value of 0 (zero) indicates that a response or event notification has been successfully processed. Any other return value is an error.

The following errors can be returned by this subroutine. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_EINVALIDOPT
The specified option is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_get_descriptor" on page 76
This subroutine returns a descriptor that can be used in a **select** or **poll** system call.

mc_end_session
This subroutine ends a session with the RMC subsystem.

Purpose

Ends a session with the RMC subsystem.

Library

RMC Library (**libct_mc.a**)

Syntax

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_end_session(
          mc_sess_hndl_t               session_hndl)
```

Parameters

INPUT

session_hndl
The session handle that identifies the RMC subsystem session to be ended. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

Description

The **mc_end_session** subroutine can be used by the application to end a session with the RMC subsystem. When ending a session with the subroutine, any application threads that are blocked in the RMC API subroutines, including the **mc_dispatch** subroutine, will return an error indicating the session has ended. Any responses or event notifications that are being processed by callbacks are allowed to complete, and then the associated threads will return with an error indicating that the session has ended. However, the **mc_end_session** subroutine does not wait for the callbacks to finish, but will return immediately.

Return values

A return value of 0 (zero) indicates that the session with the RMC subsystem has ended. The session handle can no longer be used to send commands to, or receive responses from, the RMC subsystem. Any non-zero value is an error. If an error is returned, the session will not have ended.

The following errors can be returned by this subroutine. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_ESESSENDED
The session has been ended.

MC_EINVALIDSESS
The specified session handle is invalid.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_session_info" on page 193
This subroutine gets information about a session.

"mc_start_session" on page 206
This subroutine establishes a session with the RMC subsystem.

mc_enumerate_permitted_rsrcs_*

This subroutine enumerates the resources of a resource class using attribute selection. Enumerates only those resources for which the calling application has specified permissions.

Purpose

Enumerates the resources of a resource class using attribute selection. Enumerates only those resources for which the calling application has specified permissions.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_enumerate_permitted_rsrcs_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_enumerate_permitted_rsrcs_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_enumerate_permitted_rsrcs_bp(
          mc_sess_hndl_t            sess_hndl,
          mc_enumerate_rsp_t        **rsp_array,
```

```
        ct_uint32_t             *array_cnt,
        ct_char_t               *rsrc_class_name,
        ct_char_t               *select_attrs
        ct_uint32_t             perms)
```

- The **mc_enumerate_permitted_rsrcs_ap** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_enumerate_permitted_rsrcs_ap(
        mc_cmdgrp_hndl_t        cmdgrp_hndl,
        mc_enumerate_rsp_t      **rsp_array,
        ct_uint32_t             *array_cnt,
        ct_char_t               *rsrc_class_name,
        ct_char_t               *select_attrs
        ct_uint32_t             perms)
```

- The **mc_enumerate_permitted_rsrcs_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_enumerate_permitted_rsrcs_bc(
        mc_sess_hndl_t          sess_hndl,
        mc_enumerate_cb_t       *enumerate_cb,
        void                    *enumerate_cb_arg,
        ct_char_t               *rsrc_class_name,
        ct_char_t               *select_attrs
        ct_uint32_t             perms)
```

  The definition for the response callback is:

```
typedef void (mc_enumerate_cb_t)(mc_sess_hndl_t,
mc_enumerate_rsp_t *,
void *);
```

- The **mc_enumerate_permitted_rsrcs_ac** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_enumerate_permitted_rsrcs_ac(
        mc_cmdgrp_hndl_t        cmdgrp_hndl,
        mc_enumerate_cb_t       *enumerate_cb,
        void                    *enumerate_cb_arg,
        ct_char_t               *rsrc_class_name,
        ct_char_t               *select_attrs
        ct_uint32_t             perms)
```

  The definition for the response callback is:

```
typedef void (mc_enumerate_cb_t)(mc_sess_hndl_t,
mc_enumerate_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl

The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

enumerate_cb
Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

enumerate_cb_arg
Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

rsrc_class_name
Pointer to a resource class name. Identifies the resource class whose resources are to be enumerated.

select_attrs
A pointer to a selection string expression that limits the resource enumeration to a subset of resources in the resource class. The selection string expression filters the available resources by one or more persistent attributes of the resource class. Only the resources that match the selection string (and whose associated resource ACL contains the permissions specified by the *perms* parameter for the user of the calling application) will be enumerated.

If this parameter is a NULL pointer, then all resources of the resource class identified by the *rsrc_class_name* parameter are selected.

perms Identifies the permissions required by the user of the calling application. Resource handles will be returned only if the user has the correct permissions in the resource's resource ACL.

OUTPUT

rsp_array
A pointer to a location in which the RMC API will return a pointer to a response array of *array_cnt* elements. This parameter applies only to variations of this subroutine that use the pointer response method.

array_cnt
Identifies a location in which the RMC API will return the number of elements in the response array *rsp_array*. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_enumerate_permitted_rsrcs_*** subroutines can be used by the application to enumerate the resources of a specified resource class using attribute selection. Unlike the **mc_enumerate_resources_*** subroutines, these subroutines enumerate only those resources for which the calling application has correct permissions (as specified by the *perms* parameter).

The response for these subroutines is a structure of type **mc_enumerate_rsp_t**, and is described in Response structure.

This command cannot be used in an ordered command group.

Security

To enumerate resources of a resource class, the user of the calling application must have either **l** or **r** permission specified in an ACL entry for the resource class.

Return values

For the **mc_enumerate_permitted_rsrcs_bp** and **mc_enumerate_permitted_rsrcs_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and one or more one or more response have been received and processed.

For the **mc_enumerate_permitted_rsrcs_ap** and **mc_enumerate_permitted_rsrcs_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_EORDERGROUP
An attempt was made to add the command to an ordered command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_EINVALIDCNTPTR
Invalid response count pointer specified.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before all responses could be received. If subroutine uses pointer response, check the appropriate array count for any responses. If the subroutine uses callback response, the callback will have been invoked for any responses received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_enumerate_rsp_t**. If any of the enumerate arguments are invalid, then the response indicates the error.

The response structure definition is:

```
typedef struct mc_enumerate_rsp         mc_enumerate_rsp_t;
struct mc_enumerate_rsp {
     mc_errnum_t                        mc_error;
     ct_char_t                          *mc_class_name;
     ct_resource_handle_t               *mc_rsrc_handles;
     ct_uint32_t                        mc_rsrc_handle_count;
};
```

The fields of this structure contain the following:

mc_error
If 0 (zero), this field indicates that the enumeration was successful. Any other value is an error and indicates that the RMC subsystem or a resource manager could not provide some or all of the requested information. The error may also indicate that the command arguments were in error. The error codes imply which of the remaining fields in the structure are defined.

mc_class_name
Specifies the name of the resource class whose resources are contained in this response.

mc_rsrc_handles
A pointer to an array of resource handles for the selected resources.

mc_rsrc_handle_count
Indicates the number of resource handles in the array

Location

/usr/lib/libct_mc.a

Related reference:

"mc_enumerate_resources_*"
This subroutine enumerates the resources of a resource class using attribute selection.

mc_enumerate_resources_*

This subroutine enumerates the resources of a resource class using attribute selection.

Purpose

Enumerates the resources of a resource class using attribute selection.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_enumerate_resources_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_enumerate_resources_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_enumerate_resources_bp(
          mc_sess_hndl_t            sess_hndl,
          mc_enumerate_rsp_t        **rsp_array,
          ct_uint32_t               *array_cnt,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 *select_attrs)
```

- The **mc_enumerate_resources_ap** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_enumerate_resources_ap(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_enumerate_rsp_t        **rsp_array,
          ct_uint32_t               *array_cnt,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 *select_attrs)
```

- The **mc_enumerate_resources_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_enumerate_resources_bc(
          mc_sess_hndl_t            sess_hndl,
          mc_enumerate_cb_t         *enumerate_cb,
          void                      *enumerate_cb_arg,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 *select_attrs)
```

  The definition for the response callback is:

```
typedef void (mc_enumerate_cb_t)(mc_sess_hndl_t,
mc_enumerate_rsp_t *,
void *);
```

- The **mc_enumerate_resources_ac** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_enumerate_resources_ac(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
```

```
mc_enumerate_cb_t          *enumerate_cb,
void                       *enumerate_cb_arg,
ct_char_t                  *rsrc_class_name,
ct_char_t                  *select_attrs)
```

The definition for the response callback is:

```
typedef void (mc_enumerate_cb_t)(mc_sess_hndl_t,
mc_enumerate_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
: The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
: The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

enumerate_cb
: Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

enumerate_cb_arg
: Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

rsrc_class_name
: Pointer to a resource class name. Identifies the resource class whose resources are to be enumerated.

select_attrs
: A pointer to a selection string expression that limits the resource enumeration to a subset of resources in the resource class. The selection string expression filters the available resources by one or more persistent attributes of the resource class. Only the resources that match the selection string will be enumerated.

  If this parameter is a NULL pointer, then all resources of the resource class identified by the *rsrc_class_name* parameter are selected.

OUTPUT

rsp_array
: A pointer to a location in which the RMC API will return a pointer to a response array of *array_cnt* elements. This parameter applies only to variations of this subroutine that use the pointer response method.

array_cnt
: Identifies a location in which the RMC API will return the number of elements in the response array *rsp_array*. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_enumerate_resources_*** subroutines can be used by the application to enumerate the resources of a specified resource class using attribute selection.

The response for these subroutines is a structure of type **mc_enumerate_rsp_t**, and is described in Response structure.

This command cannot be used in an ordered command group.

Security

To enumerate resources of a resource class, the user of the calling application must have either **l** or **r** permission specified in an ACL entry for the resource class.

Return values

For the **mc_enumerate_resources_bp** and **mc_enumerate_resources_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and one or more responses have been received and processed.

For the **mc_enumerate_resources_ap** and **mc_enumerate_resources_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_EORDERGROUP
An attempt was made to add the command to an ordered command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before all responses could be

received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_EINVALIDCNTPTR
Invalid response count pointer specified.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before all responses could be received. If subroutine uses pointer response, check the appropriate array count for any responses. If the subroutine uses callback response, the callback will have been invoked for any responses received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_enumerate_rsp_t**. If any of the enumerate arguments are invalid, then the response indicates the error.

The response structure definition is:

```
typedef struct mc_enumerate_rsp         mc_enumerate_rsp_t;
struct mc_enumerate_rsp {
     mc_errnum_t                        mc_error;
     ct_char_t                          *mc_class_name;
     ct_resource_handle_t               *mc_rsrc_handles;
     ct_uint32_t                        mc_rsrc_handle_count;
};
```

The fields of this structure contain the following:

mc_error
If 0 (zero), this field indicates that the enumeration was successful. Any other value is an error and indicates that the RMC subsystem or a resource manager could not provide some or all of the requested information. The error may also indicate that the command arguments were in error. The error codes imply which of the remaining fields in the structure are defined.

mc_class_name
Specifies the name of the resource class whose resources are contained in this response.

mc_rsrc_handles
A pointer to an array of resource handles for the selected resources.

mc_rsrc_handle_count
Indicates the number of resource handles in the array

Location

/usr/lib/libct_mc.a

Related reference:

"mc_enumerate_permitted_rsrcs_*" on page 65
This subroutine enumerates the resources of a resource class using attribute selection. Enumerates only those resources for which the calling application has specified permissions.

"mc_free_response" on page 75
This subroutine frees a response or event notification structure.

mc_free_descriptor

This subroutine frees a descriptor that was previously obtained by the **mc_get_descriptor** subroutine.

Purpose

Frees a descriptor that was previously obtained by the **mc_get_descriptor** subroutine.

Library

RMC Library (**libct_mc.a**)

Syntax

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_free_descriptor(
          mc_sess_hndl_t              session_hndl,
          int                         descriptor)
```

Parameters

INPUT

session_hndl
: The session handle that identifies the RMC subsystem session for which the descriptor was obtained. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

descriptor
: Identifies the descriptor to be freed.

Description

The **mc_free_descriptor** subroutine can be used by the application to free a descriptor that was previously obtained by the **mc_get_descriptor** subroutine. Once this subroutine returns successfully, the descriptor can no longer be used in a **select** or **poll** system call.

Return values

A return value of 0 (zero) indicates that the descriptor has been successfully freed and can no longer be used in a select or poll system call. Any other value is an error and indicates that the descriptor has not been freed.

The following errors can be returned by this subroutine. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_ESESSENDED
The session has been ended.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_EINVALIDDSCRP
The specified descriptor is invalid.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_get_descriptor" on page 76
This subroutine returns a descriptor that can be used in a **select** or **poll** system call.

mc_free_response

This subroutine frees a response or event notification structure.

Purpose

Frees a response or event notification structure.

Library

RMC Library (**libct_mc.a**)

Syntax

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_free_response(
          void                          *rsp_ptr)
```

Parameters

INPUT

rsp_ptr A pointer to the response, response array or event notification, previously passed to the application by the RMC API, to be freed.

Description

The **mc_free_response** subroutine can be used by the application to free the storage used by a response, response array, or event notification structure.

If the response array contains more than one element, note that the entire array must be freed. The *rsp_prt* parameter must not point to a response array element other than the first element in the array.

Return values

A return value of 0 (zero) indicates that the storage has been successfully freed. Any other value is an error and indicates that the storage has not been freed.

The following errors can be returned by this subroutine. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDDATA
The specified pointer does not point to a response or event notification structure.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_class_query_d_*" on page 42
This subroutine queries the RMC subsystem to obtain the dynamic attribute values of a resource class.

"mc_class_set_*" on page 53
This subroutine sets persistent attribute values of a resource class.

"mc_enumerate_resources_*" on page 69
This subroutine enumerates the resources of a resource class using attribute selection.

"mc_invoke_action_*" on page 78
This subroutine invokes an action on a resource.

"mc_offline_*" on page 88
This subroutine sends a request to the RMC subsystem to take a resource offline.

"mc_qdef_actions_*" on page 97
This subroutine queries the RMC subsystem to obtain the definitions of resource class actions.

"mc_qdef_p_attribute_*" on page 108
This subroutine queries the RMC subsystem to obtain the persistent attribute definitions for a resource or resource class.

"mc_reg_event_handle_*" on page 168
This subroutine registers a resource event with the RMC subsystem using a resource handle.

mc_get_descriptor

This subroutine returns a descriptor that can be used in a **select** or **poll** system call.

Purpose

Returns a descriptor that can be used in a **select** or **poll** system call.

Library

RMC Library (**libct_mc.a**)

Syntax

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_get_descriptor(
          mc_sess_hndl_t              session_hndl,
          int                         *descriptor)
```

Parameters

INPUT

session_hndl
The session handle that identifies the RMC subsystem session for which this descriptor is being obtained. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

OUTPUT

descriptor
The address, allocated by the application, for the descriptor. The subroutine, if successful, returns the descriptor at this address.

Description

The **mc_get_descriptor** subroutine can be used by the application to obtain a descriptor that can be used in the **select** or **poll** system call. The descriptor is made ready for read whenever the RMC API has received a response or event notification for the session identified by the *session_hndl* parameter and there is no thread available to the RMC API to process the response.

The application can use this descriptor in a **select** or **poll** system call to determine when it should call the **mc_dispatch** subroutine. This is intended for applications that call the **mc_dispatch** subroutine with the **MC_DISPATCH_OPTS_WAIT** option. When the **MC_DISPATCH_OPTS_WAIT** option is specified, the thread provided by the **mc_dispatch** subroutine will block execution if no response or event notification needs to be processed. Using a descriptor returned by the **mc_get_descriptor** subroutine in a **select** or **poll** system call enables the application to call the **mc_dispatch** subroutine only when the thread is needed.

The descriptor is also made ready to read by the RMC API if the session is interrupted and there are no application threads blocked in the RMC API. In this situation, if the application were to call the **mc_dispatch** subroutine, and error would be returned indicating the session was interrupted.

The application can call this subroutine only once during a session, unless it calls the **mc_free_descriptor** subroutine to free the specified descriptor.

Although this subroutine is intended for single-threaded applications, it can also be used by multithreaded applications.

Return values

A return value of 0 (zero) indicates that a descriptor has been successfully returned at the address specified by the *descriptor* parameter. Any other value is an error and indicates that the descriptor has not been returned.

The following errors can be returned by this subroutine. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ENODSCRP
A descriptor cannot be allocated by the API.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_EBUSY
A descriptor has already been allocated to the specified session.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_dispatch" on page 62
This subroutine provides a thread to the RMC API to enable it to invoke a callback to process a response or event notification.

"mc_free_descriptor" on page 74
This subroutine frees a descriptor that was previously obtained by the **mc_get_descriptor** subroutine.

mc_invoke_action_*

This subroutine invokes an action on a resource.

Purpose

Invokes an action on a resource.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_invoke_action_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_invoke_action_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_invoke_action_bp(
          mc_sess_hndl_t            sess_hndl,
          mc_action_rsp_t           **rsp_array,
          ct_uint32_t               *array_cnt,
          ct_resource_handle_t      rsrc_hndl,
          ct_char_t                 *action_name,
          ct_structured_data_t      *data)
```

- The **mc_invoke_action_ap** subroutine adds the command to a command group. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_invoke_action_ap(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_action_rsp_t           **rsp_array,
          ct_uint32_t               *array_cnt,
          ct_resource_handle_t      rsrc_hndl,
          ct_char_t                 *action_name,
          ct_structured_data_t      *data)
```

- The **mc_invoke_action_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_invoke_action_bc(
          mc_sess_hndl_t            sess_hndl,
          mc_action_cb_t            *action_cb,
          void                      *action_cb_arg,
          ct_resource_handle_t      rsrc_hndl,
          ct_char_t                 *action_name,
          ct_structured_data_t      *data)
```

The definition for the response callback is:

```
typedef void (mc_action_cb_t)(mc_sess_hndl_t,
mc_action_rsp_t *,
void *);
```

- The **mc_invoke_action_ac** subroutine adds the command to a command group. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_invoke_action_ac(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_action_cb_t            *action_cb,
          void                      *action_cb_arg,
          ct_resource_handle_t      rsrc_hndl,
          ct_char_t                 *action_name,
          ct_structured_data_t      *data)
```

The definition for the response callback is:

```
typedef void (mc_action_cb_t)(mc_sess_hndl_t,
mc_action_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
: The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
: The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

action_cb
: Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

action_cb_arg
: Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

rsrc_hndl
: The resource handle that identifies the resource upon which the action is to be invoked. A resource handle is returned in the response structure for many RMC API subroutines including the **mc_define_resource_*** subroutine. An array of resource handles for resources of a particular resource class is returned in the response structure for the **mc_enumerate_resources_*** and

mc_enumerate_permitted_rsrcs_* subroutines. To validate the resource handle before calling this subroutine, the application can call one of the **mc_validate_rsrc_hndl_*** subroutines.

action_name
Pointer to an action name. Identifies the action to be invoked.

data A pointer to structured data to be used as input to the action. If the action identified by the *action_name* parameter does not accept input, the data parameter should be a NULL pointer.

To obtain the format of input structured data accepted by the action, the application can use the **mc_get_sd_*** subroutines.

OUTPUT

rsp_array
A pointer to a location in which the RMC API will return a pointer to a response array of *array_cnt* elements. This parameter applies only to variations of this subroutine that use the pointer response method.

array_cnt
Identifies a location in which the RMC API will return the number of elements in the response array *rsp_array*. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_invoke_action_*** subroutines can be used by the application to invoke an action (identified by the *action_name* parameter) on a resource (identified by the *rsrc_hndl* parameter). If the action accepts structured data as input, the application can provide this using the *data* parameter.

The response for these subroutines is a structure of type **mc_action_rsp_t**, and is described in Response structure.

Security

To invoke a resource action, the user of the calling application must have the permission defined for the action specified in an ACL entry for the resource. If no permission is defined for the action, then the user of the calling application must have either the **s** or **w** permission specified in an ACL entry for the resource.

Return values

For the **mc_invoke_action_bp** and **mc_invoke_action_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and one or more responses have been received and processed.

For the **mc_invoke_action_ap** and **mc_invoke_action_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_ETARGETMISMATCH
The target specified for the command does not match the target of the command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_EINVALIDDATATYPE
Invalid attribute data type specified.

MC_EINVALIDVALUEPTR
Invalid attribute value pointer specified.

MC_EINVALIDSDTYPE
Invalid structured data subtype specified.

MC_EINVALIDCNTPTR
Invalid response count pointer specified.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before all responses could be received. If subroutine uses pointer response, check the appropriate array count for any responses. If the subroutine uses callback response, the callback will have been invoked for any responses received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_action_rsp_t**. This command may have more than one response.

The response structure definition is:

```
typedef struct mc_action_rsp              mc_action_rsp_t;
struct mc_action_rsp {
     mc_errnum_t                         mc_error;
     ct_resource_handle_t                mc_rsrc_hndl;
     ct_structured_data_t                **mc_data;
     ct_uint32_t                         mc_count;
};
```

The fields of this structure contain the following:

mc_error
: If 0 (zero), this field indicates that the command was successful, and the action was successfully processed. Any other value is an error, and indicates a problem in processing the action.

mc_rsrc_hndl
: The resource handle of the resource that was the target of the command.

mc_data
: Optionally contains a block of response data from the action. If the *mc_count* field is zero, then this field is undefined. If the *mc_count* field is non-zero, this field is a pointer to an array of pointers to structured data (SD) resulting from the action. Each SD in the response has an identical format, as defined for the action resulting in the response. More than one response may be generated from invoking an action, each containing one or more SDs.

  To obtain the format of the structured data resulting from the action, the application can use the **mc_qdef_sd_*** subroutines.

mc_count
: Indicates the number of entries in the *mc_data* array.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_free_response" on page 75
This subroutine frees a response or event notification structure.

"mc_invoke_class_action_*"
This subroutine invokes an action on a resource class.

"mc_qdef_sd_*" on page 120
This subroutine queries the RMC subsystem to obtain the definition of structured data.

mc_invoke_class_action_*

This subroutine invokes an action on a resource class.

Purpose

Invokes an action on a resource class.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_invoke_class_action_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_invoke_class_action_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_invoke_class_action_bp(
          mc_sess_hndl_t            sess_hndl,
          mc_class_action_rsp_t     **rsp_array,
          ct_uint32_t               *array_cnt,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 *action_name,
          ct_char_t                 **names,
          ct_uint32_t               name_count,
          mc_list_usage_t           list_use,
          ct_structured_data_t      *data)
```

- The **mc_invoke_class_action_ap** subroutine adds the command to a command group. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_invoke_class_action_ap(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_class_action_rsp_t     **rsp_array,
          ct_uint32_t               *array_cnt,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 *action_name,
          ct_char_t                 **names,
          ct_uint32_t               name_count,
          mc_list_usage_t           list_use,
          ct_structured_data_t      *data)
```

- The **mc_invoke_class_action_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_invoke_class_action_bc(
          mc_sess_hndl_t            sess_hndl,
          mc_class_action_cb_t      *action_cb,
          void                      *action_cb_arg,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 *action_name,
          ct_char_t                 **names,
          ct_uint32_t               name_count,
          mc_list_usage_t           list_use,
          ct_structured_data_t      *data)
```

  The definition for the response callback is:

```
typedef void (mc_class_action_cb_t)(mc_sess_hndl_t,
mc_class_action_rsp_t *,
void *);
```

- The **mc_invoke_class_action_ac** subroutine adds the command to a command group. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_invoke_class_action_ac(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
```

```
mc_class_action_cb_t       *action_cb,
void                       *action_cb_arg,
ct_char_t                  *rsrc_class_name,
ct_char_t                  *action_name,
ct_char_t                  **names,
ct_uint32_t                name_count,
mc_list_usage_t            list_use,
ct_structured_data_t       *data)
```

The definition for the response callback is:

```
typedef void (mc_class_action_cb_t)(mc_sess_hndl_t,
mc_class_action_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
: The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
: The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

action_cb
: Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

action_cb_arg
: Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

rsrc_class_name
: Pointer to a resource class name. Identifies the resource class upon which the action is to be invoked.

action_name
: Pointer to an action name. Identifies the action to be invoked.

names

Table 19. mc_invoke_class_action_* subroutine conditional names parameter functions

If:	Then:
The management style of the resource class is subdivided.	This parameter is a pointer to an array of *name_count* node names, identifying where the class action should be invoked. The *list_use* parameter must specify the **MC_LIST_USAGE_NODES** setting. To specify that the class action should be invoked on all nodes of the cluster, this parameter should be a NULL pointer and the *name_count* parameter should be 0 (zero).

Table 19. mc_invoke_class_action_* subroutine conditional names parameter functions (continued)

If:	Then:
The management style of the resource class is globalized, and the session scope is DM	This parameter is a pointer to an array of *name_count* peer domain names identifying the peer domains where the class action should be invoked. The *list_use* parameter must specify the **MC_LIST_USAGE_PEER_DOMAINS** setting. To specify that the class action should be invoked on all peer domains within the management domain, this parameter should be a NULL pointer and the *name_count* parameter should be 0 (zero).

name_count
Identifies the number of pointers in the *names* array. If *names* is a NULL pointer, this parameter must be 0 (zero).

list_use
Indicates whether the *names* parameter is a pointer to an array of node names or peer domain names. Valid values are:

MC_LIST_USAGE_NODES
The *names* parameters identifies node names.

MC_LIST_USAGE_PEER_DOMAINS
The *names* parameter identifies peer domain names.

data A pointer to structured data to be used as input to the action. If the action identified by the *action_name* parameter does not accept input, the data parameter should be a NULL pointer.

To obtain the format of input structured data accepted by the action, the application can use the **mc_get_sd_*** subroutines.

OUTPUT

rsp_array
A pointer to a location in which the RMC API will return a pointer to a response array of *array_cnt* elements. This parameter applies only to variations of this subroutine that use the pointer response method.

array_cnt
Identifies a location in which the RMC API will return the number of elements in the response array *rsp_array*. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_invoke_class_action_*** subroutines can be used by the application to invoke an action (identified by the *action_name* parameter) on a resource class (identified by the *rsrc_class_name* parameter). If the action accepts structured data as input, the application can provide this using the *data* parameter.

The response for these subroutines is a structure of type **mc_class_action_rsp_t**, and is described in Response structure.

Security

To invoke a resource class action, the user of the calling application must have the permission defined for the action specified in an ACL entry for the resource class. If no permission is defined for the action, then the user of the calling application must have either the **s** or **w** permission specified in an ACL entry for the resource class.

Return values

For the **mc_invoke_class_action_bp** and **mc_invoke_class_action_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and one or more responses have been received and processed.

For the **mc_invoke_class_action_ap** and **mc_invoke_class_action_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_EORDERGROUP
An attempt was made to add the command to an ordered command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_EINVALIDDATATYPE
Invalid attribute data type specified.

MC_EINVALIDVALUEPTR
Invalid attribute value pointer specified.

MC_EINVALIDSDTYPE
Invalid structured data subtype specified.

MC_EINVALIDCNTPTR
Invalid response count pointer specified.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before all responses could be received. If subroutine uses pointer response, check the appropriate array count for any responses. If the subroutine uses callback response, the callback will have been invoked for any responses received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_class_action_rsp_t**. This command may have more than one response.

The response structure definition is:

```
typedef struct mc_class_action_rsp      mc_class_action_rsp_t;
struct mc_class_action_rsp {
     mc_errnum_t                        mc_error;
     ct_char_t                          *mc_class_name;
     ct_structured_data_t               **mc_data;
     ct_uint32_t                        mc_count;
     ct_char_t                          *mc_node_name;
     ct_char_t                          *mc_peer_domain_name;
};
```

The fields of this structure contain the following:

mc_error
If 0 (zero), this field indicates that the command was successful, and the action was successfully processed. Any other value is an error, and indicates a problem in processing the action.

mc_class_name
The resource class that was the target of the invoke action command.

mc_data
Optionally contains a block of response data from the action. If the *mc_count* field is zero, then this field is undefined. If the *mc_count* field is non-zero, this field is a pointer to an array of pointers to structured data (SD) resulting from the action. Each SD in the response has an identical format, as defined for the action resulting in the response. More than one response may be generated from invoking an action, each containing one or more SDs.

To obtain the format of the structured data resulting from the action, the application can use the **mc_qdef_sd_*** subroutines.

mc_count
The number of entries in the *mc_data* array.

mc_node_name
The primary node name of the node where the class action was invoked.

mc_peer_domain_name
The name of the peer domain where the class action was invoked. The name is a NULL string if the response is from a node not in a peer domain.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_invoke_action_*" on page 78
This subroutine invokes an action on a resource.

"mc_qdef_sd_*" on page 120
This subroutine queries the RMC subsystem to obtain the definition of structured data.

"mc_qdef_actions_*" on page 97
This subroutine queries the RMC subsystem to obtain the definitions of resource class actions.

mc_offline_*

This subroutine sends a request to the RMC subsystem to take a resource offline.

Purpose

Sends a request to the RMC subsystem to take a resource offline.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_offline_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_offline_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_offline_bp(
          mc_sess_hndl_t            sess_hndl,
          mc_rsrc_hndl_rsp_t        **response,
          mc_offline_opts_t         options,
          ct_resource_handle_t      rsrc_hndl,
          ct_structured_data_t      *data)
```

- The **mc_offline_ap** subroutine adds the command to a command group. If used in an ordered command group, however, please note that all other commands in the command group must also be the form that uses a resource handle to specify a command target. To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_offline_ap(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_rsrc_hndl_rsp_t        **response,
          mc_offline_opts_t         options,
          ct_resource_handle_t      rsrc_hndl,
          ct_structured_data_t      *data)
```

- The **mc_offline_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_offline_bc(
          mc_sess_hndl_t            sess_hndl,
          mc_offline_cb_t           *offline_cb,
          void                      *offline_cb_arg,
          mc_offline_opts_t         options,
          ct_resource_handle_t      rsrc_hndl,
          ct_structured_data_t      *data)
```

The definition for the response callback is:

```
typedef void (mc_offline_cb_t)(mc_sess_hndl_t,
mc_rsrc_hndl_rsp_t *,
void *);
```

- The **mc_offline_ac** subroutine adds the command to a command group. If used in an ordered command group, however, please note that all other commands in the command group must also be the form that uses a resource handle to specify a command target. To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_offline_ac(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_offline_cb_t           *offline_cb,
          void                      *offline_cb_arg,
          mc_offline_opts_t         options,
          ct_resource_handle_t      rsrc_hndl,
          ct_structured_data_t      *data)
```

The definition for the response callback is:

```
typedef void (mc_offline_cb_t)(mc_sess_hndl_t,
mc_rsrc_hndl_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
: The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
: The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

offline_cb
: Identifies the callback routine that will be invoked by the RMC API to return the command response to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

offline_cb_arg
: Identifies the argument that the RMC API will use to pass the command response to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

options Specifies one of the following:

MC_OFFLINE_OPTS_FAILED
The target state of the resource is to be *failed off-line*. If this option is not specified, the target state of the resource will be *off-line*.

MC_OFFLINE_OPTS_NONE
There are no options.

rsrc_hndl
The resource handle that identifies the resource to be taken offline. A resource handle is returned in the response structure for many RMC API subroutines including the **mc_define_resource_*** subroutine. An array of resource handles for resources of a particular resource class is returned in the response structure for the **mc_enumerate_resources_*** and **mc_enumerate_permitted_rsrcs_*** subroutines. To validate the resource handle before calling this subroutine, the application can call one of the **mc_validate_rsrc_hndl_*** subroutines.

data A pointer to structured data containing resource class specific options for taking the resource offline. To accept the default values (or if the resource class does not define options) for going offline, the data parameter should be a NULL pointer.

To obtain the syntax and semantics for the structured data required by the resource class for specifying offline options, the application can use the **mc_qdef_sd_*** subroutine.

OUTPUT

response
A pointer to a location in which the RMC API will return a pointer to the response. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_offline_*** subroutines can be used by the application to request the RMC subsystem to take a resource (identified by the *rsrc_hndl* parameter) offline. If the resource manager accepts structured data options for taking a resource offline, the application can provide this using the *data* parameter. The request is performed by the resource's associated resource manager. If the resource is online on multiple nodes, then it will be taken offline on each of the nodes.

The response for these subroutines is a structure of type **mc_rsrc_hndl_rsp_t**, and is described in Response structure.

If this command is used in an ordered command group, please note that all other commands in the command group must be the form that uses a resource handle to specify the command target. The same resource handle must be used on all commands in the command group.

Security

To take a resource offline, the user of the calling application must have either **o** or **w** permission specified in an ACL entry for this resource.

Return values

For the **mc_offline_bp** and **mc_offline_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and a response has been received and processed.

For the **mc_offline_ap** and **mc_offline_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_ETARGETMISMATCH
The target specified for the command does not match the target of the command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before the response could be received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before the response could be received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_EINVALIDDATATYPE
Invalid attribute data type specified.

MC_EINVALIDVALUEPTR
Invalid attribute value pointer specified.

MC_EINVALIDSDTYPE
Invalid structured data subtype specified.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before the response could be received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_rsrc_hndl_rsp_t**. Although the **mc_error** field of the **mc_rsrc_hndl_rsp_t** structure indicates whether or not the resource manager has successfully processed the command, it does not mean that the resource is offline. To determine if the resource is actually offline, the application must register an event to monitor the resource's OpState attribute value. This command results in only one response.

The response structure definition is:

```
typedef struct mc_rsrc_hndl_rsp         mc_rsrc_hndl_rsp_t;
struct mc_rsrc_hndl_rsp {
     mc_errnum_t                        mc_error;
     ct_resource_handle_t               mc_rsrc_hndl;
};
```

The fields of this structure contain the following:

mc_error
: If 0 (zero), this field indicates that the command was successful. Any other value is an error. If there is an error, the error codes indicate whether the resource handle contained in *mc_rsrc_hndl* is invalid or the command could not be completed for the resource specified by the resource handle. The error may also indicate that the command arguments were in error.

mc_rsrc_hndl
: The resource handle that identifies the resource that was the target of the command.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_free_response" on page 75
This subroutine frees a response or event notification structure.

"mc_online_*"
This subroutine requests the RMC subsystem to bring a resource online.

"mc_reset_*" on page 184
This subroutine requests the RMC subsystem to force a resource offline.

"mc_qdef_sd_*" on page 120
This subroutine queries the RMC subsystem to obtain the definition of structured data.

mc_online_*

This subroutine requests the RMC subsystem to bring a resource online.

Purpose

Requests the RMC subsystem to bring a resource online.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_online_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_online_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_online_bp(
          mc_sess_hndl_t             sess_hndl,
          mc_rsrc_hndl_rsp_t         **response,
          ct_resource_handle_t       rsrc_hndl,
          ct_char_t                  **node_names,
          ct_uint32_t                name_count,
          ct_structured_data_t       *data)
```

- The **mc_online_ap** subroutine adds the command to a command group. If used in an ordered command group, however, please note that all other commands in the command group must also be the form that uses a resource handle to specify a command target. To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_online_ap(
          mc_cmdgrp_hndl_t           cmdgrp_hndl,
          mc_rsrc_hndl_rsp_t         **response,
          ct_resource_handle_t       rsrc_hndl,
          ct_char_t                  **node_names,
          ct_uint32_t                name_count,
          ct_structured_data_t       *data)
```

- The **mc_online_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_online_bc(
          mc_sess_hndl_t             sess_hndl,
          mc_online_cb_t             *online_cb,
          void                       *online_cb_arg,
          ct_resource_handle_t       rsrc_hndl,
          ct_char_t                  **node_names,
          ct_uint32_t                name_count,
          ct_structured_data_t       *data)
```

  The definition for the response callback is:

```
typedef void (mc_online_cb_t)(mc_sess_hndl_t,
mc_rsrc_hndl_rsp_t *,
void *);
```

- The **mc_online_ac** subroutine adds the command to a command group. If used in an ordered command group, however, please note that all other commands in the command group must also be the form that uses a resource handle to specify a command target. To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_online_ac(
          mc_cmdgrp_hndl_t           cmdgrp_hndl,
          mc_online_cb_t             *online_cb,
          void                       *online_cb_arg,
          ct_resource_handle_t       rsrc_hndl,
          ct_char_t                  **node_names,
          ct_uint32_t                name_count,
          ct_structured_data_t       *data)
```

  The definition for the response callback is:

```
typedef void (mc_online_cb_t)(mc_sess_hndl_t,
mc_rsrc_hndl_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

online_cb
Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

online_cb_arg
Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

rsrc_hndl
The resource handle that identifies the resource to be brought online. A resource handle is returned in the response structure for many RMC API subroutines including the **mc_define_resource_*** subroutine. An array of resource handles for resources of a particular resource class is returned in the response structure for the **mc_enumerate_resources_*** and **mc_enumerate_permitted_rsrcs_*** subroutines. To validate the resource handle before calling this subroutine, the application can call one of the **mc_validate_rsrc_hndl_*** subroutines.

node_names
Specifies the nodes on which the resource should be brought online. Specifies this using a pointer to an array of *name_count* elements of type **ct_char_t**. Each element in the array specifies a node name.

name_count
Specifies the number of elements in the *node_names* array.

data A pointer to structured data containing resource-class specific options for bringing a resource online. To accept the default values (or if a resource class does not define options) for going online, the data parameter should be a NULL pointer.

To obtain the syntax and semantics for the structured data required by this resource class for specifying online options, the application can use the **mc_qdef_sd_*** subroutines.

OUTPUT

response
A pointer to a location in which the RMC API will return a pointer to the response. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_online_*** subroutines can be used by the application to request the RMC subsystem to bring a resource (identified by the *rsrc_hndl* parameter) online on a set of nodes (identified by the *node_names* parameter). If the resource accepts structured data as options for bringing a resource online, the application can provide this using the *data* parameter.

The response for these subroutines is a structure of type **mc_rsrc_hndl_rsp_t**, and is described in Response structure.

If this command is used in an ordered command group, please note that all other commands in the command group must be the form that uses a resource handle to specify the command target. The same resource handle must be used on all commands in the command group.

Security

To bring a resource online, the user of the calling application must have either **o** or **w** permission specified in an ACL entry for this resource.

Return values

For the **mc_online_bp** and **mc_online_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and a response has been received and processed.

For the **mc_online_ap** and **mc_online_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_ETARGETMISMATCH
The target specified for the command does not match the target of the command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before the response could be received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before the response could be received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_EINVALIDDATATYPE
Invalid attribute data type specified.

MC_EINVALIDVALUEPTR
Invalid attribute value pointer specified.

MC_EINVALIDSDTYPE
Invalid structured data subtype specified.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before the response could be received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_rsrc_hndl_rsp_t**. Although the **mc_error** field of the **mc_rsrc_hndl_rsp_t** structure indicates whether or not the resource manager has successfully processed the command, it does not mean that the resource is online. To determine if the resource is actually online, the application must register an event to monitor the resource's OpState attribute value. This command results in only one response.

The response structure definition is:

```
typedef struct mc_rsrc_hndl_rsp         mc_rsrc_hndl_rsp_t;
struct mc_rsrc_hndl_rsp {
     mc_errnum_t                        mc_error;
     ct_resource_handle_t               mc_rsrc_hndl;
};
```

The fields of this structure contain the following:

mc_error
If 0 (zero), this field indicates that the command was successful. Any other value is an error. If there is an error, the error codes indicate whether the resource handle contained in *mc_rsrc_hndl* is invalid or the command could not be completed for the resource specified by the resource handle. The error may also indicate that the command arguments were in error.

mc_rsrc_hndl
The resource handle that identifies the resource that was the target of the command.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_offline_*" on page 88
This subroutine sends a request to the RMC subsystem to take a resource offline.
"mc_qdef_sd_*" on page 120
This subroutine queries the RMC subsystem to obtain the definition of structured data.

mc_qdef_actions_*

This subroutine queries the RMC subsystem to obtain the definitions of resource class actions.

Purpose

Queries the RMC subsystem to obtain the definitions of resource class actions.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_qdef_actions_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_qdef_actions_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_qdef_actions_bp(
          mc_sess_hndl_t            sess_hndl,
          mc_qdef_actions_rsp_t     **rsp_array,
          ct_uint32_t               *array_cnt,
          mc_qdef_opts_t            options,
          ct_char_t                 *rsrc_class_name,
          ct_uint32_t               query_class_actions,
          ct_char_t                 **names,
          ct_uint32_t               count)
```

- The **mc_qdef_actions_ap** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_qdef_actions_ap(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_qdef_actions_rsp_t     **rsp_array,
          ct_uint32_t               *array_cnt,
          mc_qdef_opts_t            options,
          ct_char_t                 *rsrc_class_name,
          ct_uint32_t               query_class_actions,
          ct_char_t                 **names,
          ct_uint32_t               count)
```

- The **mc_qdef_actions_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_qdef_actions_bc(
          mc_sess_hndl_t            sess_hndl,
          mc_qdef_actions_cb_t      *qdef_cb,
          void                      *qdef_cb_arg,
```

```
        mc_qdef_opts_t          options,
        ct_char_t               *rsrc_class_name,
        ct_uint32_t             query_class_actions,
        ct_char_t               **names,
        ct_uint32_t             count)
```

The definition for the response callback is:

```
typedef void (mc_qdef_actions_cb_t)(mc_sess_hndl_t,
mc_qdef_actions_rsp_t *,
void *);
```

- The **mc_qdef_actions_ac** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
    mc_qdef_actions_ac(
        mc_cmdgrp_hndl_t        cmdgrp_hndl,
        mc_qdef_actions_cb_t    *qdef_cb,
        void                    *qdef_cb_arg,
        mc_qdef_opts_t          options,
        ct_char_t               *rsrc_class_name,
        ct_uint32_t             query_class_actions,
        ct_char_t               **names,
        ct_uint32_t             count)
```

The definition for the response callback is:

```
typedef void (mc_qdef_actions_cb_t)(mc_sess_hndl_t,
mc_qdef_actions_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
: The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
: The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

qdef_cb Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

qdef_cb_arg
: Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

options Specifies one of the following:

MC_QDEF_OPTS_NODSCRP
: No descriptions or display names are returned in the response structure.

MC_QDEF_OPTS_NONE
: There are no options.

rsrc_class_name
Pointer to a resource class name. Identifies the resource class whose resource class action definitions or resource action definitions are to be returned.

query_class_actions
Indicates whether information on actions that can be invoked on the resource class itself should be returned. If this parameter is 0 (zero), then the subroutine returns information on the actions that can be invoked on resources of the resource class. If this parameter is any non-zero value, then the subroutine returns information on actions that can be invoked on the resource class itself.

names Specifies the action(s) whose information you want returned. Specifies this using a pointer to an array of *count* elements of type **ct_char_t.** Each element in the array identifies an action name. If the *count* parameter is 0, then the *names* parameter must be a NULL pointer. In this case, information is returned for all actions that can be invoked on either the resource class or resources of the resource class (as indicated by the *query_class_actions* parameter.

count Specifies the number of elements in the *names* array. If 0, the *names* parameter must be a NULL pointer.

OUTPUT

rsp_array
A pointer to a location in which the RMC API will return a pointer to a response array of *array_cnt* elements. This parameter applies only to variations of this subroutine that use the pointer response method.

array_cnt
Identifies a location in which the RMC API will return the number of elements in the response array *rsp_array*. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_qdef_actions_*** subroutines can be used by the application to obtain the definitions of the actions that can be invoked on either a resource of a resource class, or on the resource class itself (as indicated by the *query_class_actions* parameter).

The response for these subroutines is a structure of type **mc_qdef_actions_rsp_t**, and is described in Response structure.

This command cannot be used in an ordered command group.

Return values

For the **mc_qdef_actions_bp** and **mc_qdef_actions_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and one response has been received and processed.

For the **mc_qdef_actions_ap** and **mc_qdef_actions_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_EORDERGROUP
An attempt was made to add the command to an ordered command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_EINVALIDCNTPTR
Invalid response count pointer specified.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before all responses could be received. If subroutine uses pointer response, check the appropriate array count for any responses. If the subroutine uses callback response, the callback will have been invoked for any responses received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_qdef_actions_rsp_t**. The response contains the requested information for one or more actions that can be invoked on a resource of the resource class or on the resource class itself. The amount of information can vary depending on the *options* and *names* parameters.

The response structure definition is:

```
typedef struct mc_qdef_actions_rsp      mc_qdef_actions_rsp_t;
struct mc_qdef_actions_rsp {
     mc_errnum_t                        mc_error;
     mc_action_t                        *mc_actions;
     ct_uint32_t                        mc_action_count;
     rmc_resource_class_id_t            mc_class_id;
};
```

The fields of this structure contain the following:

mc_error
: If 0 (zero), this field indicates that the query was successful. Any other value is an error and indicates that the resource monitoring and control (RMC) subsystem could not provide some or all of the requested information. The error may also indicate that the command arguments were in error. The error codes imply which of the remaining fields in the structure are defined.

mc_actions
: A pointer to an array of *mc_action_count* elements of type **mc_action_t**;

```
typedef struct mc_action            mc_action_t;
struct mc_action {
     mc_action_props_t              mc_properties;
     ct_char_t                      *mc_action_name;
     ct_char_t                      *mc_display_name;
     ct_char_t                      *mc_description;
     ct_char_t                      *mc_confirm_prompt;
     rmc_action_id_t                mc_action_id;
     mc_variety_t                   *mc_variety_list;
     ct_uint32_t                    mc_variety_count;
     ct_uint32_t                    mc_timeout;
     ct_uint32_t                    mc_permissions;
};
```

The fields of this structure contain the following:

mc_properties
: A bit field that specifies the properties of the action, as defined by the **mc_action_props_t** enumeration.

```
typedef enum mc_action_props mc_action_props_t;
enum mc_action_props {
     MC_RSRC_ACTION_LONG_RUNNING = 0x0001,
     MC_RSRC_ACTION_PUBLIC = 0x0002
};
```

mc_action_name
: A pointer to the programmatic name of the action (this name is provided as input to other RMC API subroutines).

mc_display_name
: A pointer to a string that is suitable to display as the name of this action in a Graphical User Interface.

mc_description
: A pointer to a string that contains a description of the action. The description may contain multiple lines of text.

mc_confirm_prompt
A pointer to a string that is used by a GUI to prompt the user for confirmation to perform the action.

mc_action_id
The action ID.

mc_variety_list
A pointer to an array of *mc_variety_count* elements of type **mc_variety_t** . Each element of the array is a range of resource variety numbers. This action then applies to any resource of the class (or the resource class itself if this response is that of a resource class action) that has one of the indicated variety numbers.

mc_variety_count
The number of elements in the *mc_variety_list* array.

mc_timeout
The approximate amount of time, in seconds, that the action should take to complete. This value is defined only if the **MC_RSRC_ACTION_LONG_RUNNING** property is set.

mc_permissions
The permissions required to execute this action.

mc_action_count
Indicates the number of entries in the *mc_actions* array.

mc_class_id
The ID of the resource class for which these actions are being returned.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_free_response" on page 75
This subroutine frees a response or event notification structure.

"mc_invoke_class_action_*" on page 82
This subroutine invokes an action on a resource class.

mc_qdef_d_attribute_*

This subroutine queries the RMC subsystem to obtain dynamic attribute definitions for a resource or resource class.

Purpose

Queries the RMC subsystem to obtain dynamic attribute definitions for a resource or resource class.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_qdef_d_attribute_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_qdef_d_attribute_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_qdef_d_attribute_bp(
          mc_sess_hndl_t            sess_hndl,
          mc_qdef_dattr_rsp_t       **rsp_array,
          ct_uint32_t               *array_cnt,
          mc_qdef_opts_t            options,
          ct_char_t                 *rsrc_class_name,
          ct_uint32_t               query_class_attrs,
          ct_char_t                 **attr_names,
          ct_uint32_t               attr_count)
```

- The **mc_qdef_d_attribute_ap** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_qdef_d_attribute_ap(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_qdef_dattr_rsp_t       **rsp_array,
          ct_uint32_t               *array_cnt,
          mc_qdef_opts_t            options,
          ct_char_t                 *rsrc_class_name,
          ct_uint32_t               query_class_attrs,
          ct_char_t                 **attr_names,
          ct_uint32_t               attr_count)
```

- The **mc_qdef_d_attribute_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_qdef_d_attribute_bc(
          mc_sess_hndl_t            sess_hndl,
          mc_qdef_dattr_cb_t        *qdef_cb,
          void                      *qdef_cb_arg,
          mc_qdef_opts_t            options,
          ct_char_t                 *rsrc_class_name,
          ct_uint32_t               query_class_attrs,
          ct_char_t                 **attr_names,
          ct_uint32_t               attr_count)
```

  The definition for the response callback is:

```
typedef void (mc_qdef_dattr_cb_t)(mc_sess_hndl_t,
mc_qdef_dattr_rsp_t *,
void *);
```

- The **mc_qdef_d_attribute_ac** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_qdef_d_attribute_ac(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_qdef_dattr_cb_t        *qdef_cb,
          void                      *qdef_cb_arg,
          mc_qdef_opts_t            options,
          ct_char_t                 *rsrc_class_name,
          ct_uint32_t               query_class_attrs,
          ct_char_t                 **attr_names,
          ct_uint32_t               attr_count)
```

  The definition for the response callback is:

```
typedef void (mc_qdef_dattr_cb_t)(mc_sess_hndl_t,
mc_qdef_dattr_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

qdef_cb Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

qdef_cb_arg
Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

options Specifies one of the following:

MC_QDEF_OPTS_NODSCRP
No descriptions or display names are returned in the response structure.

MC_QDEF_OPTS_NONE
There are no options.

rsrc_class_name
Pointer to a resource class name. Identifies the resource class whose dynamic class attribute definitions or resource dynamic attribute definitions are to be returned.

query_class_attrs
Indicates whether the subroutine should return the dynamic attribute definitions for the resource class itself. If this parameter is 0 (zero), then the subroutine returns dynamic attribute definitions for resources of the resource class. If this parameter is any non-zero value, then the subroutine returns dynamic attribute definitions for the resource class itself.

attr_names
Specifies the dynamic attribute(s) whose definition(s) you want returned. Specifies this using an array of *attr_count* elements of type **ct_char_t**. Each element in the array identifies a dynamic attribute.

If the *attr_count* parameter is 0, then the *attr_names* parameter must be a NULL pointer. In this case, information is returned for all dynamic attributes of either the resource class or resources of the resource class (as indicated by the *query_class_attrs* parameter).

attr_count
Specifies the number of elements in the *attr_names* array. If 0, the *attr_names* parameter must be a NULL pointer.

OUTPUT

rsp_array
A pointer to a location in which the RMC API will return a pointer to a response array of *array_cnt* elements. This parameter applies only to variations of this subroutine that use the pointer response method.

array_cnt
Identifies a location in which the RMC API will return the number of elements in the response array *rsp_array*. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_qdef_d_attribute** subroutines can be used by the application to obtain dynamic attribute definitions for a resource or resource class (as indicated by the *query_class_attrs* parameter).

This command cannot be used in an ordered command group.

Return values

For the **mc_qdef_d_attribute_bp** and **mc_qdef_d_attribute_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and one or more responses have been received and processed.

For the **mc_qdef_d_attribute_ap** and **mc_qdef_d_attribute_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_EORDERGROUP
An attempt was made to add the command to an ordered command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_EINVALIDCNTPTR
Invalid response count pointer specified.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before all responses could be received. If subroutine uses pointer response, check the appropriate array count for any responses. If the subroutine uses callback response, the callback will have been invoked for any responses received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_qdef_dattr_rsp_t**. The response contains the requested information for one dynamic attribute of the specified resource or resource class. If more that one attribute is specified by the *attr_names* parameter, then there will be one response for each attribute. The amount of information will vary depending on the options and *attr_names* parameters. If any of the query arguments are invalid, then only one response is returned, indicating the error (even if more than one attribute was specified by the *attr_names* parameter).

The response structure definition is:

```
typedef struct mc_qdef_dattr_rsp        mc_qdef_dattr_rsp_t;
struct mc_qdef_dattr_rsp {
     mc_errnum_t                        mc_error;
     mc_dattr_props_t                   mc_properties;
     ct_char_t                          *mc_program_name;
     ct_char_t                          *mc_display_name;
     ct_char_t                          *mc_group_name;
     ct_char_t                          *mc_description;
     rmc_attribute_id_t                 mc_attribute_id;
     ct_uint32_t                        mc_group_id;
     ct_data_type_t                     mc_data_type;
     rmc_variable_type_t                mc_variable_type;
     mc_variety_t                       *mc_variety_list;
     ct_uint32_t                        mc_variety_count;
     ct_value_t                         mc_init_value;
     ct_value_t                         mc_min_value;
     ct_value_t                         mc_max_value;
     ct_char_t                          *mc_expression;
     ct_char_t                          *mc_expression_description;
```

```
    ct_char_t                       *mc_rearm_expression;
    ct_char_t                       *mc_rearm_description;
    ct_char_t                       *mc_PTX_name;
    ct_uint32_t                     mc_reporting_interval;
};
```

The fields of this structure contain the following:

mc_error
: If 0 (zero), this field indicates that the query was successful. Any other value is an error and indicates that the RMC subsystem could not provide some or all of the requested information. The error may also indicate that the command arguments were in error. The error codes imply which of the remaining fields in the structure are defined.

mc_properties
: A bit field that specifies the properties of the dynamic attribute, as defined by the **mc_dattr_props_t** enumeration.

```
typedef enum mc_dattr_props             mc_dattr_props_t;
enum mc_dattr_props {
    MC_RSRC_DATTR_PUBLIC                    = 0x0020
    MC_RSRC_DATTR_QRY_REQS_MONITORING       = 0x0040
};
```

  The **MC_RSRC_DATTR_QRY_REQS_MONITORING** property indicates that, in order to query the value of this dynamic attribute, the attribute must already be monitored as a result of some current event registration.

mc_program_name
: A pointer to the programmatic name of the attribute (this name is provided as input to other RMC API subroutines).

mc_display_name
: A pointer to a string that is suitable to display as the name of this attribute in a Graphical User Interface.

mc_group_name
: A pointer to a string containing the name of the group to which this attribute belongs. This name is designed to be suitable to display in a Graphical User Interface.

mc_description
: A pointer to a string that contains a description of the dynamic attribute. The description may contain multiple lines of text.

mc_attribute_id
: The attribute ID.

mc_group_id
: The group ID. This ID is used to group related attributes of a resource or the resource class.

mc_data_type
: The data type of the dynamic attribute.

mc_variable_type
: The variable type of the dynamic attribute. A dynamic attribute can have a variable type of Counter, Quantity, State or Quantum.

mc_variety_list
: A pointer to an array of **mc_variety_t** types. Each element of the array is a range of resource variety numbers. This attribute then applies to any resource of the class, or the resource class itself if this response is that of a resource class dynamic attribute, which has one of the indicated variety numbers.

mc_variety_count
The number of elements in the *mc_variety_list* array.

mc_init_value
The initial value that this variable assumes the first time it is monitored. Some event expressions compare the currently observed value of an attribute with the previously observed value. For those expressions, the initial value specified in this field will, upon first observation of this variable, be reference as the previously observed value.

mc_min_value
The lowest value in a dynamic range for the variable that should be displayed in a GUI. The highest value in this dynamic range is specified in the *mc_max_value* field. This dynamic range is defined only if the attribute type is an arithmetic type.

mc_max_value
The highest value in a dynamic range for the variable that should be displayed in a GUI. The lowest value in this dynamic range is specified in the *mc_min_value* field. This dynamic range is defined only if the attribute type is an arithmetic type.

mc_expression
A NULL pointer or a pointer to a string that is an example expression for this variable.

mc_expression_description
A NULL pointer or a pointer to a string that is a description of the event generated by the example expression.

mc_rearm_expression
A NULL pointer or a pointer to a string that is an example rearm expression for this variable.

mc_rearm_description
A NULL pointer or a pointer to a string that is a description of the example rearm expression.

mc_PTX_name
A NULL pointer or a pointer or a pointer to a string that is the PTX path name for this variable.

mc_reporting_interval
If the attribute is of variable type **Counter** or **Quantity** and is being monitored, a new attribute value is reported every *mc_reporting_interval* seconds.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_class_query_d_*" on page 42
This subroutine queries the RMC subsystem to obtain the dynamic attribute values of a resource class.

"mc_query_d_handle_*" on page 133
This subroutine queries the RMC subsystem to obtain the dynamic attribute values of a resource.

"mc_qdef_p_attribute_*"
This subroutine queries the RMC subsystem to obtain the persistent attribute definitions for a resource or resource class.

mc_qdef_p_attribute_*

This subroutine queries the RMC subsystem to obtain the persistent attribute definitions for a resource or resource class.

Purpose

Queries the RMC subsystem to obtain the persistent attribute definitions for a resource or resource class.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_qdef_p_attribute_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_qdef_p_attribute_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_qdef_p_attribute_bp(
          mc_sess_hndl_t            sess_hndl,
          mc_qdef_pattr_rsp_t       **rsp_array,
          ct_uint32_t               *array_cnt,
          mc_qdef_opts_t            options,
          ct_char_t                 *rsrc_class_name,
          ct_uint32_t               query_class_attrs,
          ct_char_t                 **attr_names,
          ct_uint32_t               attr_count)
```

- The **mc_qdef_p_attribute_ap** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_qdef_p_attribute_ap(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_qdef_pattr_rsp_t       **rsp_array,
          ct_uint32_t               *array_cnt,
          mc_qdef_opts_t            options,
          ct_char_t                 *rsrc_class_name,
          ct_uint32_t               query_class_attrs,
          ct_char_t                 **attr_names,
          ct_uint32_t               attr_count)
```

- The **mc_qdef_p_attribute_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_qdef_p_attribute_bc(
          mc_sess_hndl_t            sess_hndl,
          mc_qdef_pattr_cb_t        *qdef_cb,
          void                      *qdef_cb_arg,
          mc_qdef_opts_t            options,
          ct_char_t                 *rsrc_class_name,
          ct_uint32_t               query_class_attrs,
          ct_char_t                 **attr_names,
          ct_uint32_t               attr_count)
```

  The definition for the response callback is:

```
typedef void (mc_qdef_pattr_cb_t)(mc_sess_hndl_t,
mc_qdef_pattr_rsp_t *,
void *);
```

- The **mc_qdef_p_attribute_ac** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_qdef_p_attribute_ac(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_qdef_pattr_cb_t        *qdef_cb,
          void                      *qdef_cb_arg,
          mc_qdef_opts_t            options,
          ct_char_t                 *rsrc_class_name,
          ct_uint32_t               query_class_attrs,
          ct_char_t                 **attr_names,
          ct_uint32_t               attr_count)
```

The definition for the response callback is:

```
typedef void (mc_qdef_pattr_cb_t)(mc_sess_hndl_t,
mc_qdef_pattr_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
: The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
: The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

qdef_cb Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

qdef_cb_arg
: Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

options Specifies one of the following:

MC_QDEF_OPTS_NODSCRP
: No descriptor or display names are returned in the response structure.

MC_QDEF_OPTS_NONE
: There are no options.

rsrc_class_name
: Pointer to a resource class name. Identifies the resource class whose persistent class attribute definitions or resource persistent attribute definitions are to be returned.

query_class_attrs
: Indicates whether the subroutine should return the persistent attribute definitions for the resource class itself. If this parameter is 0 (zero), then the subroutine returns persistent attribute definitions for resources of the resource class. If this parameter is any non-zero value, then the subroutine returns persistent attribute definitions for the resource class itself.

attr_names
: Specifies the persistent attribute(s) whose definition(s) you want returned. Specifies this using an array of *attr_count* elements of type **ct_char_t**. Each element in the array identifies a persistent attribute.

If the *attr_count* parameter is 0, then the *attr_names* parameter must be a NULL pointer. In this case, information is returned for all persistent attributes of either the resource class or resources of the resource class (as indicated by the *query_class_attrs* parameter).

attr_count
Specifies the number of elements in the *attr_names* array. If 0, the *attr_names* parameter must be a NULL pointer.

OUTPUT

rsp_array
A pointer to a location in which the RMC API will return a pointer to a response array of *array_cnt* elements. This parameter applies only to variations of this subroutine that use the pointer response method.

array_cnt
Identifies a location in which the RMC API will return the number of elements in the response array *rsp_array*. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_qdef_p_attribute** subroutines can be used by the application to obtain persistent attribute definitions for a resource or resource class (as indicated by the *query_class_attrs* parameter).

The response for these subroutines is a structure of type **mc_qdef_pattr_rsp_t**, and is described in Response structure.

This command cannot be used in an ordered command group.

Return values

For the **mc_qdef_p_attribute_bp** and **mc_qdef_p_attribute_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and one or more responses have been received and processed.

For the **mc_qdef_p_attribute_ap** and **mc_qdef_p_attribute_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_EORDERGROUP
An attempt was made to add the command to an ordered command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_EINVALIDCNTPTR
Invalid response count pointer specified.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before all responses could be received. If subroutine uses pointer response, check the appropriate array count for any responses. If the subroutine uses callback response, the callback will have been invoked for any responses received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_qdef_pattr_rsp_t**. The response contains the requested information for one persistent attribute of the specified resource or resource class. If more that one attribute is specified by the *attr_names* parameter, then there will be one response for each attribute. The amount of information will vary depending on the options and *attr_names* parameters. If any of the query arguments are invalid, then only one response is returned, indicating the error (even if more than one attribute was specified by the *attr_names* parameter).

The response structure definition is:

```
typedef struct mc_qdef_pattr_rsp        mc_qdef_pattr_rsp_t;
struct mc_qdef_pattr_rsp {
     mc_errnum_t                        mc_error;
     mc_pattr_props_t                   mc_properties;
     ct_char_t                          *mc_program_name;
     ct_char_t                          *mc_display_name;
```

```
    ct_char_t                       *mc_group_name;
    ct_char_t                       *mc_description;
    rmc_attribute_id_t              mc_attribute_id;
    ct_uint32_t                     mc_group_id;
    ct_data_type_t                  mc_data_type;
    mc_variety_t                    *mc_variety_list;
    ct_uint32_t                     mc_variety_count;
    ct_value_t                      mc_default_value;
};
```

The fields of this structure contain the following:

mc_error
: If 0 (zero), this field indicates that the query was successful. Any other value is an error and indicates that the RMC subsystem could not provide some or all of the requested information. The error may also indicate that the command arguments were in error. The error codes imply which of the remaining fields in the structure are defined.

mc_properties
: A bit field that specifies the properties of the persistent attribute, as defined by the **mc_pattr_props_t** enumeration.

```
typedef enum mc_pattr_props       mc_pattr_props_t;
enum mc_pattr_props {
     MC_RSRC_PATTR_READ_ONLY             = 0x0001,
     MC_RSRC_PATTR_REQD_FOR_DEFINE       = 0x0002,
     MC_RSRC_PATTR_INVAL_FOR_DEFINE      = 0x0004,
     MC_RSRC_PATTR_OPTION_FOR_DEFINE     = 0x0008,
     MC_RSRC_PATTR_SELECTABLE            = 0x0010,
     MC_RSRC_PATTR_PUBLIC                = 0x0020
};
```

mc_program_name
: A pointer to the programmatic name of the attribute (this name is provided as input to other RMC API subroutines).

mc_display_name
: A pointer to a string that is suitable to display as the name of this attribute in a Graphical User Interface.

mc_group_name
: A pointer to a string containing the name of the group to which this attribute belongs. This name is designed to be suitable to display in a Graphical User Interface.

mc_description
: A pointer to a string that contains a description of the persistent attribute. The description may contain multiple lines of text.

mc_attribute_id
: The attribute ID.

mc_group_id
: The group ID. This ID is used to group related attributes of a resource or the resource class.

mc_data_type
: The data type of the persistent attribute.

mc_variety_list
: A pointer to an array of **mc_variety_t** types. Each element of the array is a range of resource variety numbers. This attribute then applies to any resource of the class (or the resource class itself if this response is that of a resource class persistent attribute) which has one of the indicated variety numbers.

mc_variety_count
: The number of elements in the *mc_variety_list* array.

mc_default_value
The default value this attribute assumes when a resource, for which this attribute is defined, is created.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_define_resource_*" on page 58
This subroutine defines a new resource.

"mc_free_response" on page 75
This subroutine frees a response or event notification structure.

"mc_qdef_d_attribute_*" on page 102
This subroutine queries the RMC subsystem to obtain dynamic attribute definitions for a resource or resource class.

"mc_set_select_*" on page 199
This subroutine sets persistent attribute values of one or more resources of a particular resource class. The resources are identified by attribute selection.

"mc_qdef_resource_class_*"
This subroutine queries the RMC subsystem to obtain the definition of a resource class or all definitions of all resource classes.

mc_qdef_resource_class_*

This subroutine queries the RMC subsystem to obtain the definition of a resource class or all definitions of all resource classes.

Purpose

Queries the RMC subsystem to obtain the definition of a resource class or all definitions of all resource classes.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_qdef_resource_class_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_qdef_resource_class_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_qdef_resource_class_bp(
          mc_sess_hndl_t            sess_hndl,
          mc_qdef_rsrc_class_rsp_t  **rsp_array,
          ct_uint32_t               *array_cnt,
          mc_qdef_opts_t            options,
          ct_char_t                 *rsrc_class_name)
```

- The **mc_qdef_resource_class_ap** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_qdef_resource_class_ap(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_qdef_rsrc_class_rsp_t  **rsp_array,
          ct_uint32_t               *array_cnt,
          mc_qdef_opts_t            options,
          ct_char_t                 *rsrc_class_name)
```

- The **mc_qdef_resource_class_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_qdef_resource_class_bc(
          mc_sess_hndl_t            sess_hndl,
          mc_qdef_rsrc_class_cb_t   *qdef_cb,
          void                      *qdef_cb_arg,
          mc_qdef_opts_t            options,
          ct_char_t                 *rsrc_class_name)
```

  The definition for the response callback is:

```
typedef void (mc_qdef_rsrc_class_cb_t)(mc_sess_hndl_t,
mc_qdef_rsrc_class_rsp_t *,
void *);
```

- The **mc_qdef_resource_class_ac** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_qdef_resource_class_ac(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_qdef_rsrc_class_cb_t   *qdef_cb,
          void                      *qdef_cb_arg,
          mc_qdef_opts_t            options,
          ct_char_t                 *rsrc_class_name)
```

  The definition for the response callback is:

```
typedef void (mc_qdef_rsrc_class_cb_t)(mc_sess_hndl_t,
mc_qdef_rsrc_class_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
: The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
: The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

qdef_cb Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

qdef_cb_arg
Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

options Specifies one of the following:

MC_QDEF_OPTS_NODSCRP
No description or display names are retuned in the response structure.

MC_QDEF_OPTS_NONE
There are no options.

rsrc_class_name
Pointer to a resource class name identifying the resource class whose definition is to be returned. If this parameter is a Null pointer, then definitions are returned for all resource classes.

OUTPUT

rsp_array
A pointer to a location in which the RMC API will return a pointer to a response array of *array_cnt* elements. This parameter applies only to variations of this subroutine that use the pointer response method.

array_cnt
Identifies a location in which the RMC API will return the number of elements in the response array *rsp_array*. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_qdef_resource_class_*** subroutines can be used by the application to obtain the definition of one resource class or the definitions for all resource classes (as specified by the *rsrc_class_name* parameter).

The response for these subroutines is a structure of type **mc_qdef_rsrc_class_rsp_t**, and is described in Response structure.

This command cannot be used in an ordered command group.

Return values

For the **mc_qdef_resource_class_bp** and **mc_qdef_resource_class_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and one or more responses have been received and processed.

For the **mc_qdef_resource_class_ap** and **mc_qdef_resource_class_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_EORDERGROUP
An attempt was made to add the command to an ordered command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_EINVALIDCNTPTR
Invalid response count pointer specified.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before all responses could be received. If subroutine uses pointer response, check the appropriate array count for any responses. If the subroutine uses callback response, the callback will have been invoked for any responses received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_qdef_rsrc_class_rsp_t**. The response contains the requested information for one resource class. If the *rsrc_class_name* parameter is a Null pointer (which indicates that definitions for all resource classes should be returned), then there will be one response for each resource class. The amount of information will vary depending on the *options* parameter. If any of the query arguments are not valid, then only one response is returned, indicating the error.

The response structure definition is:

```
typedef struct mc_qdef_rsrc_class_rsp   mc_qdef_rsrc_class_rsp_t;
struct mc_qdef_rsrc_class_rsp {
     mc_errnum_t                        mc_error;
     mc_rsrc_class_props_t              mc_properties;
     ct_char_t                          *mc_class_name;
     rmc_resource_class_id_t            mc_class_id;
     ct_char_t                          *mc_display_name;
     ct_char_t                          *mc_description;
     ct_char_t                          *mc_locator;
     ct_uint32_t                        mc_class_pattr_count;
     ct_uint32_t                        mc_class_dattr_count;
     ct_uint32_t                        mc_pattr_count;
     ct_uint32_t                        mc_dattr_count;
     ct_uint32_t                        mc_action_count;
     ct_uint32_t                        mc_class_action_count;
     ct_uint32_t                        mc_error_count;
     mc_rsrc_mgr_t                      *mc_rsrc_mgrs;
     ct_uint32_t                        mc_rsrc_mgr_count;
};
```

The fields of this structure contain the following:

mc_error
: If 0 (zero), this field indicates that the query was successful. Any other value is an error and indicates that the resource monitoring and control (RMC) subsystem could not provide some or all of the requested information. The error may also indicate that the command arguments were in error. The error codes imply which of the remaining fields in the structure are defined.

mc_properties
: A bit field that specifies the properties of the resource class, as defined by the **mc_rsrc_class_props_t** enumeration.

```
typedef enum mc_rsrc_class_props    mc_rsrc_class_props_t;
enum mc_rsrc_class_props {
     MC_RSRC_CLASS_HAS_IW_ACCESS        = 0x0001,
     MC_RSRC_CLASS_HAS_RSRC_INSTS       = 0x0002,
     MC_RSRC_CLASS_HAS_CTRL_INTERFACE   = 0x0004,
     MC_RSRC_CLASS_CAN_DEFINE_UNDEFINE  = 0x0008,
     MC_RSRC_CLASS_SUPPORTS_MOVE        = 0x0200,
     MC_RSRC_CLASS_CAN_BATCH_DEFINE     = 0x0400,
     MC_RSRC_CLASS_CAN_BATCH_UNDEFINE   = 0x0800,
     MC_RSRC_CLASS_CAN_BATCH_SET_ATTR   = 0x1000,
     MC_RSRC_CLASS_MTYPE_SUBDIVIDED     = 0x2000,
     MC_RSRC_CLASS_MTYPE_GLOBALIZED     = 0x8000,
     MC_RSRC_CLASS_ACT_QUORUM_CHANGE    = 0x00010000,
     MC_RSRC_CLASS_QRY_REQS_MONITORING  = 0x00020000
};
```

mc_class_name
: Specifies the name of the resource class that was queried and whose definition is contained in this response.

mc_class_id
: Specifies the resource class ID of the resource class that was queried and whose definition is contained in this response.

mc_display_name
: A pointer to a string that is suitable to display as the name of this resource class in a graphical user interface.

mc_description
: A pointer to a string that contains a description of the resource class; the description may contains multiple lines of text.

mc_locator
The name of a persistent attribute of a resource of this resource class that implies the location of the resource.

mc_class_pattr_count
The number of persistent attributes defined for the resource class itself.

mc_class_dattr_count
The number of dynamic attributes defined for the resource class itself.

mc_pattr_count
The number of persistent attributes defined for a resource of the resource class.

mc_dattr_count
The number of dynamic attributes defined for a resource of the resource class.

mc_action_count
The number of different actions that can be invoked against the resources of the resource class.

mc_class_action_count
The number of different actions that can be invoked against the resource class itself.

mc_error_count
The number of different errors that can be injected into the resources of the resource class.

mc_rsrc_mgrs
A pointer to an array of **mc_rsrc_mgr_t** types. Each element of this array specifies a resource manager that implements the resource class.

```
typedef struct mc_rsrc_mgr          mc_rsrc_mgr_t;
struct mc_rsrc_mgr {
     ct_char_t      *mc_mgr_name;
     ct_uint32_t     mc_first_key;
     ct_uint32_t     mc_last_key;
};
```

The fields of this structure contain the following:

mc_mgr_name
The name of the resource manager that supports the resource class

mc_first_key
The start of the range of resource manager ClassKeys that implement the resource class.

mc_last_key
The end of the range of resource manager ClassKeys that implement the resource class.

Note that there may be multiple entries specifying the same resource manager in the *mc_rsrc_mgrs* array. If there are multiple entries, however, each entry specifies non-overlapping ranges of ClassKeys.

mc_rsrc_mgr_count
The number of elements in the *mc_rsrc_mgrs* array.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_qdef_p_attribute_*" on page 108
This subroutine queries the RMC subsystem to obtain the persistent attribute definitions for a resource or resource class.

mc_qdef_sd_*

This subroutine queries the RMC subsystem to obtain the definition of structured data.

Purpose

Queries the RMC subsystem to obtain the definition of structured data.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_qdef_sd_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_qdef_sd_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_qdef_sd_bp(
          mc_sess_hndl_t            sess_hndl,
          mc_qdef_sd_rsp_t          **rsp_array,
          ct_uint32_t               *array_cnt,
          mc_qdef_opts_t            options,
          ct_char_t                 *rsrc_class_name,
          mc_sd_usage_t             sd_use,
          ct_char_t                 **names,
          ct_uint32_t               count)
```

- The **mc_qdef_sd_ap** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_qdef_sd_ap(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_qdef_sd_rsp_t          **rsp_array,
          ct_uint32_t               *array_cnt,
          mc_qdef_opts_t            options,
          ct_char_t                 *rsrc_class_name,
          mc_sd_usage_t             sd_use,
          ct_char_t                 **names,
          ct_uint32_t               count)
```

- The **mc_qdef_sd_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_qdef_sd_bc(
          mc_sess_hndl_t            sess_hndl,
          mc_qdef_sd_cb_t           *qdef_cb,
          void                      *qdef_cb_arg,
          mc_qdef_opts_t            options,
          ct_char_t                 *rsrc_class_name,
          mc_sd_usage_t             sd_use,
          ct_char_t                 **names,
          ct_uint32_t               count)
```

  The definition for the response callback is:

```
typedef void (mc_qdef_sd_cb_t)(mc_sess_hndl_t,
mc_qdef_sd_rsp_t *,
void *);
```

- The **mc_qdef_sd_ac** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_qdef_sd_ac(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_qdef_sd_cb_t           *qdef_cb,
          void                      *qdef_cb_arg,
          mc_qdef_opts_t            options,
          ct_char_t                 *rsrc_class_name,
          mc_sd_usage_t             sd_use,
          ct_char_t                 **names,
          ct_uint32_t               count)
```

  The definition for the response callback is:

```
typedef void (mc_qdef_sd_cb_t)(mc_sess_hndl_t,
mc_qdef_sd_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
: The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
: The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

qdef_cb Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

qdef_cb_arg
: Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

options Specifies one of the following:

MC_QDEF_OPTS_NODSCRP
: No descriptions or display names are returned in the response structure.

MC_QDEF_OPTS_NONE
: There are no options.

rsrc_class_name
: Pointer to a resource class name. Identifies the resource class for which structured data (SD) information is to be returned.

sd_use Specifies, using one of the following values, the structured data information to be returned.

MC_SD_USAGE_PATTR_RESOURCE
: SD persistent attributes of the resource of the specified resource class

MC_SD_USAGE_PATTR_RSRC_CLASS
SD persistent attributes of the specified resource class

MC_SD_USAGE_DATTR_RESOURCE
SD dynamic attributes of the resource of the specified resource class

MC_SD_USAGE_DATTR_RSRC_CLASS
SD dynamic attributes of the specified resource class

MC_SD_USAGE_RSRC_ACTION_INPUT
SD input formats for the resource actions of the specified resource class

MC_SD_USAGE_RSRC_ACTION_RESPONSE
SD response formats for the resource actions of the specified resource class

MC_SD_USAGE_CLASS_ACTION_INPUT
SD input formats for the class actions of the specified resource class

MC_SD_USAGE_CLASS_ACTION_RESPONSE
SD response formats for the class actions of the specified resource class

MC_SD_USAGE_DEFINE_ARG
SD format for the define resource command argument of the specified resource class

MC_SD_USAGE_UNDEFINE_ARG
SD format for the undefine resource command argument of the specified resource class

MC_SD_USAGE_ONLINE_ARG
SD format for the online command argument of the specified resource class

MC_SD_USAGE_OFFLINE_ARG
SD format for the offline command argument of the specified resource class

MC_SD_USAGE_RESET_ARG
SD format for the reset command argument of the specified resource class

names

Table 20. mc_qdef_sd_* subroutine conditional names parameter functions

If the *sd_use* parameter specifies that structured data (SD) should be returned for:	**Then:**
attributes or actions	This parameter specifies the attributes or actions for which information will be retuned. This parameter specifies this using an array of *count* elements of type **ct_char_t**. Each element in the array identifies an SD type attribute name or an action name as appropriate. If the *count* parameter is 0, then the *names* parameter must be a NULL pointer. In this case, structured data information is returned for all SD attributes or actions of the resource class.
command arguments	This parameter must be NULL and the count parameter must be 0.

count Specifies the number of elements in the *names* array. If 0, the *names* array must be a NULL pointer.

OUTPUT

rsp_array
A pointer to a location in which the RMC API will return a pointer to a response array of *array_cnt* elements. This parameter applies only to variations of this subroutine that use the pointer response method.

array_cnt
Identifies a location in which the RMC API will return the number of elements in the response array *rsp_array*. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_qdef_sd_*** subroutines can be used by the application to obtain the definition of structured data (SD). Structured data definitions will be returned for the type of structured data specified by the *sd_use* parameter for the resource class identified by the *rsrc_class_name* parameter. If the *sd_use* parameter specifies that structured data should be returned for SD attributes or actions, then the *names* parameter identifies the SD attributes or actions for which information will be returned.

Structured data:

- can be a data type of attributes
- is used to pass input, if necessary, to an action when calling the **mc_invoke_action_*** or **mc_invoke_class_action_*** subroutines.
- is the type of response data resulting from calls to the **mc_invoke_action_*** or **mc_invoke_class_action_*** subroutines.
- is the type of an optional argument for passing resource class-specific options to the **mc_define_resource_***, **mc_undefine_resource_***, **mc_online_***, **mc_offline_***, and **mc_reset_*** subroutines.

The structured data definitions include the

- Structured data element program name
- Element display name
- Element description
- Data type
- Structured data element index value

The response for these subroutines is a structure of type **mc_qdef_sd_rsp_t**, and is described in Response structure.

This command cannot be used in an ordered command group.

Return values

For the **mc_qdef_sd_bp** and **mc_qdef_sd_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and one or more responses have been received and processed.

For the **mc_qdef_sd_ap** and **mc_qdef_sd_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_EORDERGROUP
An attempt was made to add the command to an ordered command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_EINVALIDCNTPTR
Invalid response count pointer specified.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before all responses could be received. If subroutine uses pointer response, check the appropriate array count for any responses. If the subroutine uses callback response, the callback will have been invoked for any responses received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_qdef_sd_rsp_t**. The response contains the requested structured data information for one of the following (depending on the value specified by the *sd_use* parameter).

- one structured data type attribute

- one action input format
- one action response format
- one command option

If more than one attribute or action name is specified by the names parameter, then there is a response for each. The amount of information in any response can vary depending on the value specified by the *options* parameter. The *mc_error* field of the **mc_qdef_sd_rsp_t** structure indicates whether or not the query was successful. If any of the query arguments are invalid, then only one response is returned, indicating the error.

The response structure definition is:

```
typedef struct mc_qdef_sd_rsp              mc_qdef_sd_rsp_t;
struct mc_qdef_sd_rsp {
     mc_errnum_t                           mc_error;
     mc_sd_element_t                       *mc_sd_elements;
     ct_uint32_t                           mc_element_count;
     ct_uint32_t                           mc_id;
     ct_char_t                             *mc_program_name;
     mc_sd_usage_t                         mc_usage;
};
```

The fields of this structure contain the following:

mc_error
: If 0 (zero), this field indicates that the query was successful. Any other value is an error and indicates that the RMC subsystem could not provide some or all of the requested information. The error may also indicate that the command arguments were in error. The error codes imply which of the remaining fields in the structure are defined.

mc_sd_elements
: A pointer to an array of *mc_element_count* elements of type **mc_sd_element_t**;

```
typedef struct mc_sd_element          mc_sd_element_t;
struct mc_sd_element {
     ct_char_t                        *mc_element_name;
     ct_char_t                        *mc_display_name;
     ct_char_t                        *mc_description;
     ct_data_type_t                   mc_element_data_type;
     ct_uint32_t                      mc_element_index;
};
```

The fields of this structure contain the following:

mc_element_name
: A pointer to the programmatic name of the structured data (SD) element (this name is provided as input to other RMC API functions).

mc_display_name
: A pointer to a string that is suitable to display as the name of this element in a graphical user interface.

mc_description
: A pointer to a string that contains a description of the SD element. The description may contain multiple lines of text.

mc_element_data_type
: The data type of the SD element.

mc_element_index
: The index of this SD element. This number can be used as an index into the array specified by the *elements* field of a **ct_structured_data_t** type.

mc_element_count
Specifies the number of elements in the *mc_sd_elements* array.

mc_id If SD information for an attribute is being returned, this field is the ID of the attribute. If SD information is being returned for an action, this field is the ID of the action. If SD information for a command argument is being returned, this field is undefined. The type of SD information being returned is indicated in the *mc_usage* field.

mc_program_name
If SD information for an attribute is being returned, this field is a pointer to the programmatic name of the attribute. If SD information is being returned for an action, this field is a pointer to the programmatic name of the action. If SD information for a command argument is being returned, this field is undefined. The type of SD information being returned is indicated in the *mc_usage* field.

mc_usage
Indicates the type of SD information being returned using the **mc_sd_usage_t** enumeration.

```
typedef enum mc_sd_usage mc_sd_usage_t;
enum mc_sd_usage {
     MC_SD_USAGE_PATTR_RSRC_CLASS,
     MC_SD_USAGE_PATTR_RESOURCE,
     MC_SD_USAGE_DATTR_RSRC_CLASS,
     MC_SD_USAGE_DATTR_RESOURCE,
     MC_SD_USAGE_RSRC_ACTION_INPUT,
     MC_SD_USAGE_RSRC_ACTION_RESPONSE,
     MC_SD_USAGE_CLASS_ACTION_INPUT,
     MC_SD_USAGE_CLASS_ACTION_RESPONSE,
     MC_SD_USAGE_DEFINE_ARG,
     MC_SD_USAGE_UNDEFINE_ARG,
     MC_SD_USAGE_ONLINE_ARG,
     MC_SD_USAGE_OFFLINE_ARG,
     MC_SD_USAGE_RESET_ARG
};
```

Location

/usr/lib/libct_mc.a

Related reference:

"mc_define_resource_*" on page 58
This subroutine defines a new resource.

"mc_invoke_action_*" on page 78
This subroutine invokes an action on a resource.

"mc_invoke_class_action_*" on page 82
This subroutine invokes an action on a resource class.

"mc_offline_*" on page 88
This subroutine sends a request to the RMC subsystem to take a resource offline.

"mc_online_*" on page 92
This subroutine requests the RMC subsystem to bring a resource online.

mc_qdef_valid_values_*

This subroutine queries the RMC subsystem to obtain the definition of valid values.

Purpose

Queries the RMC subsystem to obtain the definition of valid values.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_qdef_valid_values_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_qdef_valid_values_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_qdef_valid_values_bp(
          mc_sess_hndl_t             sess_hndl,
          mc_qdef_valid_vals_rsp_t   **rsp_array,
          ct_uint32_t                *array_cnt,
          mc_qdef_opts_t             options,
          ct_char_t                  *rsrc_class_name,
          mc_vv_usage_t              vv_use,
          ct_char_t                  **names,
          ct_uint32_t                count)
```

- The **mc_qdef_valid_values_ap** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_qdef_valid_values_ap(
          mc_cmdgrp_hndl_t           cmdgrp_hndl,
          mc_qdef_valid_vals_rsp_t   **rsp_array,
          ct_uint32_t                *array_cnt,
          mc_qdef_opts_t             options,
          ct_char_t                  *rsrc_class_name,
          mc_vv_usage_t              vv_use,
          ct_char_t                  **names,
          ct_uint32_t                count)
```

- The **mc_qdef_valid_values_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_qdef_valid_values_bc(
          mc_sess_hndl_t             sess_hndl,
          mc_qdef_valid_vals_cb_t    *qdef_cb,
          void                       *qdef_cb_arg,
          mc_qdef_opts_t             options,
          ct_char_t                  *rsrc_class_name,
          mc_vv_usage_t              vv_use,
          ct_char_t                  **names,
          ct_uint32_t                count)
```

  The definition for the response callback is:

```
typedef void (mc_qdef_valid_vals_cb_t)(mc_sess_hndl_t,
mc_qdef_valid_vals_rsp_t *,
void *);
```

- The **mc_qdef_valid_values_ac** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_qdef_valid_values_ac(
          mc_cmdgrp_hndl_t           cmdgrp_hndl,
          mc_qdef_valid_vals_cb_t    *qdef_cb,
          void                       *qdef_cb_arg,
          mc_qdef_opts_t             options,
          ct_char_t                  *rsrc_class_name,
          mc_vv_usage_t              vv_use,
          ct_char_t                  **names,
          ct_uint32_t                count)
```

The definition for the response callback is:

```
typedef void (mc_qdef_valid_vals_cb_t)(mc_sess_hndl_t,
mc_qdef_valid_vals_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

qdef_cb Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

qdef_cb_arg
Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

options Specifies one of the following

MC_QDEF_OPTS_NODSCRP
No descriptions are returned.

MC_QDEF_OPTS_NONE
There are no options.

rsrc_class_name
Pointer to a resource class name. Identifies the resource class for which valid value information is to be returned.

vv_use Specifies, using one of the following values, the valid value information to be returned.

MC_VV_USAGE_PATTR_RESOURCE
Valid values of the persistent attributes of the resource of the specified resource class

MC_VV_USAGE_PATTR_RSRC_CLASS
Valid values of the persistent attributes of the specified resource class

MC_VV_USAGE_DATTR_RESOURCE
Valid values of the dynamic attributes, of variable type RMC_STATE, of the resource of the specified resource class

MC_VV_USAGE_DATTR_RSRC_CLASS
Valid values of the dynamic attributes, of variable type RMC_STATE, of the specified resource class

MC_VV_USAGE_RSRC_ACTION_INPUT
Valid values for the input to the resource actions of the specified resource class

MC_VV_USAGE_CLASS_ACTION_INPUT
Valid values for the input to the class actions of the specified resource class

MC_VV_USAGE_DEFINE_ARG
Valid values for the define resource command argument of the specified resource class

MC_VV_USAGE_UNDEFINE_ARG
Valid values for the undefine resource command argument of the specified resource class

MC_VV_USAGE_ONLINE_ARG
Valid values for the online command argument of the specified resource class

MC_VV_USAGE_OFFLINE_ARG
Valid values for the offline command argument of the specified resource class

MC_VV_USAGE_RESET_ARG
Valid values for the reset command argument of the specified resource class

names Specifies the attributes or actions for which valid value information should be returned. Specifies this using a pointer to an array of *count* elements of type **ct_char_t**. Each element in this array identifies an attribute or action (depending on the value of the *vv_use* parameter).

If the *count* parameter is 0, then the *names* parameter must be a NULL pointer. In this case, information is returned for all attributes or actions of the resource class.

count Specifies the number of elements in the *names* array. If 0, the *names* parameter must be a NULL pointer.

OUTPUT

rsp_array
A pointer to a location in which the RMC API will return a pointer to a response array of *array_cnt* elements. This parameter applies only to variations of this subroutine that use the pointer response method.

array_cnt
Identifies a location in which the RMC API will return the number of elements in the response array *rsp_array*. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_qdef_valid_values_*** subroutines can be used by the application to obtain the definition of valid values from the RMC subsystem. Valid values are a set of values and/or a set of value ranges that an attribute may assume or that may be used as input to the **mc_invoke_action_***, **mc_invoke_class_action_***, **mc_define_resource_*, mc_undefine_resource_***, **mc_online_***, **mc_offline_***, and **mc_reset_*** subroutines.

The definition of valid values often includes:

- the valid values
- the range of valid values
- a descriptive label for each value or range of values

However, some resource classes, instead of having a statically defined set of valid values, may have an action defined that can be invoked to obtain a set of valid values for an attribute or command argument. In such a case, the definition of valid values includes:

- the name of the action
- the action type (whether it is a resource action or a resource class action).

The response for these subroutines is a structure of type **mc_qdef_valid_vals_rsp_t**, and is described in Response structure.

This command cannot be used in an ordered command group.

Return values

For the **mc_qdef_valid_values_bp** and **mc_qdef_valid_values_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and one or more responses have been received and processed.

For the **mc_qdef_valid_values_ap** and **mc_qdef_valid_values_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_EORDERGROUP
An attempt was made to add the command to an ordered command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_EINVALIDCNTPTR
Invalid response count pointer specified.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before all responses could be received. If subroutine uses pointer response, check the appropriate array count for any responses. If the subroutine uses callback response, the callback will have been invoked for any responses received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_qdef_valid_vals_rsp_t**. Depending on the value of the *vv_use* parameter, the response contains the requested information for one persistent attribute, one dynamic attribute that has a variable type of RMC_STATE, one action input, or one command argument of the specified resource class. If more than one attribute or action name is specified by the *names* parameter, then there will be one response for each. In addition, if the valid values are structured data, then the response contains valid values for each element of the structured data. The amount of information returned will vary depending on the *options* and *names* parameters. If any of the query arguments are invalid, then only one response is returned, indicating the error (even if more that one attribute or action is specified).

The response structure definition is:

```
typedef struct mc_qdef_valid_vals_rsp   mc_qdef_valid_vals_rsp_t;
struct mc_qdef_valid_vals_rsp {
     mc_errnum_t                        mc_error;
     mc_valid_value_t                   *mc_valid_values;
     ct_uint32_t                        mc_count;
     ct_uint32_t                        mc_id;
     mc_vv_usage_t                      mc_usage;
     ct_data_type_t                     mc_data_type;
     ct_char_t                          *mc_action_name;
     mc_action_type_t                   mc_action_type;
};
```

The fields of this structure contain the following:

mc_error
If 0 (zero), this field indicates that the query was successful. Any other value is an error and indicates that the RMC subsystem could not provide some or all of the requested information. The error may also indicate that the command arguments were in error. The error codes imply which of the remaining fields in the structure are defined.

mc_valid_values

If the *mc_count* field is not zero, this field is a pointer to an array of *mc_count* elements of type **mc_valid_value_t**. Each entry in this array specifies a valid value or range of valid values for the attribute, input or argument.

```
typedef struct mc_valid_value       mc_valid_value_t;
struct mc_valid_value {
     ct_value_t                     mc_low_value;
     ct_value_t                     mc_high_value;
     ct_char_t                      *mc_label;
     ct_uint32_t                    mc_sd_element_index;
     ct_data_type_t                 mc_sd_element_data_type;
};
```

The fields of this structure contain the following:

mc_low_value

If the *mc_data_type* field specifies an arithmetic type and this field is less than the *mc_high_value* field, then this field is the lowest value in a range. The highest value in the range is specified in the *mc_high_value* field. If this field and the *mc_high_value* field specify the same value, then they both specify a single valid value.

If the *mc_data_type* field specifies a non-arithmetic type, but not CT_SD_PTR or CT_SD_PTR_ARRAY, this field specifies the valid value and the **mc_high_value** field is undefined.

mc_high_value

If the *mc_data_type* field specifies an arithmetic type and this field is greater than the *mc_low_value* field, then this field is the highest value in a range. The lowest value in the range is specified in the *mc_low_value* field. If this field and the *mc_low_value* field specify the same value, then they both specify a single valid value.

If the *mc_data_type* field specifies a non-arithmetic type, but not CT_SD_PTR or CT_SD_PTR_ARRAY, this field is undefined and the *mc_low_value* field specifies the valid value

mc_label

If non-NULL, this field specifies a pointer to a string containing a short description of the valid value or range.

mc_sd_element_index

If the *mc_data_type* field is CT_SD_PTR, this field specifies the index of the structured data (SD) element. If the *mc_data_type* field is CT_SD_PTR_ARRAY, then the valid values returned apply to all elements with the identical index value in each SD of the array. If the *mc_data_type* field is not CT_SD_PTR or CT_SD_PTR_ARRAY, then this field is undefined.

mc_sd_element_data_type

If the *mc_data_type* field is CT_SD_PTR or CT_SD_PTR_ARRAY, the data type of the element. If the *mc_data_type* field is not CT_SD_PTR or CT_SD_PTR_ARRAY, then this field is undefined.

mc_count

The number of entries in the *mc_valid_values* array.

mc_id

Depending on the type of valid value information being returned, this field contains the ID of a either a persistent attribute, a dynamic attribute or an action for which these valid values are being returned (or for which the named action can be used to obtain valid values). If the valid values are that of a command argument, this field is undefined. The type of valid value information being returned is indicated in the *mc_usage* field.

mc_usage

Indicates if the valid value information being returned is for:

- a persistent attribute
- a dynamic attribute that has a variable type of RMC_STATE
- action input
- a command argument used when calling the **mc_define_resource_***, **mc_undefine_resource_***, **mc_online_***, **mc_offline_*** or **mc_reset_*** subroutines.

The possible values for this field are defined by the **mc_vv_usage_t** enumeration.

```
typedef enum mc_vv_usage           mc_vv_usage_t;
enum mc_vv_usage {

     MC_VV_USAGE_PATTR_RSRC_CLASS,
     MC_VV_USAGE_PATTR_RESOURCE,
     MC_VV_USAGE_DATTR_RSRC_CLASS,
     MC_VV_USAGE_DATTR_RESOURCE,
     MC_VV_USAGE_RSRC_ACTION_INPUT,
     MC_VV_USAGE_CLASS_ACTION_INPUT,
     MC_VV_USAGE_DEFINE_ARG,
     MC_VV_USAGE_UNDEFINE_ARG,
     MC_VV_USAGE_ONLINE_ARG,
     MC_VV_USAGE_OFFLINE_ARG,
     MC_VV_USAGE_RESET_ARG
};
```

mc_data_type

If the returned valid values are those of an attribute, this field specifies the data type of the attribute. If the returned valid values are those of an action input or a command argument, this field is set to CT_SD_PTR.

mc_action_name

If the *mc_count* field is zero, then this field is the name of an action that can be invoked to obtain valid values.

mc_action_type

Indicates whether the action identified by the *mc_action_name* field is a resource action or class action. The possible values for this field are defined by the **mc_action_type_t** enumeration.

```
typedef enum mc_action_type mc_action_type_t;
enum mc_action_type {
     MC_ACTION_TYPE_RESOURCE,
     MC_ACTION_TYPE_CLASS
};
```

Location

/usr/lib/libct_mc.a

mc_query_d_handle_*

This subroutine queries the RMC subsystem to obtain the dynamic attribute values of a resource.

Purpose

Queries the RMC subsystem to obtain the dynamic attribute values of a resource. The resource is identified using a resource handle.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_query_d_handle_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_query_d_handle_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_query_d_handle_bp(
          mc_sess_hndl_t            sess_hndl,
          mc_query_rsp_t            **response,
          ct_resource_handle_t      rsrc_hndl,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count)
```

- The **mc_query_d_handle_ap** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_query_d_handle_ap(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_query_rsp_t            **response,
          ct_resource_handle_t      rsrc_hndl,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count)
```

- The **mc_query_d_handle_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_query_d_handle_bc(
          mc_sess_hndl_t            sess_hndl,
          mc_query_cb_t             *query_cb,
          void                      *query_cb_arg,
          ct_resource_handle_t      rsrc_hndl,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count)
```

  The definition for the response callback is:

```
typedef void (mc_query_cb_t)(mc_sess_hndl_t,
mc_query_rsp_t *,
void *);
```

- The **mc_query_d_handle_ac** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_query_d_handle_ac(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_query_cb_t             *query_cb,
          void                      *query_cb_arg,
          ct_resource_handle_t      rsrc_hndl,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count)
```

  The definition for the response callback is:

```
typedef void (mc_query_cb_t)(mc_sess_hndl_t,
mc_query_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

query_cb
Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

query_cb_arg
Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

rsrc_hndl
The resource handle that identifies the resource whose dynamic attribute values are being requested by the application. A resource handle is returned in the response structure for many RMC API subroutines including the **mc_define_resource_*** subroutine. An array of resource handles for resources of a particular resource class is returned in the response structure for the **mc_enumerate_resources_*** and **mc_enumerate_permitted_rsrcs_*** subroutines. To validate the resource handle before calling this subroutine, the application can call one of the **mc_validate_rsrc_hndl_*** subroutines.

return_attrs
An array of *attr_count* pointers to dynamic attribute names. This parameter, in conjunction with the *attr_count* parameter, enables the application to specify dynamic attributes to be included in the query response. If any of the specified attributes are not supported by the resource, they will not be included in the query response.

If *attr_count* is 0 (zero), *return_attrs* should be a NULL pointer. In this case, all dynamic attribute values for a resource are returned.

attr_count
Indicates the number of pointers to dynamic attribute names in the *return_attrs* array. This parameter, in conjunction with the *return_attrs* parameter, enables the application to specify dynamic attributes to be included in the query response. If set to 0, all dynamic attribute values for the resource are returned.

OUTPUT

response
A pointer to a location in which the RMC API will return a pointer to the response. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_query_d_handle_*** subroutines can be used by the application to obtain dynamic attribute values of a resource from the RMC subsystem. The resource is identified using a resource handle (specified by the *rsrc_hndl* parameter).

Dynamic attributes that are of variable type Quantum cannot be queried since such attributes have no value.

The response for these subroutines is a structure of type **mc_query_rsp_t**, and is described in Response structure.

Custom dynamic attributes can be specified. However, all selected resources must have matching values for their **CustomDynamicAttributes** persistent resource attributes if any custom dynamic attributes are used in this subroutine. Furthermore, if the RMC subsystem cannot obtain the values of the **CustomDynamicAttributes** persistent resource attributes for the selected resources, this subroutine returns an error in the Query Response, indicating that the specified custom dynamic attributes could not be validated.

This subroutine cannot be used in an ordered command group.

Security

To query resource dynamic attributes, the user of the calling application must have either **q** or **r** permission specified in an ACL entry for the resource.

Return values

For the **mc_query_d_handle_bp** and **mc_query_d_handle_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and a response has been received and processed.

For the **mc_query_d_handle_ap** and **mc_query_d_handle_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_EORDERGROUP
An attempt was made to add the command to an ordered command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before the response could be received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before the response could be received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before the response could be received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_query_rsp_t**. The response contains the values of the requested attributes for the specified resource.

The response structure definition is:

```
typedef struct mc_query_rsp             mc_query_rsp_t;
struct mc_query_rsp {
     mc_errnum_t                        mc_error;
     ct_resource_handle_t               mc_rsrc_hndl;
     mc_attribute_t                     *mc_attrs;
     ct_uint32_t                        mc_attr_count;
};
```

The fields of this structure contain the following:

mc_error
If 0 (zero), this field indicates that the query was successful. Any other value is an error and indicates that the resource monitoring and control (RMC) subsystem or a resource manager could not provide some or all of the requested information. The error may also indicate that the command arguments were in error. The error codes imply which of the remaining fields in the structure are defined.

mc_rsrc_hndl
Specifies the resource handle of the resource that was queried, and whose attributes are contained in this response.

mc_attrs
A pointer to an array of the requested attributes.

mc_attr_count
Indicates the number of entries in the *mc_attrs* array.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_class_query_d_*" on page 42
This subroutine queries the RMC subsystem to obtain the dynamic attribute values of a resource class.

"mc_qdef_d_attribute_*" on page 102
This subroutine queries the RMC subsystem to obtain dynamic attribute definitions for a resource or resource class.

mc_query_d_select_*

This subroutine queries the RMC subsystem to obtain the dynamic attribute values of one or more resources of a resource class.

Purpose

Queries the RMC subsystem to obtain the dynamic attribute values of one or more resources of a resource class. The resources are identified using attribute selection.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_query_d_select_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_query_d_select_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_query_d_select_bp(
          mc_sess_hndl_t            sess_hndl,
          mc_query_rsp_t            **rsp_array,
          ct_uint32_t               *array_cnt,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 *select_attrs,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count)
```

- The **mc_query_d_select_ap** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_query_d_select_ap(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_query_rsp_t            **rsp_array,
          ct_uint32_t               *array_cnt,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 *select_attrs,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count)
```

- The **mc_query_d_select_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_query_d_select_bc(
          mc_sess_hndl_t            sess_hndl,
          mc_query_cb_t             *query_cb,
          void                      *query_cb_arg,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 *select_attrs,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count)
```

  The definition for the response callback is:

```
typedef void (mc_query_cb_t)(mc_sess_hndl_t,
mc_query_rsp_t *,
void *);
```

- The **mc_query_d_select_ac** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_query_d_select_ac(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_query_cb_t             *query_cb,
          void                      *query_cb_arg,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 *select_attrs,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count)
```

  The definition for the response callback is:

```
typedef void (mc_query_cb_t)(mc_sess_hndl_t,
mc_query_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
: The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
: The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

query_cb
: Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

query_cb_arg
: Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

rsrc_class_name
Pointer to a resource class name. Identifies the parent resource class of the resource(s) whose dynamic attribute values are being requested by the application.

select_attrs
A selection string expression that identifies one or more resources of the resource class identified by the *rsrc_class_name* parameter. Dynamic attribute values will be returned for all resources of the resource that match the selection string expression.

return_attrs
An array of *attr_count* pointers to dynamic attribute names. This parameter, in conjunction with the *attr_count* parameter, enables the application to specify dynamic attributes to be included in the query response. If any of the specified attributes are not supported by the resource, they will not be included in the query response.

If *attr_count* is 0 (zero), *return_attrs* should be a NULL pointer. In this case, all dynamic attribute values for a resource are returned.

attr_count
Indicates the number of pointers to dynamic attribute names in the *return_attrs* array. This parameter, in conjunction with the *return_attrs* parameter, enables the application to specify dynamic attributes to be included in the query response. If set to 0, all dynamic attribute values for the resource are returned.

OUTPUT

rsp_array
A pointer to a location in which the RMC API will return a pointer to a response array of *array_cnt* elements. This parameter applies only to variations of this subroutine that use the pointer response method.

array_cnt
Identifies a location in which the RMC API will return the number of elements in the response array *rsp_array*. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_query_d_select_*** subroutines can be used by the application to obtain the dynamic attribute values of one or more resources of a resource class (identified by the **rsrc_class_name** parameter). The resource or resources are identified using a selection string (specified by the *select_attrs* parameter)

Dynamic attributes that are of variable type Quantum cannot be queried since such attributes have no value.

The response for these subroutines is a structure of type **mc_query_rsp_t**, and is described in Response structure.

Custom dynamic attributes can be specified. However, all selected resources must have matching values for their **CustomDynamicAttributes** persistent resource attributes if any custom dynamic attributes are used in this subroutine. Furthermore, if the RMC subsystem cannot obtain the values of the **CustomDynamicAttributes** persistent resource attributes for the selected resources, this subroutine returns an error in the Query Response, indicating that the specified custom dynamic attributes could not be validated.

This subroutine cannot be used in an ordered command group.

Security

To query resource dynamic attributes, the user of the calling application must have either **q** or **r** permission specified in the ACL entry for the resource(s).

Return values

For the **mc_query_d_select_bp** and **mc_query_d_select_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and one or more responses have been received and processed.

For the **mc_query_d_select_ap** and **mc_query_d_select_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_EORDERGROUP
An attempt was made to add the command to an ordered command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_EINVALIDCNTPTR
Invalid response count pointer specified.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before all responses could be received. If subroutine uses pointer response, check the appropriate array count for any responses. If the subroutine uses callback response, the callback will have been invoked for any responses received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_query_rsp_t**. A response contains the values of the requested attributes for one specified resource. If more than one resource is identified by the *select_attrs* parameter, then there will be one response for each resource. If any of the query arguments are invalid, then only one response is returned, indicating the error (even if more than one resource was identified by the *select_attrs* parameter).

The response structure definition is:

```
typedef struct mc_query_rsp             mc_query_rsp_t;
struct mc_query_rsp {
     mc_errnum_t                        mc_error;
     ct_resource_handle_t               mc_rsrc_hndl;
     mc_attribute_t                     *mc_attrs;
     ct_uint32_t                        mc_attr_count;
};
```

The fields of this structure contain the following:

mc_error
If 0 (zero), this field indicates that the query was successful. Any other value is an error and indicates that the resource monitoring and control (RMC) subsystem or a resource manager could not provide some or all of the requested information. The error may also indicate that the command arguments were in error. The error codes imply which of the remaining fields in the structure are defined.

mc_rsrc_hndl
Specifies the resource handle of the resource that was queried, and whose attributes are contained in this response.

mc_attrs
A pointer to an array of the requested attributes.

mc_attr_count
Indicates the number of entries in the *mc_attrs* array.

Location

/usr/lib/libct_mc.a

Related reference:

“mc_class_query_d_*” on page 42
This subroutine queries the RMC subsystem to obtain the dynamic attribute values of a resource class.
“mc_class_query_p_*” on page 47
This subroutine queries the RMC subsystem to obtain the persistent attribute values of a resource class.

mc_query_event_*

This subroutine queries the RMC subsystem to obtain an event's current state.

Purpose

Queries the RMC subsystem to obtain an event's current state.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_query_event_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_query_event_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_query_event_bp(
          mc_sess_hndl_t             sess_hndl,
          mc_query_event_rsp_t       **response,
          mc_registration_id_t       registration_id)
```

- The **mc_query_event_ap** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_query_event_ap(
          mc_cmdgrp_hndl_t           cmdgrp_hndl,
          mc_query_event_rsp_t       **response,
          mc_registration_id_t       registration_id)
```

- The **mc_query_event_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_query_event_bc(
          mc_sess_hndl_t             sess_hndl,
          mc_query_event_cb_t        *query_event_cb,
          void                       *query_event_cb_arg,
          mc_registration_id_t       registration_id)
```

  The definition for the response callback is:

```
typedef void (mc_query_event_cb_t)(mc_sess_hndl_t,
mc_query_event_rsp_t *,
void *);
```

- The **mc_query_event_ac** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_query_event_ac(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_query_event_cb_t       *query_event_cb,
          void                      *query_event_cb_arg,
          mc_registration_id_t      registration_id)
```

The definition for the response callback is:

```
typedef void (mc_query_event_cb_t)(mc_sess_hndl_t,
mc_query_event_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
: The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
: The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

query_event_cb
: Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

query_event_cb_arg
: Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

registration_id
: The event registration ID that identifies the event whose state information is being requested. A valid event registration ID is one returned in a successful response from one of the **mc_reg_event_select_*** or **mc_reg_event_handle_*** subroutines. In order for the registration ID to be valid, the response from the **mc_reg_event_select_*** or **mc_reg_event_handle_*** subroutine must have indicated that the event was successfully registered.

  An event returned in a Registration Response from one of the **mc_reg_class_event_*** subroutines is not valid.

OUTPUT

response
: A pointer to a location in which the RMC API will return a pointer to the response. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_query_event_*** subroutines can be used by the application to obtain the current state of an event previously registered by the application using the **mc_reg_event_select_*** or **mc_reg_event_handle_*** subroutines. These subroutines return the current event state by forcing the generation of an event notification for each resource assigned to the event registration (specified by the *registration_id* parameter).

Since the event registrations are forced and not the result of the event expression or rearm event expression evaluating to TRUE, the expression being evaluated will not be toggled.

If the event notification is an error event, then the error indicates the monitoring status as one of the following:

- waiting for monitoring to commence
- monitoring is suspended due to termination of the resource manager
- monitoring is suspended due to the departure of the node from the cluster

As these events are generated, the expression used in the evaluation is not toggled. If an expression includes a qualifier, the qualifier is ignored when evaluating the expression and the internal state of the qualifier remains unchanged.

The response for these subroutines is a structure of type **mc_query_event_rsp_t**, and is described in Response structure.

If the **mc_query_event_bp** or **mc_query_event_ap** subroutines are called, the RMC subsystem guarantees that the response is not delivered to the application (in other words, the command or command group does not complete) until all event notification callbacks for the specified event, resulting from this command, have been executed. The application cannot invoke any other **mc_query_event_*** subroutine, specifying the same registration ID, until the response is delivered.

If the **mc_query_event_bc** or **mc_query_event_ac** subroutines are called, the RMC subsystem guarantees that the response callback is invoked only when all event notification callbacks for the specified event, resulting from this command, have been executed. The application cannot invoke any other **mc_query_event_*** subroutine, specifying the same registration ID, until the response has returned.

In order to avoid deadlocks, the **mc_query_event_bp** and **mc_query_event_bc** subroutines may not be called from within an event notification callback. Furthermore, the **mc_send_cmd_grp_wait** subroutine may not be called from within an event notification callback if the associated command group contains commands added by the **mc_query_event_ap** or **mc_query_event_ac** subroutines.

This command cannot be used in an ordered command group.

Security

To query an event state, the user of the calling application must have either **e** or **r** permission specified in an ACL entry for the associated resource or resource class.

Return values

For the **mc_query_event_bp** and **mc_query_event_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and a response has been received and processed.

For the **mc_query_event_ap** and **mc_query_event_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_EORDERGROUP
An attempt was made to add the command to an ordered command group.

MC_EDEADLOCK
An attempt was made to invoke the command from within an event notification callback.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_EINVALIDEID
The specified registration ID is invalid.

MC_ECLASSEID
The registration ID specified a class event

MC_EQEVENTACTIVE
A previous query event command has not yet completed.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before the response could be received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before the response could be received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before the response could be received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_query_event_rsp_t**.

The response structure definition is:

```
typedef struct mc_query_event_rsp       mc_query_event_rsp_t;
struct mc_query_event_rsp {
     mc_errnum_t                        mc_error;
     mc_registration_id_t               mc_registration_id;
     ct_uint32_t                        mc_event_count;
};
```

The fields of this structure contain the following:

mc_error
: If 0 (zero), this field indicates that the event query command has completed. Any other value is an error and indicates that the RMC subsystem could not complete the command. The error may also indicate that the command arguments were in error. The error codes imply which of the remaining fields in the structure are defined.

mc_registration_id
: The event registration ID of the event that was queried.

mc_event_count
: The number of events that were generated as a result of this command. If this field is not zero, then the RMC subsystem guarantees that this response is passed to the application only after all event notification callbacks for the events generated by this command have been executed.

Location

/usr/lib/libct_mc.a

mc_query_p_handle_*

This subroutine queries the RMC subsystem to obtain the persistent attribute values of a resource.

Purpose

Queries the RMC subsystem to obtain the persistent attribute values of a resource. The resource is identified using a resource handle.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_query_p_handle_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_query_p_handle_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

  ```
  #include <rsct/ct_mc_v6.h>

  ct_int32_t
       mc_query_p_handle_bp(
            mc_sess_hndl_t              sess_hndl,
  ```

```
          mc_query_rsp_t            **response,
          ct_resource_handle_t      rsrc_hndl,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count)
```

- The **mc_query_p_handle_ap** subroutine adds the command to a command group. If used in an ordered command group, however, please note that all other commands in the command group must also be the form that uses a resource handle to specify the command target. The same resource handle must be used on all commands in the command group.

  To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_query_p_handle_ap(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_query_rsp_t            **response,
          ct_resource_handle_t      rsrc_hndl,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count)
```

- The **mc_query_p_handle_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_query_p_handle_bc(
          mc_sess_hndl_t            sess_hndl,
          mc_query_cb_t             *query_cb,
          void                      *query_cb_arg,
          ct_resource_handle_t      rsrc_hndl,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count)
```

  The definition for the response callback is:

```
typedef void (mc_query_cb_t)(mc_sess_hndl_t,
mc_query_rsp_t *,
void *);
```

- The **mc_query_p_handle_ac** subroutine adds the command to a command group. If used in an ordered command group, however, please note that all other commands in the command group must also be the form that uses a resource handle to specify the command target.

  To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_query_p_handle_ac(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_query_cb_t             *query_cb,
          void                      *query_cb_arg,
          ct_resource_handle_t      rsrc_hndl,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count)
```

  The definition for the response callback is:

```
typedef void (mc_query_cb_t)(mc_sess_hndl_t,
mc_query_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

query_cb
Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

query_cb_arg
Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

rsrc_hndl
The resource handle that identifies the resource whose persistent attribute values are being requested by the application. A resource handle is returned in the response structure for many RMC API subroutines including the **mc_define_resource_*** subroutine. An array of resource handles for resources of a particular resource class is returned in the response structure for the **mc_enumerate_resources_*** and **mc_enumerate_permitted_rsrcs_*** subroutines. To validate the resource handle before calling this subroutine, the application can call one of the **mc_validate_rsrc_hndl_*** subroutines.

return_attrs
An array of *attr_count* pointers to persistent attribute names. This parameter, in conjunction with the *attr_count* parameter, enables the application to specify persistent attributes to be included in the query response. If any of the specified attributes are not supported by the resource, they will not be included in the query response.

If *attr_count* is 0 (zero), *return_attrs* should be a NULL pointer. In this case, all persistent attribute values for a resource are returned.

attr_count
Indicates the number of pointers to persistent attribute names in the *return_attrs* array. This parameter, in conjunction with the *return_attrs* parameter, enables the application to specify persistent attributes to be included in the query response. If set to 0, all persistent attribute values for the resource are returned.

OUTPUT

response
A pointer to a location in which the RMC API will return a pointer to the response. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_query_p_handle_*** subroutines can be used by the application to obtain persistent attribute values of a resource from the RMC subsystem. The resource is identified using a resource handle (specified by the *rsrc_hndl* parameter).

Persistent attributes that are of variable type Quantum cannot be queried since such attributes have no value.

The response for these subroutines is a structure of type **mc_query_rsp_t**, and is described in Response structure.

If this command is used in an ordered command group, please note that all other commands in the command group must be the form that uses a resource handle to specify the command target. The same resource handle must be used on all commands in the command group.

Security

To query resource persistent attributes, the user of the calling application must have either **q** or **r** permission specified in an ACL entry for the resource.

Return values

For the **mc_query_p_handle_bp** and **mc_query_p_handle_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and a response has been received and processed.

For the **mc_query_p_handle_ap** and **mc_query_p_handle_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_ETARGETMISMATCH
The target specified for the command does not match the target of the command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before the response could be received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before the response could be received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before the response could be received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_query_rsp_t**. The response contains the values of the requested attributes for the specified resource.

The response structure definition is:

```
typedef struct mc_query_rsp             mc_query_rsp_t;
struct mc_query_rsp {
     mc_errnum_t                        mc_error;
     ct_resource_handle_t               mc_rsrc_hndl;
     mc_attribute_t                     *mc_attrs;
     ct_uint32_t                        mc_attr_count;
};
```

The fields of this structure contain the following:

mc_error
If 0 (zero), this field indicates that the query was successful. Any other value is an error and indicates that the RMC subsystem or a resource manager could not provide some or all of the requested information. The error may also indicate that the command arguments were in error. The error codes imply which of the remaining fields in the structure are defined.

mc_rsrc_hndl
Specifies the resource handle of the resource that was queried, and whose attributes are contained in this response.

mc_attrs
A pointer to an array of the requested attributes.

mc_attr_count
Indicates the number of entries in the *mc_attrs* array.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_class_query_d_*" on page 42
This subroutine queries the RMC subsystem to obtain the dynamic attribute values of a resource class.

mc_query_p_select_*

This subroutine queries the RMC subsystem to obtain the persistent attribute values of one or more resources of a resource class.

Purpose

Queries the RMC subsystem to obtain the persistent attribute values of one or more resources of a resource class. The resources are identified using attribute selection.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_query_p_select_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_query_p_select_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_query_p_select_bp(
          mc_sess_hndl_t            sess_hndl,
          mc_query_rsp_t            **rsp_array,
          ct_uint32_t               *array_cnt,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 *select_attrs,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count)
```

- The **mc_query_p_select_ap** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_query_p_select_ap(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_query_rsp_t            **rsp_array,
          ct_uint32_t               *array_cnt,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 *select_attrs,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count)
```

- The **mc_query_p_select_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_query_p_select_bc(
          mc_sess_hndl_t            sess_hndl,
          mc_query_cb_t             *query_cb,
          void                      *query_cb_arg,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 *select_attrs,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count)
```

  The definition for the response callback is:

```
typedef void (mc_query_cb_t)(mc_sess_hndl_t,
mc_query_rsp_t *,
void *);
```

- The **mc_query_p_select_ac** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_query_p_select_ac(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_query_cb_t             *query_cb,
          void                      *query_cb_arg,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 *select_attrs,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count)
```

  The definition for the response callback is:

```
typedef void (mc_query_cb_t)(mc_sess_hndl_t,
mc_query_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
: The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
: The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

query_cb
: Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

query_cb_arg
: Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

rsrc_class_name
: Pointer to a resource class name. Identifies the parent resource class of the resource(s) whose persistent attribute values are being requested by the application.

select_attrs
: A selection string expression that identifies one or more resources of the resource class identified by the *rsrc_class_name* parameter. Persistent attribute values will be returned for all resources of the resource that match the selection string expression.

return_attrs
: An array of *attr_count* pointers to persistent attribute names. This parameter, in conjunction with the *attr_count* parameter, enables the application to specify persistent attributes to be included in the query response. If any of the specified attributes are not supported by the resource, they will not be included in the query response.

  If *attr_count* is 0 (zero), *return_attrs* should be a NULL pointer. In this case, all persistent attribute values for a resource are returned.

attr_count
Indicates the number of pointers to persistent attribute names in the *return_attrs* array. This parameter, in conjunction with the *return_attrs* parameter, enables the application to specify persistent attributes to be included in the query response. If set to 0, all persistent attribute values for the resource are returned.

OUTPUT

rsp_array
A pointer to a location in which the RMC API will return a pointer to a response array of *array_cnt* elements. This parameter applies only to variations of this subroutine that use the pointer response method.

array_cnt
Identifies a location in which the RMC API will return the number of elements in the response array *rsp_array*. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_query_d_select_*** subroutines can be used by the application to obtain the persistent attribute values of one or more resources of a resource class (identified by the **rsrc_class_name** parameter). The resource or resources are identified using a selection string (specified by the *select_attrs* parameter)

Persistent attributes that are of variable type Quantum cannot be queried since such attributes have no value.

The response for these subroutines is a structure of type **mc_query_rsp_t**, and is described in Response structure.

This command cannot be used in an ordered command group.

Security

To query resource persistent attributes, the user of the calling application must have either **q** or **r** permission specified in the ACL entry for the resource(s).

Return values

For the **mc_query_p_select_bp** and **mc_query_p_select_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and one or more responses have been received and processed.

For the **mc_query_p_select_ap** and **mc_query_p_select_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_ETARGETMISMATCH
The target specified for the command does not match the target of the command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_EINVALIDCNTPTR
Invalid response count pointer specified.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before all responses could be received. If subroutine uses pointer response, check the appropriate array count for any responses. If the subroutine uses callback response, the callback will have been invoked for any responses received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_query_rsp_t**. The response contains the values of the requested attributes for one specified resource. If more than one resource is identified by the *select_attrs* parameter, then there will be one response for each resource. If any of the query arguments are invalid, then only one response is returned, indicating the error (even if more than one resource was identified by the *select_attrs* parameter).

The response structure definition is:

```
typedef struct mc_query_rsp             mc_query_rsp_t;
struct mc_query_rsp {
     mc_errnum_t                        mc_error;
     ct_resource_handle_t               mc_rsrc_hndl;
     mc_attribute_t                     *mc_attrs;
     ct_uint32_t                        mc_attr_count;
};
```

The fields of this structure contain the following:

mc_error
: If 0 (zero), this field indicates that the query was successful. Any other value is an error and indicates that the RMC subsystem or a resource manager could not provide some or all of the requested information. The error may also indicate that the command arguments were in error. The error codes imply which of the remaining fields in the structure are defined.

mc_rsrc_hndl
: Specifies the resource handle of the resource that was queried, and whose attributes are contained in this response.

mc_attrs
: A pointer to an array of the requested attributes.

mc_attr_count
: Indicates the number of entries in the *mc_attrs* array.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_class_query_d_*" on page 42
This subroutine queries the RMC subsystem to obtain the dynamic attribute values of a resource class.

"mc_class_query_p_*" on page 47
This subroutine queries the RMC subsystem to obtain the persistent attribute values of a resource class.

mc_refresh_config_*

This subroutine refreshes the configuration of resources within a resource class.

Purpose

Refreshes the configuration of resources within a resource class.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_refresh_config_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_refresh_config_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

  ```
  #include <rsct/ct_mc_v6.h>

  ct_int32_t
       mc_refresh_config_bp(
  ```

```
        mc_sess_hndl_t          sess_hndl,
        mc_class_name_rsp_t     **rsp_array,
        ct_uint32_t             *array_cnt,
        ct_char_t               *rsrc_class_name)
```

- The **mc_refresh_config_ap** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
    mc_refresh_config_ap(
        mc_cmdgrp_hndl_t        cmdgrp_hndl,
        mc_class_name_rsp_t     **rsp_array,
        ct_uint32_t             *array_cnt,
        ct_char_t               *rsrc_class_name)
```

- The **mc_refresh_config_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>
ct_int32_t
    mc_refresh_config_bc(
        mc_sess_hndl_t          sess_hndl,
        mc_refresh_cfg_cb_t     *refresh_cfg_cb,
        void                    *refresh_cfg_cb_arg,
        ct_char_t               *rsrc_class_name)
```

  The definition for the response callback is:

```
typedef void (mc_refresh_cfg_cb_t)(mc_sess_hndl_t,
mc_class_name_rsp_t *,
void *);
```

- The **mc_refresh_config_ac** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
    mc_refresh_config_ac(
        mc_cmdgrp_hndl_t        cmdgrp_hndl,
        mc_refresh_cfg_cb_t     *refresh_cfg_cb,
        void                    *refresh_cfg_cb_arg,
        ct_char_t               *rsrc_class_name)
```

  The definition for the response callback is:

```
typedef void (mc_refresh_cfg_cb_t)(mc_sess_hndl_t,
mc_class_name_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
: The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
: The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

refresh_cfg_cb
Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

refresh_cfg_cb_arg
Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

rsrc_class_name
Pointer to a resource class name. Identifies the resource class whose resource configuration is to be refreshed.

OUTPUT

rsp_array
A pointer to a location in which the RMC API will return a pointer to a response array of *array_cnt* elements. This parameter applies only to variations of this subroutine that use the pointer response method.

array_cnt
Identifies a location in which the RMC API will return the number of elements in the response array *rsp_array*. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_refresh_config_*** subroutines can be used by the application to have the RMC subsystem refresh the configuration of the resources within one resource class (identified by the *rsrc_class_name* parameter). If the application is monitoring any of the resources within the resource class, it may receive events as the configuration is refreshed.

The response for these subroutines is a structure of type **mc_class_name_rsp_t**, and is described in Response structure.

This command cannot be used in an ordered command group.

Security

To refresh resource configuration, the user of the calling application must have either **c** or **w** permission specified in an ACL entry for the resource class.

Return values

For the **mc_refresh_config_bp** and **mc_refresh_config_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and one or more responses have been received and processed.

For the **mc_refresh_config_ap** and **mc_refresh_config_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_EORDERGROUP
An attempt was made to add the command to an ordered command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_EINVALIDCNTPTR
Invalid response count pointer specified.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before all responses could be received. If subroutine uses pointer response, check the appropriate array count for any responses. If the subroutine uses callback response, the callback will have been invoked for any responses received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_class_name_rsp_t**. This command results in one response if there are no errors, and one or more responses if there are one or more errors.

The response structure definition is:

```
typedef struct mc_class_name_rsp        mc_class_name_rsp_t;
struct mc_class_name_rsp {
     mc_errnum_t                        mc_error;
     ct_char_t                          *mc_class_name;
};
```

The fields of this structure contain the following:

mc_error
: If 0 (zero), this field indicates that the command was successful. Any other value is an error. If there is an error, the error codes indicate whether the class name specified by the *mc_class_name* field is invalid or if the command could not be completed for the resource class. The error may also indicate that the command arguments were in error.

mc_class_name
: The name of the resource class that was the target of the command.

Location

/usr/lib/libct_mc.a

mc_reg_class_event_*

This subroutine registers a resource class event with the RMC subsystem.

Purpose

Registers a resource class event with the RMC subsystem.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_reg_class_event_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_reg_class_event_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_reg_class_event_bp(
          mc_sess_hndl_t            sess_hndl,
          mc_reg_rsp_t              **response,
          mc_reg_opts_t             options,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 **pd_names,
          ct_uint32_t               name_count,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count,
```

```
        ct_char_t                   *expr,
        ct_char_t                   *raexpr,
        mc_class_event_cb_t         *event_cb,
        void                        *event_cb_arg)
```

The definition for the event notification callback is:

```
typedef void (mc_class_event_cb_t)(mc_sess_hndl_t,
mc_class_event_t *,
void *);
```

- The **mc_reg_class_event_ap** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
    mc_reg_class_event_ap(
        mc_cmdgrp_hndl_t            cmdgrp_hndl,
        mc_reg_rsp_t                **response,
        mc_reg_opts_t               options,
        ct_char_t                   *rsrc_class_name,
        ct_char_t                   **pd_names,
        ct_uint32_t                 name_count,
        ct_char_t                   **return_attrs,
        ct_uint32_t                 attr_count,
        ct_char_t                   *expr,
        ct_char_t                   *raexpr,
        mc_class_event_cb_t         *event_cb,
        void                        *event_cb_arg)
```

The definition for the event notification callback is:

```
typedef void (mc_class_event_cb_t)(mc_sess_hndl_t,
mc_class_event_t *,
void *);
```

- The **mc_reg_class_event_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
    mc_reg_class_event_bc(
        mc_sess_hndl_t              sess_hndl,
        mc_reg_cb_t                 *reg_cb,
        void                        *reg_cb_arg,
        mc_reg_opts_t               options,
        ct_char_t                   *rsrc_class_name,
        ct_char_t                   **pd_names,
        ct_uint32_t                 name_count,
        ct_char_t                   **return_attrs,
        ct_uint32_t                 attr_count,
        ct_char_t                   *expr,
        ct_char_t                   *raexpr,
        mc_class_event_cb_t         *event_cb,
        void                        *event_cb_arg)
```

The definition for the response callback is:

```
typedef void (mc_reg_cb_t)(mc_sess_hndl_t,
mc_reg_rsp_t *,
void *);
```

The definition for the event notification callback is:

```
typedef void (mc_class_event_cb_t)(mc_sess_hndl_t,
mc_class_event_t *,
void *);
```

- The **mc_reg_class_event_ac** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_reg_class_event_ac(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_reg_cb_t               *reg_cb,
          void                      *reg_cb_arg,
          mc_reg_opts_t             options,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 **pd_names,
          ct_uint32_t               name_count,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count,
          ct_char_t                 *expr,
          ct_char_t                 *raexpr,
          mc_class_event_cb_t       *event_cb,
          void                      *event_cb_arg)
```

  The definition for the response callback is:

```
typedef void (mc_reg_cb_t)(mc_sess_hndl_t,
mc_reg_rsp_t *,
void *);
```

  The definition for the event notification callback is:

```
typedef void (mc_class_event_cb_t)(mc_sess_hndl_t,
mc_class_event_t *,
void *);
```

Parameters

INPUT

sess_hndl
: The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
: The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

reg_cb Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

reg_cb_arg
: Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

options Specifies the value **MC_REG_OPTS_NONE** to indicate that there are no options or is the bitwise inclusive OR of one or more of the following options:

MC_REG_OPTS_IMMED_EVAL
: Generate an event at the first observation of the resource class specified in the event expression (identified by the *expr* parameter). Generate the event even if the event expression evaluates to FALSE. This option enables the application to obtain the value of the attribute when the event is registered.

MC_REG_OPTS_NO_REG
Do not register the event. Instead, merely evaluate the remaining arguments for syntactical correctness and return an appropriate response.

MC_REG_OPTS_NO_TOGGLE
If this option is set when a re-arm expression is specified, the re-arm expression is not a toggle. Rather, upon each evaluation, the primary expression is evaluated first. If the expression evaluates to True, an event is generated. If the expression evaluates to FALSE, the re-arm expression is then evaluated. If the re-arm expression is True, an event is generated. This evaluation procedure occurs if any of the attributes in either expression is reported by the resource manager to have a new value.

This option implicitly sets the MC_REG_OPTS_REARM_EVENT option. If a re-arm expression is not specified, the MC_REG_OPTS_NO_TOGGLE option is ignored.

The re-arm expression is considered an alternative expression to be evaluated; it does not "re-arm" a trigger that is specified by the primary expression.

MC_REG_OPTS_REARM_EVENT
Also trigger events when the rearm event expression (identified by the *raexpr* parameter) evaluates to True.

rsrc_class_name
Pointer to a resource class name. Identifies the resource class for which the event is being registered.

pd_names

Table 21. mc_reg_class_event_* subroutine conditional pd_names parameter functions

If:	Then:
The management style of the resource class is globalized and the session scope is DM	The parameter should be a pointer to an array of *name_count* peer domain names. Since a session scope of DM refers to a CSM management domain, and, since such a domain can contain multiple peer domains, this enables the application to specify the peer domain(s) where the resource class event should be registered. To specify all peer domains within the management domain, this parameter should be a Null pointer, and the *name_count* parameter should be 0 (zero).
The management style of the resource class is subdivided or the session scope is not DM	This parameter should be a Null pointer, and the *name_count* parameter should be 0 (zero)

name_count
Identifies the number of pointers in the *pd_names* array. If *pd_names* is a Null pointer, this parameter must be 0 (zero)

return_attrs
An array of *attr_count* pointers to persistent attribute names. This parameter, in conjunction with the *attr_count* parameter, enables the application to specify additional persistent attribute values to be included in the event notification.

If the *attr_count* is 0 (zero), *return_attrs* should be a Null pointer.

attr_count
Identifies the number of pointers to persistent attribute names in the *return_attrs* array. This parameter, in conjunction with the *return_attrs* parameter, enables the application to specify additional persistent attribute values to be included in the event notification.

expr A pointer to the event expression. In many cases, the event expression consists of an attribute name, a mathematical expression symbol, and a constant.

raexpr A pointer to a rearm event expression.

If no rearm event expression is needed for the resource class event, then this parameter can be a Null pointer.

If a pointer to a rearm event expression is provided, then RMC will stop evaluating the event expression once it evaluates to True, and instead will evaluate the rearm event expression until it is True. Once the rearm event expression evaluates to True, the resource class event is rearmed. In other words, RMC will once again evaluate the event expression identified by the *expr* parameter.

If a pointer to a rearm event expression is provided, and the command option **MC_REG_OPTS_REARM_EVENT** is set (using the *options* parameter), then events will be generated when the rearm event expression evaluates to True. Otherwise, no events are generated when the rearm event expression is True.

event_cb
Identifies the callback routine that will be invoked by the RMC API when the event expression identified by the *expr* parameter evaluates to True. If the **MC_REG_OPTS_REARM_EVENT** option is set (using the *options* parameter), then the RMC API will also invoke this callback routine when the rearm event expression identified by the *raexpr* parameter evaluates to True.

The callback is invoked using a thread supplied to the RMC API by the application calling the **mc_dispatch** subroutine.

event_cb_arg
Specifies an argument that the RMC API will pass to the callback routine identified by the *event_cb* parameter.

OUTPUT

response
A pointer to a location in which the RMC API will return a pointer to the response. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_reg_class_event** subroutines can be used by the application to register a resource class event with the RMC subsystem. When an event occurs, the RMC subsystem sends an event notification to the application. Included in the event notification is the name of the resource class associated with the event. In order to get additional information about the resource class associated with the event, the application can, when calling these subroutines, use the *return_attrs* and *attr_count* parameters to specify that the RMC subsystem should return additional persistent attribute information.

This command cannot be used in an ordered command group.

If the resource class specified by the *rsrc_class_name* parameter is not available when the application calls the subroutine, the event registration will still succeed (as long as there are no errors). If the specified resource class becomes available, then the registration will be completed. Resource classes can become available when a node joins a cluster or a resource manager is started.

The response for these subroutines is a structure of type **mc_reg_rsp_t**, and is described in Response structure.

The event notification consists of a structure of type **mc_class_event_t**.

```
typedef struct mc_class_event              mc_class_event_t;
struct mc_class_event {
     mc_errnum_t                           mc_error;
     mc_event_flags_t                      mc_event_flags;
     struct timeval                        mc_timestamp;
     ct_char_t                             *mc_class_name;
     ct_char_t                             *mc_peer_domain_name;
```

```
    mc_attribute_t                  *mc_attrs;
    ct_uint32_t                     mc_attr_count;
    ct_uint32_t                     mc_e_attr_count;
};
```

The fields of this structure contain the following:

mc_error
: Indicates whether or not the event is an error event. If an error event, the error codes imply which of the remaining fields in the structure are defined. If not an error event, all of the remaining fields are defined.

mc_event_flags
: A bit field that describes the event using values defined by the **mc_event_flags_t** enumeration.

```
Event flags are defined by the mc_event_flags_t enumeration.
typedef enum mc_event_flags mc_event_flags_t;
enum mc_event_flags {
    MC_EVENT_RE_ARM = 0x0001,
    MC_EVENT_EXPR_FALSE = 0x0002,
    MC_EVENT_IMMED_EVAL = 0x0004,
    MC_EVENT_REFRESH = 0x0008,
    MC_EVENT_MISSING_PATTR = 0x0010,
    MC_EVENT_UNASSIGN = 0x0020,
    MC_EVENT_UNASSIGN_UNDEF = 0x0040,
    MC_EVENT_UNASSIGN_NO_MATCH = 0x0080,
    MC_EVENT_UNASSIGN_NO_GROUP = 0x0100,
    MC_EVENT_QUERY_EVENT = 0x0200,
    MC_EVENT_ASSIGN_RESOURCE = 0x0400,
    MC_EVENT_ASSIGN_NEW_RESOURCE = 0x0800
};
```

The **mc_event_flags_t** enumeration defines the following values:

MC_EVENT_RE_ARM
: The event was generated from the rearm expression.

MC_EVENT_EXPR_FALSE
: The expression evaluated to false. However, an event was generated for one of the following reasons:
  - The application requested immediate evaluation when the event was registered.
  - The resource variable was refreshed.
  - The event was queried using one of the **mc_query_event_*** subroutines.

MC_EVENT_IMMED_EVAL
: The event was generated as the result of an immediate evaluation.

MC_EVENT_REFRESH
: The event was generated as the result of a refresh of the resource variable. For example, monitoring of the variable resumed after a resource manager recovered from a failure.

MC_EVENT_MISSING_PATTR
: One or more requested persistent attributes could not be returned because they are not supported in the resource class identified in the *mc_class_name* field.

MC_EVENT_UNASSIGN
: The resource variable specified in the event has been unassigned from the event registration. The MC_EVENT_UNASSIGN_UNDEF, MC_EVENT_UNASSIGN_NO_MATCH, or MC_EVENT_UNASSIGN_NO_GROUP values indicate why the event has been unassigned.

MC_EVENT_UNASSIGN_UNDEF
: The associated resource has been undefined.

MC_EVENT_UNASSIGN_NO_MATCH
The persistent attributes of the associated resource no longer match the select string or the associated resource is located on a node that has been unconfigured.

MC_EVENT_UNASSIGN_NO_GROUP
The associated resource is located on a node that is no longer in the node group specified in the select string that was supplied to the event registration command.

MC_EVENT_QUERY_EVENT
The event was generated as the result of a query event command.

MC_EVENT_ASSIGN_RESOURCE
The associated resource has been assigned subsequent to the initial event registration. This flag is only present in the first event notification after the resource is assigned. Note that, unless the event registration specified immediate evaluation, some time may elapse between the time the resource was assigned to the registration and when the event notification was generated.

MC_EVENT_ASSIGN_NEW_RESOURCE
Indicates that the resource assigned subsequent to the initial event registration has recently been created. This flag is only present in the first event notification after the resource is assigned.

This field is undefined in any error event.

mc_timestamp
The time the event was generated.

mc_class_name
The name of the resource class, whose state change resulted in the generation of this event.

mc_peer_domain_name
The peer domain where the class event was generated. The name is a Null string if the response is from a node not in a peer domain.

mc_attrs
Specifies attributes using a pointer to an array of *mc_attr_count* elements of type *mc_attribute_t*.

mc_attr_count
The number of elements in the *mc_attrs* array.

mc_e_attr_count
The number of attributes in the *mc_attrs* array that were specified in the event expressions.

The attribute array specified by *mc_attrs* is considered a list, consisting of up to two sub-lists. The first sub-list consists of the attributes that are found in the event expressions. The second sub-list consists of the persistent attributes that are specified as return attributes to this subroutine. There are no duplicate attributes between the two sub-lists. If a persistent attribute is specified in an event expression and as a return attribute, it is only placed in the first sub-list.

The number of attributes in the first sub-list is given by *mc_e_attr_count*. The number of attributes in the second sub-list is given by the expression:

```
mc_attr_count - mc_e_attr_count
```

This expression is guaranteed to be greater than or equal to zero. The **MC_RETURN_ATTR_COUNT** macro can be used to calculate the number of return attributes.

Security

To register a resource class event with the RMC subsystem, the user of the calling application must have either **e** or **r** permission specified in an ACL entry for the resource class.

Return values

For the **mc_reg_class_event_bp** and **mc_reg_class_event_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and a response has been received and processed.

For the **mc_reg_class_event_ap** and **mc_reg_class_event_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_EORDERGROUP
An attempt was made to add the command to an ordered command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before the response could be received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before the response could be received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before the response could be received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_reg_rsp_t**. The response structure definition is:

```
typedef struct mc_reg_rsp              mc_reg_rsp_t;
struct mc_reg_rsp {
     mc_errnum_t                       mc_error;
     mc_registration_id_t              mc_registration_id;
};
```

The fields of this structure contain the following:

mc_error
If 0 (zero), this field indicates that the event has been successfully registered by the RMC subsystem. Any other value is an error. If this field indicates an error in the arguments supplied with the event registration command, the error code indicates which argument is in error.

mc_registration_id
If the *mc_error* field indicates the event has been registered successfully, this field contains the registration ID. The registration ID can be used by the application to unregister the event using the **mc_unreg_event_*** subroutines.

If the response indicates an error, the event has not been registered by the RMC subsystem and this field is undefined. Note that:

- a successful registration does not indicate that events will ever be generated.
- If events are generated in a multithreaded application, the event notification callback can be invoked before the Registration Response is processed by the application.

Location

/usr/lib/libct_mc.a

mc_reg_event_handle_*

This subroutine registers a resource event with the RMC subsystem using a resource handle.

Purpose

Registers a resource event with the RMC subsystem using a resource handle.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_reg_event_handle_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_reg_event_handle_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_reg_event_handle_bp(
          mc_sess_hndl_t             sess_hndl,
```

```
        mc_reg_rsp_t              **response,
        mc_reg_opts_t             options,
        ct_resource_handle_t      rsrc_hndl,
        ct_char_t                 **return_attrs,
        ct_uint32_t               attr_count,
        ct_char_t                 *expr,
        ct_char_t                 *raexpr,
        mc_event_cb_t             *event_cb,
        void                      *event_cb_arg)
```

The definition for the event notification callback is:

```
typedef void (mc_event_cb_t)(mc_sess_hndl_t,
mc_event_t *,
void *);
```

- The **mc_reg_event_handle_ap** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_reg_event_handle_ap(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_reg_rsp_t              **response,
          mc_reg_opts_t             options,
          ct_resource_handle_t      rsrc_hndl,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count,
          ct_char_t                 *expr,
          ct_char_t                 *raexpr,
          mc_event_cb_t             *event_cb,
          void                      *event_cb_arg)
```

The definition for the event notification callback is:

```
typedef void (mc_event_cb_t)(mc_sess_hndl_t,
mc_event_t *,
void *);
```

- The **mc_reg_event_handle_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_reg_event_handle_bc(
          mc_sess_hndl_t            sess_hndl,
          mc_reg_cb_t               *reg_cb,
          void                      *reg_cb_arg,
          mc_reg_opts_t             options,
          ct_resource_handle_t      rsrc_hndl,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count,
          ct_char_t                 *expr,
          ct_char_t                 *raexpr,
          mc_event_cb_t             *event_cb,
          void                      *event_cb_arg)
```

The definition for the response callback is:

```
typedef void (mc_reg_cb_t)(mc_sess_hndl_t,
mc_reg_rsp_t *,
void *);
```

The definition for the event notification callback is:

```
typedef void (mc_event_cb_t)(mc_sess_hndl_t,
mc_event_t *,
void *);
```

- The **mc_reg_event_handle_ac** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_reg_event_handle_ac(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_reg_cb_t               *reg_cb,
          void                      *reg_cb_arg,
          mc_reg_opts_t             options,
          ct_resource_handle_t      rsrc_hndl,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count,
          ct_char_t                 *expr,
          ct_char_t                 *raexpr,
          mc_event_cb_t             *event_cb,
          void                      *event_cb_arg)
```

  The definition for the response callback is:

```
typedef void (mc_reg_cb_t)(mc_sess_hndl_t,
mc_reg_rsp_t *,
void *);
```

  The definition for the event notification callback is:

```
typedef void (mc_event_cb_t)(mc_sess_hndl_t,
mc_event_t *,
void *);
```

Parameters

INPUT

sess_hndl
: The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
: The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

reg_cb
: Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

reg_cb_arg
: Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

options
: Specifies the value **MC_REG_OPTS_NONE** to indicate that there are no options or is the bitwise inclusive OR of one or more of the following options:

  MC_REG_OPTS_IMMED_EVAL
  : Generate an event at the first observation of the resource class specified in the event expression (identified by the *expr* parameter). Generate the event even if the event expression evaluates to FALSE. This option enables the application to obtain the value of the attribute when the event is registered.

MC_REG_OPTS_NO_REG
Do not register the event. Instead, merely evaluate the remaining arguments for syntactical correctness and return an appropriate response.

MC_REG_OPTS_NO_TOGGLE
If this option is set when a re-arm expression is specified, the re-arm expression is not a toggle. Rather, upon each evaluation, the primary expression is evaluated first. If the expression evaluates to True, an event is generated. If the expression evaluates to FALSE, the re-arm expression is then evaluated. If the re-arm expression is True, an event is generated. This evaluation procedure occurs if any of the attributes in either expression is reported by the resource manager to have a new value.

This option implicitly sets the MC_REG_OPTS_REARM_EVENT option. If a re-arm expression is not specified, the MC_REG_OPTS_NO_TOGGLE option is ignored.

The re-arm expression is considered an alternative expression to be evaluated; it does not "re-arm" a trigger that is specified by the primary expression.

MC_REG_OPTS_REARM_EVENT
Also trigger events when the rearm event expression (identified by the *raexpr* parameter) evaluates to True.

rsrc_hndl
The resource handle that identifies the resource for which the event is being registered. A resource handle is returned in the response structure for many RMC API subroutines including the **mc_define_resource_*** subroutine. An array of resource handles for resources of a particular resource class is returned in the response structure for the **mc_enumerate_resources_*** and **mc_enumerate_permitted_rsrcs_*** subroutines. To validate the resource handle before calling this subroutine, the application can call one of the **mc_validate_rsrc_hndl_*** subroutines.

If the specified resource is not currently available, the event registration will still succeed provided there are no other errors. When the specified resource becomes available, the event registration is extended to include the resource.

return_attrs
An array of *attr_count* pointers to persistent attribute names. This parameter, in conjunction with the *attr_count* parameter, enables the application to specify additional persistent attribute values to be included in the event notification.

If the *attr_count* is 0 (zero), *return_attrs* should be a Null pointer.

attr_count
Identifies the number of pointers to persistent attribute names in the *return_attrs* array. This parameter, in conjunction with the *return_attrs* parameter, enables the application to specify additional persistent attribute values to be included in the event notification.

expr A pointer to the event expression. In many cases, the event expression consists of an attribute name, a mathematical expression symbol, and a constant.

raexpr A pointer to a rearm event expression.

If no rearm event expression is needed for the event, this parameter can be a Null pointer.

If a pointer to a rearm event expression is provided, then RMC will stop evaluating the event expression once it evaluates to True, and instead will evaluate the rearm event expression until is True. Once the rearm event expression evaluates to True, the event is rearmed. In other words, RMC will once again evaluate the event expression identified by the *expr* parameter.

If a pointer to a rearm event expression is provided, and the command option **MC_REG_OPTS_REARM_EVENT** is set (using the *options* parameter), then events will be generated when the rearm event expression evaluates to True. Otherwise, no events are generated when the rearm event expression is True.

event_cb

Identifies the callback routine that will be invoked by the RMC API when the event expression identified by the *expr* parameter evaluates to True. If the **MC_REG_OPTS_REARM_EVENT** option is set (using the *options* parameter), then the RMC API will also invoke this callback routine when the rearm event expression identified by the *raexpr* parameter evaluates to True.

The callback is invoked using a thread supplied to the RMC API by the application calling the **mc_dispatch** subroutine.

event_cb_arg

Specifies an argument that the RMC API will pass to the callback routine identified by the *event_cb* parameter.

OUTPUT

response

A pointer to a location in which the RMC API will return a pointer to the response. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

Usage: These subroutines *cannot* be used in an ordered command group.

The **mc_reg_event_handle_*** subroutines can be used by the application to register a resource class event with the RMC subsystem. The resource is identified using a resource handle (as specified by the *rsrc_hndl* parameter). When an event occurs, the RMC subsystem sends an event notification to the application.

The primary node name of the node where the resource is being monitored, which is not necessarily the same node as where the resource is located, is included in the event notification. In order to get additional information about the resource associated with the event, the application can, when calling these subroutines, use the *return_attrs* and *attr_count* parameters to specify that the RMC subsystem should return additional persistent attribute information.

If the resource specified by the *rsrc_hndl* parameter is not available when the application calls the subroutine, the event registration will still succeed (as long as there are no errors). If the specified resource becomes available, the event registration will be automatically extended to include the resource. Resources can become available when a node joins a cluster or a resource manager is started.

The response for these subroutines is a structure of type **mc_reg_rsp_t**, and is described in Response structure.

The event notification consists of a structure of type **mc_event_t**.

```
typedef struct mc_event                 mc_event_t;
struct mc_event {
     mc_errnum_t                        mc_error;
     mc_event_flags_t                   mc_event_flags;
     struct timeval                     mc_timestamp;
     ct_resource_handle_t               mc_rsrc_hndl;
     mc_attribute_t                     *mc_attrs;
     ct_uint32_t                        mc_attr_count;
     ct_uint32_t                        mc_e_attr_count;
     ct_char_t                          *mc_node_name;
};
```

The fields of this structure contain the following:

mc_error

Indicates whether or not the event is an error event. If an error event, the error codes imply which of the remaining fields in the structure are defined. If not an error event, all of the remaining fields are defined.

mc_event_flags

A bit field that describes the event using values defined by the **mc_event_flags_t** enumeration.

```
Event flags are defined by the mc_event_flags_t enumeration.
typedef enum mc_event_flags mc_event_flags_t;
enum mc_event_flags {
     MC_EVENT_RE_ARM = 0x0001,
     MC_EVENT_EXPR_FALSE = 0x0002,
     MC_EVENT_IMMED_EVAL = 0x0004,
     MC_EVENT_REFRESH = 0x0008,
     MC_EVENT_MISSING_PATTR = 0x0010,
     MC_EVENT_UNASSIGN = 0x0020,
     MC_EVENT_UNASSIGN_UNDEF = 0x0040,
     MC_EVENT_UNASSIGN_NO_MATCH = 0x0080,
     MC_EVENT_UNASSIGN_NO_GROUP = 0x0100,
     MC_EVENT_QUERY_EVENT = 0x0200,
     MC_EVENT_ASSIGN_RESOURCE = 0x0400,
     MC_EVENT_ASSIGN_NEW_RESOURCE = 0x0800
};
```

The **mc_event_flags_t** enumeration defines the following values:

MC_EVENT_RE_ARM

The event was generated from the rearm expression.

MC_EVENT_EXPR_FALSE

The expression evaluated to false. However, and event was generated for one of the following reasons:

- The application requested immediate evaluation when the event was registered
- The resource variable was refreshed.
- The event was queried using one of the **mc_query_event_*** subroutines.

MC_EVENT_IMMED_EVAL

The event was generated as the result of an immediate evaluation.

MC_EVENT_REFRESH

The event was generated as the result of a refresh of the resource variable. For example, monitoring of the variable resumed after a resource manager recovered from a failure.

MC_EVENT_MISSING_PATTR

One or more requested persistent attributes could not be returned because they are not supported in the resource identified in the *mc_rsrc_hndl* field.

MC_EVENT_UNASSIGN

The resource variable specified in the event has been unassigned from the event registration. The MC_EVENT_UNASSIGN_UNDEF, MC_EVENT_UNASSIGN_NO_MATCH, or MC_EVENT_UNASSIGN_NO_GROUP values indicate why the event has been unassigned.

MC_EVENT_UNASSIGN_UNDEF

The associated resource has been undefined.

MC_EVENT_UNASSIGN_NO_MATCH

The persistent attributes of the associated resource no longer match the select string or the associated resource is located on a node that has been unconfigured.

MC_EVENT_UNASSIGN_NO_GROUP

The associated resource is located on a node that is no longer in the node group specified in the select string that was supplied to the event registration command.

MC_EVENT_QUERY_EVENT
The event was generated as the result of a query event command.

MC_EVENT_ASSIGN_RESOURCE
The associated resource has been assigned subsequent to the initial event registration. This flag is only present in the first event notification after the resource is assigned. Note that, unless the event registration specified immediate evaluation, some time may elapse between the time the resource was assigned to the registration and when the event notification was generated.

MC_EVENT_ASSIGN_NEW_RESOURCE
Indicates that the resource assigned subsequent to the initial event registration has recently been created. This is flag only present in the first event notification after the resource is assigned.

This field is undefined in any error event.

mc_timestamp
The time the event was generated.

mc_rsrc_hndl
The resource handle of the resource whose state change resulted in the generation of this event.

mc_attrs
Specifies attributes using a pointer to an array of *mc_attr_count* elements of type *mc_attribute_t*.

mc_attr_count
The number of elements in the *mc_attrs* array.

mc_e_attr_count
The number of attributes in the *mc_attrs* array that were specified in the event expressions.

mc_node_name
The primary node name of the node where the resource identified by the *mc_rsrc_hndl* field, is, (or was, in the case of certain error events) being monitored.

The attribute array specified by *mc_attrs* is considered a list, consisting of up to two sub-lists. The first sub-list consists of the attributes that are found in the event expressions. The second sub-list consists of the persistent attributes that are specified as return attributes to this subroutine. There are no duplicate attributes between the two sub-lists. If a persistent attribute is specified in an event expression and as a return attribute, it is only placed in the first sub-list.

The number of attributes in the first sub-list is given by *mc_e_attr_count*. The number of attributes in the second sub-list is given by the expression:

```
mc_attr_count - mc_e_attr_count
```

This expression is guaranteed to be greater than or equal to zero. The **MC_RETURN_ATTR_COUNT** macro can be used to calculate the number of return attributes.

Security

To register a resource event with the RMC subsystem, the user of the calling application must have either the **e** or **r** permission specified in an ACL entry for the resource.

Return values

For the **mc_reg_event_handle_bp** and **mc_reg_event_handle_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and a response has been received and processed.

For the **mc_reg_event_handle_ap** and **mc_reg_event_handle_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_EORDERGROUP
An attempt was made to add the command to an ordered command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before the response could be received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before the response could be received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before the response could be received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_reg_rsp_t**. The response structure definition is:

```
typedef struct mc_reg_rsp              mc_reg_rsp_t;
struct mc_reg_rsp {
     mc_errnum_t                       mc_error;
     mc_registration_id_t              mc_registration_id;
};
```

The fields of this structure contain the following:

mc_error
: If 0 (zero), this field indicates that the event has been successfully registered by the resource monitoring and control (RMC) subsystem. Any other value is an error. If this field indicates an error in the arguments supplied with the event registration command, the error code indicates which argument is in error.

mc_registration_id
: If the *mc_error* field indicates the event has been registered successfully, this field contains the registration ID. The registration ID can be used by the application to unregister the event using the **mc_unreg_event_*** subroutines.

  If the response indicates an error, the event has not been registered by the RMC subsystem and this field is undefined. Note that:

  - a successful registration does not indicate that events will ever be generated.
  - If events are generated in a multithreaded application, the event notification callback can be invoked before the Registration Response is processed by the application.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_free_response" on page 75
This subroutine frees a response or event notification structure.

mc_reg_event_select_*

This subroutine registers a resource event with the RMC subsystem using attribute selection.

Purpose

Registers a resource event with the RMC subsystem using attribute selection.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_reg_event_select_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_reg_event_select_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

  ```
  #include <rsct/ct_mc_v6.h>

  ct_int32_t
  ```

```
mc_reg_event_select_bp(
     mc_sess_hndl_t             sess_hndl,
     mc_reg_rsp_t               **response,
     mc_reg_opts_t              options,
     ct_char_t                  *rsrc_class_name,
     ct_char_t                  *select_attrs,
     ct_char_t                  **return_attrs,
     ct_uint32_t                attr_count,
     ct_char_t                  *expr,
     ct_char_t                  *raexpr,
     mc_event_cb_t              *event_cb,
     void                       *event_cb_arg)
```

The definition for the event notification callback is:

```
typedef void (mc_event_cb_t)(mc_sess_hndl_t,
mc_event_t *,
void *);
```

- The **mc_reg_event_select_ap** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_reg_event_select_ap(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_reg_rsp_t              **response,
          mc_reg_opts_t             options,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 *select_attrs,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count,
          ct_char_t                 *expr,
          ct_char_t                 *raexpr,
          mc_event_cb_t             *event_cb,
          void                      *event_cb_arg)
```

The definition for the event notification callback is:

```
typedef void (mc_event_cb_t)(mc_sess_hndl_t,
mc_event_t *,
void *);
```

- The **mc_reg_event_select_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_reg_event_select_bc(
          mc_sess_hndl_t            sess_hndl,
          mc_reg_cb_t               *reg_cb,
          void                      *reg_cb_arg,
          mc_reg_opts_t             options,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 *select_attrs,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count,
          ct_char_t                 *expr,
          ct_char_t                 *raexpr,
          mc_event_cb_t             *event_cb,
          void                      *event_cb_arg)
```

The definition for the response callback is:

```
typedef void (mc_reg_cb_t)(mc_sess_hndl_t,
mc_reg_rsp_t *,
void *);
```

The definition for the event notification callback is:

```
typedef void (mc_event_cb_t)(mc_sess_hndl_t,
mc_event_t *,
void *);
```

- The **mc_reg_event_select_ac** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_reg_event_select_ac(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_reg_cb_t               *reg_cb,
          void                      *reg_cb_arg,
          mc_reg_opts_t             options,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 *select_attrs,
          ct_char_t                 **return_attrs,
          ct_uint32_t               attr_count,
          ct_char_t                 *expr,
          ct_char_t                 *raexpr,
          mc_event_cb_t             *event_cb,
          void                      *event_cb_arg)
```

  The definition for the response callback is:

```
typedef void (mc_reg_cb_t)(mc_sess_hndl_t,
mc_reg_rsp_t *,
void *);
```

  The definition for the event notification callback is:

```
typedef void (mc_event_cb_t)(mc_sess_hndl_t,
mc_event_t *,
void *);
```

Parameters

INPUT

sess_hndl
: The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
: The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

reg_cb
: Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

reg_cb_arg
: Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

options
: Specifies the value **MC_REG_OPTS_NONE** to indicate that there are no options or is the bitwise inclusive OR of one or more of the following options:

  MC_REG_OPTS_IMMED_EVAL
  : Generate an event at the first observation of the resource class specified in the event

expression (identified by the *expr* parameter). Generate the event even if the event expression evaluates to FALSE. This option enables the application to obtain the value of the attribute when the event is registered.

MC_REG_OPTS_KEEP_REG
Maintain event registration even if a persistent attributes of resource change such that the resource would no longer match the selection string expression specified by the *select_attrs* parameter. If this option is not specified, such a resource would no longer be assigned to the event registration and an error would be generated.

MC_REG_OPTS_NO_REG
Do not register the event. Instead, merely evaluate the remaining arguments for syntactical correctness and return an appropriate response.

MC_REG_OPTS_NO_TOGGLE
If this option is set when a re-arm expression is specified, the re-arm expression is not a toggle. Rather, upon each evaluation, the primary expression is evaluated first. If the expression evaluates to True, an event is generated. If the expression evaluates to FALSE, the re-arm expression is then evaluated. If the re-arm expression is True, an event is generated. This evaluation procedure occurs if any of the attributes in either expression is reported by the resource manager to have a new value.

This option implicitly sets the MC_REG_OPTS_REARM_EVENT option. If a re-arm expression is not specified, the MC_REG_OPTS_NO_TOGGLE option is ignored.

The re-arm expression is considered an alternative expression to be evaluated; it does not "re-arm" a trigger that is specified by the primary expression.

MC_REG_OPTS_REARM_EVENT
Also trigger events when the rearm event expression (identified by the *raexpr* parameter) evaluates to True.

rsrc_class_name
Pointer to a resource class name. Identifies the resource class of the resource(s) for which the event is being registered.

select_attrs
A selection string expression that identifies one or more resources of the resource class identified by the *rsrc_class_name* parameter. Resources of the resource class that match the selection string expression will be assigned to the event registration.

return_attrs
An array of *attr_count* pointers to persistent attribute names. This parameter, in conjunction with the *attr_count* parameter, enables the application to specify additional persistent attribute values to be included in the event notification.

If the *attr_count* is 0 (zero), *return_attrs* should be a Null pointer.

attr_count
Identifies the number of pointers to persistent attribute names in the *return_attrs* array. This parameter, in conjunction with the *return_attrs* parameter, enables the application to specify additional persistent attribute values to be included in the event notification.

expr A pointer to the event expression. In many cases, the event expression consists of an attribute name, a mathematical expression symbol, and a constant.

raexpr A pointer to a rearm event expression.

If no rearm event expression is needed for the event, then this parameter can be a Null pointer.

If a pointer to a rearm event expression is provided, then RMC will stop evaluating the event expression once it evaluates to True, and instead will evaluate the rearm event expression until is

True. Once the rearm event expression evaluates to True, the event is rearmed. In other words, RMC will once again evaluate the event expression identified by the *expr* parameter.

If a pointer to a rearm event expression is provided, and the command option **MC_REG_OPTS_REARM_EVENT** is set (using the *options* parameter), then events will be generated when the rearm event expression evaluates to True. Otherwise, no events are generated when the rearm event expression is True.

event_cb
Identifies the callback routine that will be invoked by the RMC API when the event expression identified by the *expr* parameter evaluates to True. If the **MC_REG_OPTS_REARM_EVENT** option is set (using the *options* parameter), then the RMC API will also invoke this callback routine when the rearm event expression identified by the *raexpr* parameter evaluates to True.

The callback is invoked using a thread supplied to the RMC API by the application calling the **mc_dispatch** subroutine.

event_cb_arg
Specifies an argument that the RMC API will pass to the callback routine identified by the *event_cb* parameter.

OUTPUT

response
A pointer to a location in which the RMC API will return a pointer to the response. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

Usage: These subroutines *cannot* be used in an ordered command group.

The **mc_reg_event_select_*** subroutines can be used by the application to register a resource event with the RMC subsystem. Resources of the resource class (identified by the *rsrc_class_name* parameter) that match the selection string (specified by the *select_attrs* parameter) are assigned to the event registration.

The primary node name of the node where the resource is being monitored, which is not necessarily the same node as where the resource is located, is included in the event notification. In order to get additional information about the resource associated with the event, the application can, when calling these subroutines, use the *return_attrs* and *attr_count* parameters to specify that the RMC subsystem should return additional persistent attribute information.

Even if no resource that matches the *select_attrs* selection string is found when the application calls the subroutine, the event registration will still succeed (as long as there are no other errors). If a resource becomes available that matches the *select_attrs* selection string, the event registration will be automatically extended to include the resource. Resources can become available when a node joins a cluster, a resource manager is started, or a resource manager defines new resources.

The response for these subroutines is a structure of type **mc_reg_rsp_t**, and is described in Response structure.

The event notification consists of a structure of type **mc_event_t**.

```
typedef struct mc_event                 mc_event_t;
struct mc_event {
     mc_errnum_t                        mc_error;
     mc_event_flags_t                   mc_event_flags;
     struct timeval                     mc_timestamp;
     ct_resource_handle_t               mc_rsrc_hndl;
     mc_attribute_t                     *mc_attrs;
```

```
    ct_uint32_t                     mc_attr_count;
    ct_uint32_t                     mc_e_attr_count;
    ct_char_t                       *mc_node_name;
};
```

The fields of this structure contain the following:

mc_error
: Indicates whether or not the event is an error event. If an error event, the error codes imply which of the remaining fields in the structure are defined. If not an error event, all of the remaining fields are defined.

mc_event_flags
: A bit field that describes the event using values defined by the **mc_event_flags_t** enumeration.

```
Event flags are defined by the mc_event_flags_t enumeration.
typedef enum mc_event_flags mc_event_flags_t;
enum mc_event_flags {
    MC_EVENT_RE_ARM = 0x0001,
    MC_EVENT_EXPR_FALSE = 0x0002,
    MC_EVENT_IMMED_EVAL = 0x0004,
    MC_EVENT_REFRESH = 0x0008,
    MC_EVENT_MISSING_PATTR = 0x0010,
    MC_EVENT_UNASSIGN = 0x0020,
    MC_EVENT_UNASSIGN_UNDEF = 0x0040,
    MC_EVENT_UNASSIGN_NO_MATCH = 0x0080,
    MC_EVENT_UNASSIGN_NO_GROUP = 0x0100,
    MC_EVENT_QUERY_EVENT = 0x0200,
    MC_EVENT_ASSIGN_RESOURCE = 0x0400,
    MC_EVENT_ASSIGN_NEW_RESOURCE = 0x0800
};
```

The **mc_event_flags_t** enumeration defines the following values:

MC_EVENT_RE_ARM
: The event was generated from the rearm expression.

MC_EVENT_EXPR_FALSE
: The expression evaluated to false. However, and event was generated for one of the following reasons:
  - The application requested immediate evaluation when the event was registered
  - The resource variable was refreshed.
  - The event was queried using one of the **mc_query_event_*** subroutines.

MC_EVENT_IMMED_EVAL
: The event was generated as the result of an immediate evaluation.

MC_EVENT_REFRESH
: The event was generated as the result of a refresh of the resource variable. For example, monitoring of the variable resumed after a resource manager recovered from a failure.

MC_EVENT_MISSING_PATTR
: One or more requested persistent attributes could not be returned because they are not supported in the resource identified in the *mc_rsrc_hndl* field.

MC_EVENT_UNASSIGN
: The resource variable specified in the event has been unassigned from the event registration. The MC_EVENT_UNASSIGN_UNDEF, MC_EVENT_UNASSIGN_NO_MATCH, or MC_EVENT_UNASSIGN_NO_GROUP values indicate why the event has been unassigned.

MC_EVENT_UNASSIGN_UNDEF
: The associated resource has been undefined.

MC_EVENT_UNASSIGN_NO_MATCH
The persistent attributes of the associated resource no longer match the select string or the associated resource is located on a node that has been unconfigured.

MC_EVENT_UNASSIGN_NO_GROUP
The associated resource is located on a node that is no longer in the node group specified in the select string that was supplied to the event registration command.

MC_EVENT_QUERY_EVENT
The event was generated as the result of a query event command.

MC_EVENT_ASSIGN_RESOURCE
The associated resource has been assigned subsequent to the initial event registration. This flag is only present in the first event notification after the resource is assigned. Note that, unless the event registration specified immediate evaluation, some time may elapse between the time the resource was assigned to the registration and when the event notification was generated.

MC_EVENT_ASSIGN_NEW_RESOURCE
Indicates that the resource assigned subsequent to the initial event registration has recently been created. This flag is only present in the first event notification after the resource is assigned.

This field is undefined in any error event.

mc_timestamp
The time the event was generated.

mc_rsrc_hndl
The resource handle of the resource whose state change resulted in the generation of this event.

mc_attrs
Specifies attributes using a pointer to an array of *mc_attr_count* elements of type *mc_attribute_t*.

mc_attr_count
The number of elements in the *mc_attrs* array.

mc_e_attr_count
The number of attributes in the *mc_attrs* array that were specified in the event expressions.

mc_node_name
The primary node name of the node where the resource identified by the *mc_rsrc_hndl* field, is, (or was, in the case of certain error events) being monitored.

The attribute array specified by *mc_attrs* is considered a list, consisting of up to two sub-lists. The first sub-list consists of the attributes that are found in the event expressions. The second sub-list consists of the persistent attributes that are specified as return attributes to this subroutine. There are no duplicate attributes between the two sub-lists. If a persistent attribute is specified in an event expression and as a return attribute, it is only placed in the first sub-list.

The number of attributes in the first sub-list is given by *mc_e_attr_count*. The number of attributes in the second sub-list is given by the expression:

```
mc_attr_count - mc_e_attr_count
```

This expression is guaranteed to be greater than or equal to zero. The **MC_RETURN_ATTR_COUNT** macro can be used to calculate the number of return attributes.

Security

To register a resource event with the RMC subsystem, the user of the calling application must have either the **e** or **r** permission specified in an ACL entry for the specified resource(s)

Return values

For the **mc_reg_event_select_bp** and **mc_reg_event_select_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and a response has been received and processed.

For the **mc_reg_event_select_ap** and **mc_reg_event_select_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_EORDERGROUP
An attempt was made to add the command to an ordered command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before the response could be received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before the response could be received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before the response could be received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_reg_rsp_t**. The response structure definition is:

```
typedef struct mc_reg_rsp              mc_reg_rsp_t;
struct mc_reg_rsp {
     mc_errnum_t                       mc_error;
     mc_registration_id_t              mc_registration_id;
};
```

The fields of this structure contain the following:

mc_error
If 0 (zero), this field indicates that the event has been successfully registered by the resource monitoring and control (RMC) subsystem. Any other value is an error. If this field indicates an error in the arguments supplied with the event registration command, the error code indicates which argument is in error.

mc_registration_id
If the *mc_error* field indicates the event has been registered successfully, this field contains the registration ID. The registration ID can be used by the application to unregister the event using the **mc_unreg_event_*** subroutines.

If the response indicates an error, the event has not been registered by the RMC subsystem and this field is undefined. Note that:

- a successful registration does not indicate that events will ever be generated.
- If events are generated in a multithreaded application, the event notification callback can be invoked before the Registration Response is processed by the application.

Location

/usr/lib/libct_mc.a

mc_reset_*

This subroutine requests the RMC subsystem to force a resource offline.

Purpose

Requests the RMC subsystem to force a resource offline.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_reset_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_reset_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_reset_bp(
```

```
        mc_sess_hndl_t          sess_hndl,
        mc_rsrc_hndl_rsp_t      **response,
        ct_resource_handle_t    rsrc_hndl,
        ct_structured_data_t    *data)
```

- The **mc_reset_ap** subroutine adds the command to a command group. If used in an ordered command group, however, please note that all other commands in the command group must also be the form that uses a resource handle to specify a command target. To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_reset_ap(
          mc_cmdgrp_hndl_t        cmdgrp_hndl,
          mc_rsrc_hndl_rsp_t      **response,
          ct_resource_handle_t    rsrc_hndl,
          ct_structured_data_t    *data)
```

- The **mc_reset_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_reset_bc(
          mc_sess_hndl_t          sess_hndl,
          mc_reset_cb_t           *reset_cb,
          void                    *reset_cb_arg,
          ct_resource_handle_t    rsrc_hndl,
          ct_structured_data_t    *data)
```

  The definition for the response callback is:

```
typedef void (mc_reset_cb_t)(mc_sess_hndl_t,
mc_rsrc_hndl_rsp_t *,
void *);
```

- The **mc_reset_ac** subroutine adds the command to a command group. If used in an ordered command group, however, please note that all other commands in the command group must also be the form that uses a resource handle to specify a command target. To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_reset_ac(
          mc_cmdgrp_hndl_t        cmdgrp_hndl,
          mc_reset_cb_t           *reset_cb,
          void                    *reset_cb_arg,
          ct_resource_handle_t    rsrc_hndl,
          ct_structured_data_t    *data)
```

  The definition for the response callback is:

```
typedef void (mc_reset_cb_t)(mc_sess_hndl_t,
mc_rsrc_hndl_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
: The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
: The command group handle that identifies the command group to which this command should

be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

reset_cb
Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

reset_cb_arg
Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

rsrc_hndl
The resource handle that identifies the resource to be forced offline. A resource handle is returned in the response structure for many RMC API subroutines including the **mc_define_resource_*** subroutine. An array of resource handles for resources of a particular resource class is returned in the response structure for the **mc_enumerate_resources_*** and **mc_enumerate_permitted_rsrcs_*** subroutines. To validate the resource handle before calling this subroutine, the application can call one of the **mc_validate_rsrc_hndl_*** subroutines.

data A pointer to structured data containing resource class specific options for taking the resource offline. To accept the default values (or if the resource class does not define options) for going offline, the data parameter should be a NULL pointer.

To obtain the syntax and semantics for the structured data required by the resource class for specifying offline options, the application can use the **mc_qdef_sd_*** subroutine.

OUTPUT

response
A pointer to a location in which the RMC API will return a pointer to the response. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_reset_*** subroutines can be used by the application to send a request to the RMC subsystem to force a resource (identified by the *rsrc_hndl* parameter) offline. The request is performed by the resource's associated resource manager. If the resource manager accepts structured data as options for taking a resource offline, the application can provide this using the *data* parameter.

The **mc_rest_*** subroutines are more forceful versions of the **mc_offline_*** subroutines, in that the resource manager must ensure that the resource goes offline.

The response for these subroutines is a structure of type **mc_rsrc_hndl_rsp_t**, and is described in Response structure.

If this command is used in an ordered command group, please note that all other commands in the command group must be the form that uses a resource handle to specify the command target. The same resource handle must be used on all commands in the command group.

Security

To take a resource offline, the user of the calling application must have either **o** or **w** permission specified in an ACL entry for this resource.

Return values

For the **mc_reset_bp** and **mc_reset_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and a response has been received and processed.

For the **mc_reset_ap** and **mc_reset_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_ETARGETMISMATCH
The target specified for the command does not match the target of the command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before the response could be received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before the response could be received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_EINVALIDDATATYPE
Invalid attribute data type specified.

MC_EINVALIDVALUEPTR
Invalid attribute value pointer specified.

MC_EINVALIDSDTYPE
Invalid structured data subtype specified.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before the response could be received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_rsrc_hndl_rsp_t**. The **mc_error** field of the **mc_rsrc_hndl_rsp_t** structure indicates whether or not the resource manager has successfully processed the command. If does not mean that the resource is offline. To determine if the resource is actually offline, the application must register an event to monitor the resource's OpState attribute value. This command results in only one response.

The response structure definition is:

```
typedef struct mc_rsrc_hndl_rsp         mc_rsrc_hndl_rsp_t;
struct mc_rsrc_hndl_rsp {
     mc_errnum_t                        mc_error;
     ct_resource_handle_t               mc_rsrc_hndl;
};
```

The fields of this structure contain the following:

mc_error
If 0 (zero), this field indicates that the command was successful. Any other value is an error. If there is an error, the error codes indicate whether the resource handle contained in *mc_rsrc_hndl* is invalid or the command could not be completed for the resource specified by the resource handle. The error may also indicate that the command arguments were in error.

mc_rsrc_hndl
The resource handle that identifies the resource that was the target of the command.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_offline_*" on page 88
This subroutine sends a request to the RMC subsystem to take a resource offline.

mc_send_cmd_grp

This subroutine sends a command group to the RMC subsystem.

Purpose

Sends a command group to the RMC subsystem.

Library

RMC Library (**libct_mc.a**)

Syntax

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_send_cmd_grp(
          mc_cmdgrp_hndl_t            cmd_hndl,
          mc_complete_cb_t            *complete_cb,
          void                        *cb_arg)
```

The definition for the response callback is:

```
typedef void (mc_complete_cb_t)(mc_sess_hndl_t, ct_int32_t, void *);
```

Parameters

INPUT

cmd_hndl
: The command group handle that identifies the command group to send to the RMC subsystem. A command group handle is returned by the **mc_start_cmd_group** subroutine when the application allocates a command group. The command group is sent for the RMC session specified on the call to the **mc_start_cmd_group** subroutine.

complete_cb
: A pointer to the completion callback routine. This callback will be invoked by the RMC API after all response callbacks have been returned, or all pointers have been placed in their specified locations for the commands in the command group.

cb_arg
: Identifies an argument that the RMC API will pass to the callback routine identified by the *complete_cb* parameter.

Description

The **mc_send_cmd_grp** subroutine can be used by the application to send a command group (identified by the *cmd_hndl* parameter) to the RMC subsystem. The application should have, prior to calling this subroutine, added one or more commands to the command group.

Provided the subroutine does not detect any errors, it returns immediately after sending the command group to the RMC subsystem. Unlike the **mc_send_cmd_grp_wait** subroutine, this subroutine will not cause the calling application to block in the RMC API.

Table 22. mc_send_cmd_grp subroutine processing alternatives depending on the response method

If the command in the command group uses the:	Then responses to the commands in the command group are processed by:
pointer response method	returning a pointer to the response in the location specified by the command.
callback response method	invoking the associated callback using a thread supplied by a call to the **mc_dispatch** subroutine. When there is more than one response ready to be processed using callback response, the RMC API will parallelize callback invocation if there are threads available.

After all response callbacks have been returned, or all pointers have been placed in their specified locations, for the commands in the command group, the RMC API will invoke the completion callback (identified by the *complete_cb* parameter). Like the individual response callbacks, the RMC API invokes the completion callback using a thread supplied by a call to the **mc_dispatch** subroutine. The RMC API passes the completion callback three arguments: a session handle, an error value, and the *cb_arg* argument. The error value is either 0 (zero), indicating no error, or one of the following values:

MC_ESENTENDED
The command has been sent but the session ended before all responses could be received. If pointer response was selected for any commands in the command group, check the appropriate pointer or array count, if defined, for any responses. If callback response was selected for any commands in the command group, the callbacks were invoked for any responses received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before all responses could be received. If pointer response was selected for any commands in the command group, check the appropriate pointer or array count, if defined, for any responses. If callback response was selected for any commands in the command group, the callbacks were invoked for any responses received.

If the error value is non-zero, the application can call the **cu_get_error** function from within the callback for additional error information.

The RMC API guarantees that all commands in an ordered command group, with respect to each specified resource, are processed in the order in which they were added to the command group. In addition, all responses to commands in a command group, ordered or not, from the same resource or resource class are processed serially.

Security

Authorization checks are performed for each command in the command group (if authorization checks are necessary) as each command is processed by the RMC subsystem. See the reference information for each command for information on any required authorization level.

Return values

A return value of 0 (zero) indicates that the command group has been successfully sent to the RMC subsystem. Any other return value is an error and indicates that the command group was not sent.

The following errors can be returned by this subroutine. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_EINVALIDCB
Invalid callback specified.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ENOCMDS
The command group contains no commands.

MC_ENOMEM
The API could not allocate required memory.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_cancel_cmd_grp" on page 41
This subroutine cancels a command group.

mc_send_cmd_grp_wait

This subroutine sends a command group to the RMC subsystem and waits for completion.

Purpose

Sends a command group to the RMC subsystem and waits for completion.

Library

RMC Library (**libct_mc.a**)

Syntax

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_send_cmd_grp_wait(
          mc_cmdgrp_hndl_t              cmd_hndl)
```

Parameters

INPUT

cmd_hndl
: The command group handle that identifies the command group to send to the RMC subsystem. A command group handle is returned by the **mc_start_cmd_group** subroutine when the application allocates a command group. The command group is sent for the RMC session specified on the call to the **mc_start_cmd_grooup** subroutine.

Description

The **mc_send_cmd_grp_wait** subroutine can be used by the application to send a command group (identified by the *cmd_hndl* parameter) to the RMC API. The application should have, prior to calling this subroutine, added one or more commands to the command group. Provided this subroutine does not detect any errors, the application thread will block in the RMC API until responses have been received and processed for each command in the command group.

Table 23. mc_send_cmd_grp_wait subroutine processing alternatives depending on the response method

If the command in the command group uses the:	Then responses to the commands in the command group are processed by:
pointer response method	returning a pointer to the response in the location specified by the command.
callback response method	invoking the associated callback using a thread supplied by a call to the **mc_dispatch** subroutine. When there is more than one response ready to be processed using callback response, the RMC API will parallelize callback invocation if there are threads available.

The RMC API guarantees that all commands in an ordered command group, with respect to each specified resource, are processed in the order in which they were added to the command group. In

addition, all responses to commands in a command group, ordered or not, from the same resource or resource class are processed serially.

Security

Authorization checks are performed for each command in the command group (if authorization checks are necessary) as each command is processed by the RMC subsystem. See the reference information for each command for information on any required authorization level.

Return values

A return value of 0 (zero) indicates that the command group has been successfully sent to the RMC subsystem. Any other return value is an error and indicates that the command group was not sent.

The following errors can be returned by this subroutine. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_EDEADLOCK
The subroutine was invoked from within an event notification callback and the command group contains an unregister event command.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command group has been sent but the session ended before all responses could be received. If pointer response was selected for any commands in the command group, check the appropriate pointer or array count, if defined, for any responses. If callback response was selected for any commands in the command group, the callbacks were invoked for any responses received.

MC_ESENTINTRPT
The command group has been sent but the session was interrupted before all responses could be received. If pointer response was selected for any commands in the command group, check the appropriate pointer or array count, if defined, for any responses. If callback response was selected for any commands in the command group, the callbacks were invoked for any responses received.

MC_ENOCMDS
The command group contains no commands.

MC_ETIMEDOUT
The command group has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before all responses could be received. If subroutine uses pointer response, check the appropriate pointer or array count, if defined, for any responses. If a subroutine uses callback response, the callback will have been invoked for any responses received.

MC_ENOMEM
The API could not allocate required memory.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_cancel_cmd_grp" on page 41
This subroutine cancels a command group.

mc_session_info

This subroutine gets information about a session.

Purpose

Gets information about a session.

Library

RMC Library (**libct_mc.a**)

Syntax

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
mc_session_info(
               mc_sess_hndl_t    session_hndl,
               mc_sess_info_t    info_type,
               void                          *info_return
)
```

Parameters

INPUT

session_hndl
: Specifies the handle of the session for which information is to be obtained. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

info_type
: Specifies the type of information to be returned: **MC_SESS_INFO_LOCAL_IP_ADDRESS**.

OUTPUT

info_return
: Is a pointer to a location in memory that is large enough to hold the requested information. For **MC_SESS_INFO_LOCAL_IP_ADDRESS**, this parameter must point to an **in6_addr** structure as defined in **netinet/in.h**.

Description

The **mc_session_info** subroutine is used by a cluster application to obtain information about a session with the RMC subsystem. This subroutine is supported for MC_VERSION values 2 and greater.

Return values

If this subroutine returns a value of **0**, the requested information has been returned in the specified memory location. If this subroutine returns a non-zero value, it is an error value; no information is returned.

The following errors can be returned by this subroutine. Additional error information can be returned by calling the **cu_get_error** function.

MC_EIMPROPERINFO
The requested information is not defined for the specified session.

MC_EINVALIDSESS
The specified session handle is not valid.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_EUNATTAINABLEINFO
The requested information cannot be obtained.

MC_EUNKNOWNINFO
The *info_type* parameter specifies an unknown value.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_end_session" on page 64
This subroutine ends a session with the RMC subsystem.

"mc_timed_start_session" on page 212
This subroutine establishes a session with the RMC subsystem.

mc_set_handle_*

This subroutine sets persistent attribute values of a resource identified by a resource handle.

Purpose

Sets persistent attribute values of a resource identified by a resource handle.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_set_handle_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_set_handle_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_set_handle_bp(
          mc_sess_hndl_t            sess_hndl,
```

```
        mc_set_rsp_t              **response,
        ct_resource_handle_t      rsrc_hndl,
        mc_attribute_t            *attrs,
        ct_uint32_t               count)
```

- The **mc_set_handle_ap** subroutine adds the command to a command group. If used in an ordered command group, however, note that all other commands in the command group must also be the form that uses a resource handle to specify the command target. The same resource handle must be used on all commands in the command group.

  To receive responses, This subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_set_handle_ap(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_set_rsp_t              **response,
          ct_resource_handle_t      rsrc_hndl,
          mc_attribute_t            *attrs,
          ct_uint32_t               count)
```

- The **mc_set_handle_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_set_handle_bc(
          mc_sess_hndl_t            sess_hndl,
          mc_set_cb_t               *set_cb,
          void                      *set_cb_arg,
          ct_resource_handle_t      rsrc_hndl,
          mc_attribute_t            *attrs,
          ct_uint32_t               count)
```

  The definition for the response callback is:

```
typedef void (mc_set_cb_t)(mc_sess_hndl_t,
mc_set_rsp_t *,
void *);
```

- The **mc_set_handle_ac** subroutine adds the command to a command group. If used in an ordered command group, however, please note that all other commands in the command group must also be the form that uses a resource handle to specify the command target.

  To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_set_handle_ac(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_set_cb_t               *set_cb,
          void                      *set_cb_arg,
          ct_resource_handle_t      rsrc_hndl,
          mc_attribute_t            *attrs,
          ct_uint32_t               count)
```

  The definition for the response callback is:

```
typedef void (mc_set_cb_t)(mc_sess_hndl_t,
mc_set_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

set_cb Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

set_cb_arg
Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

rsrc_hndl
The resource handle that identifies the resource whose persistent attributes are to be set. A resource handle is returned in the response structure for many RMC API subroutines including the **mc_define_resource_*** subroutine. An array of resource handles for resources of a particular resource class is returned in the response structure for the **mc_enumerate_resources_*** and **mc_enumerate_permitted_rsrcs_*** subroutines. To validate the resource handle before calling this subroutine, the application can call one of the **mc_validate_rsrc_hndl_*** subroutines.

attrs Specifies the persistent attribute values to be set for the resource identified by the *rsrc_hndl* parameter. Specifies this using an array of *count* elements of type **mc_attribute_t**. Each element in the array identifies a persistent attribute of the resource and a value. Persistent attribute of the resource that are not specified in this array remain unchanged.

count Specifies the number of elements in the *attrs* array.

OUTPUT

response
A pointer to a location in which the RMC API will return a pointer to the response. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_set_handle_*** subroutines can be used by the application to set persistent attribute values of a resource. The resource is identified using a resource handle (as specified by the *rsrc_hndl* parameter).

The response for these subroutines is a structure of type **mc_set_rsp_t**, and is described in Response structure.

If the command is used in an ordered command group, please note that all other commands in the command group must be the form that uses a resource handle to specify the command target. The same resource handle must be used on all commands in the command group.

Security

To set persistent attribute values of a resource, the user of the calling application must have either **s** or **w** permission specified in an ACL entry for the resource.

Return values

For the **mc_set_handle_bp** and **mc_set_handle_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and a response has been received and processed.

For the **mc_set_handle_ap** and **mc_set_handle_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_ETARGETMISMATCH
The target specified for the command does not match the target of the command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before the response could be received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before the response could be received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_EINVALIDDATATYPE
Invalid attribute data type specified.

MC_EINVALIDVALUEPTR
Invalid attribute value pointer specified.

MC_EINVALIDSDTYPE
Invalid structured data subtype specified.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before the response could be received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_set_rsp_t**. If any of the set attribute arguments are invalid, then the response returned indicates the error and no attributes are set.

The response structure definition is:

```
typedef struct mc_set_rsp              mc_set_rsp_t;
struct mc_set_rsp {
     mc_errnum_t                       mc_error;
     ct_resource_handle_t              mc_rsrc_hndl;
     mc_error_attr_t                   *mc_error_attrs;
     ct_uint32_t                       mc_attr_count;
};
```

The fields of this structure contain the following:

mc_error
If 0 (zero), this field indicates that the command was successful, and, if the *mc_attr_count* is 0 (zero), all the specified attributes were set. If the *mc_attr_count* field is not 0 (zero), the *mc_error_attrs* field contains a pointer to an array of attributes that could not be set. Attributes not included in the array have been set. If the *mc_error* field is not 0 (zero), the value indicates the error.

mc_rsrc_hndl
If the command is successful, this field specifies the resource handle of the resource that was set. If an error is indicated by the *mc_error* field, this field contains the resource handle that was specified on the command.

mc_error_attrs
If no error is indicated by the *mc_error* field, and only some of the attributes could be set, this field is an array of *mc_attr_count* elements of type **mc_error_attr_t**.

```
typedef struct mc_error_attr       mc_error_attr_t;
struct mc_error_attr {
     mc_errnum_t                   mc_error;
     ct_char_t                     *mc_at_name;
};
```

Each element in the array identifies an attribute that could not be set. The fields of this structure contain the following:

mc_error
Indicates the reason the attribute could not be set.

mc_at_name
Indicates the name of the attribute that could not be set.

mc_attr_count
Indicates the number of entries in the *mc_error_attrs* array.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_class_set_*" on page 53
This subroutine sets persistent attribute values of a resource class.

mc_set_select_*

This subroutine sets persistent attribute values of one or more resources of a particular resource class. The resources are identified by attribute selection.

Purpose

Sets persistent attribute values of one or more resources of a particular resource class. The resources are identified by attribute selection.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_set_select_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_set_select_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_set_select_bp(
          mc_sess_hndl_t            sess_hndl,
          mc_set_rsp_t              **rsp_array,
          ct_uint32_t               *array_cnt,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 *select_attrs,
          mc_attribute_t            *attrs,
          ct_uint32_t               count)
```

- The **mc_set_select_ap** subroutine adds the command to a command group. If used in an ordered command group, however, please note that all other commands in the command group must also be the form that uses attribute selection to specify the command target. The values of the *select_attrs* and *rsrc_class_name* parameters must be identical on all commands in the command group.

  To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_set_select_ap(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_set_rsp_t              **rsp_array,
          ct_uint32_t               *array_cnt,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 *select_attrs,
          mc_attribute_t            *attrs,
          ct_uint32_t               count)
```

- The **mc_set_select_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_set_select_bc(
          mc_sess_hndl_t            sess_hndl,
          mc_set_cb_t               *set_cb,
          void                      *set_cb_arg,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 *select_attrs,
          mc_attribute_t            *attrs,
          ct_uint32_t               count)
```

The definition for the response callback is:

```
typedef void (mc_set_cb_t)(mc_sess_hndl_t,
mc_set_rsp_t *,
void *);
```

- The **mc_set_select_ac** subroutine adds the command to a command group. The values of the *select_attrs* and *rsrc_class_name* parameters must be identical on all commands in the command group.

  To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_set_select_ac(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_set_cb_t               *set_cb,
          void                      *set_cb_arg,
          ct_char_t                 *rsrc_class_name,
          ct_char_t                 *select_attrs,
          mc_attribute_t            *attrs,
          ct_uint32_t               count)
```

  The definition for the response callback is:

```
typedef void (mc_set_cb_t)(mc_sess_hndl_t,
mc_set_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
: The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
: The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

set_cb
: Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

set_cb_arg
: Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

rsrc_class_name
: Pointer to a resource class name. Identifies the resource class of the resource(s) whose persistent attribute values are being set.

select_attrs
A selection string that identifies one or more resources of the resource class identified by the *rsrc_class_name* parameter. Resources of the resource class that match the selection string expression will have the persistent attributes (identified by the *attrs* array) set.

If this parameter is a NULL pointer, then all resources of the resource class identified by the *rsrc_class_name* parameter are selected.

attrs Specifies the persistent attributes to be set and their new values using a pointer to an array of *count* elements of type **mc_attribute_t**. Each element in the array specifies a persistent attribute of the resource and a value. Persistent attribute values of the resource that are not specified in this array remain unchanged.

count Specifies the number of elements in the *attrs* array.

OUTPUT

rsp_array
A pointer to a location in which the RMC API will return a pointer to a response array of *array_cnt* elements. This parameter applies only to variations of this subroutine that use the pointer response method.

array_cnt
Identifies a location in which the RMC API will return the number of elements in the response array *rsp_array*. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_set_selct_*** subroutine can be used by the application to set persistent attribute values of one or more resources of a resource class (identified by the *rsrc_class_name* parameter). The resource or resources are identified using a selection string (specified by the *select_attrs* parameter).

The response for these subroutines is a structure of type **mc_set_rsp_t**, and is described in Response structure.

If this command is used in an ordered command group, please note that all other commands in the command group must also be the form that uses attribute selection to specify the command target. The values of the *select_attrs* and *rsrc_class_name* parameters must be identical on all commands in the command group.

Security

To set persistent attribute values of a resource, the user of the calling application must have either **s** or **w** permission specified in an ACL entry for the resource.

Return values

For the **mc_set_select_bp** and **mc_set_select_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and one or more responses have been received and processed.

For the **mc_set_select_ap** and **mc_set_select_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_ETARGETMISMATCH
The target specified for the command does not match the target of the command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_EINVALIDDATATYPE
Invalid attribute data type specified.

MC_EINVALIDVALUEPTR
Invalid attribute value pointer specified.

MC_EINVALIDSDTYPE
Invalid structured data subtype specified.

MC_EINVALIDCNTPTR
Invalid response count pointer specified.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before all responses could be received. If subroutine uses pointer response, check the appropriate array count for any responses. If the subroutine uses callback response, the callback will have been invoked for any responses received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_set_rsp_t**. There will be one response for each resource identified by the *select_attrs* parameter. If any of the set attribute arguments are invalid, then only one response is returned, indicating the error, and no attributes are set (even if more than one resource was identified by the *select_attrs* parameter).

The response structure definition is:

```
typedef struct mc_set_rsp              mc_set_rsp_t;
struct mc_set_rsp {
     mc_errnum_t                       mc_error;
     ct_resource_handle_t              mc_rsrc_hndl;
     mc_error_attr_t                   *mc_error_attrs;
     ct_uint32_t                       mc_attr_count;
};
```

The fields of this structure contain the following:

mc_error
If 0 (zero), this field indicates that the command was successful, and, if the *mc_attr_count* is 0 (zero), all the specified attributes were set. If the *mc_attr_count* field is not 0 (zero), the *mc_error_attrs* field contains a pointer to an array of attributes that could not be set. Attributes not included in the array have been set. If the *mc_error* field is not 0 (zero), the value indicates the error.

mc_rsrc_hndl
If the command is successful, this field specifies the resource handle of the resource that was set. If an error is indicated by the *mc_error* field, this field contains the resource handle that was specified on the command.

mc_error_attrs
If no error is indicated by the *mc_error* field, and only some of the attributes could be set, this field is an array of *mc_attr_count* elements of type **mc_error_attr_t**.

```
typedef struct mc_error_attr       mc_error_attr_t;
struct mc_error_attr {
     mc_errnum_t                   mc_error;
     ct_char_t                     *mc_at_name;
};
```

Each element in the array identifies an attribute that could not be set. The fields of this structure contain the following:

mc_error
Indicates the reason the attribute could not be set.

mc_at_name
Indicates the name of the attribute that could not be set.

mc_attr_count
Indicates the number of entries in the *mc_error_attrs* array.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_class_set_*" on page 53
This subroutine sets persistent attribute values of a resource class.

"mc_qdef_p_attribute_*" on page 108
This subroutine queries the RMC subsystem to obtain the persistent attribute definitions for a resource or resource class.

mc_start_cmd_grp

This subroutine allocates a command group.

Purpose

Allocates a command group.

Library

RMC Library (**libct_mc.a**)

Syntax

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_start_cmd_grp(
          mc_sess_hndl_t              session_hndl,
          mc_cmd_grp_opts_t           options,
          mc_cmdgrp_hndl_t            *cmd_hndl)
```

Parameters

INPUT

session_hndl
The session handle that identifies the RMC subsystem session to which the command group will be sent. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

options Specifies one of the following:

MC_CMD_GRP_OPTS_ORDERED
This is an ordered command group. Process the commands in the order in which they are placed in the group.

MC_REG_OPTS_NONE
There are no options.

OUTPUT

cmd_hndl
An address allocated by the application for the command group handle. This subroutine, if successful, returns the command group handle at the specified address. The command group handle can then be used in subsequent RMC API calls.

Description

The **mc_start_cmd_group** subroutine can be used by the application to allocate a command group. A command group enables an application to send multiple commands to the RMC subsystem. The

subroutine returns a command handle that the application can use in subsequent RMC API calls to identify the command group. Many of the RMC interfaces consist of four related subroutines that issue the same command action, but vary on how the command is sent to the RMC subsystem, and how the command response is made available to the application. The subroutine variations that are suffixed by **_ap** or **_ac** add the particular command to a command group. The maximum number of commands that can be added to a command group can be determined using the **MC_CMD_GRP_LIMIT** macro.

Two subroutines are available to send a command group to the RMC subsystem. These are the:

- **mc_send_cmd_grp** subroutine which returns immediately after sending the command group
- **mc_send_cmd_grp_wait** subroutine which sends the command group and then blocks execution of the application thread until responses have been received and processed for each command in the command group.

If the application does not need to send the command group, it can use the **mc_cancel_cmd_grp** subroutine to deallocate the command group.

If the command group is an ordered command group (as determined by the *options* parameter), its commands will be processed in the order in which they were placed in the group. All commands placed in an ordered command group must specify the same resource or set of resources. If the target (or targets) of the first command in an ordered command group is specified using attribute selection, then the remaining commands in the group must also use attribute selection with identical attribute selection arguments. If the target of the first command in an ordered command group is specified using a resource handle, then the remaining commands in the group must use the identical resource handle. An ordered command group cannot contain a mix of commands that use attribute selection and resource handles to specify their targets.

Return values

A return value of 0 (zero) indicates that a command group has been allocated and a command handle has been returned at the address specified by the *cmd_hndl* parameter. Any other return value is an error and indicates that the command group was not allocated.

The following errors can be returned by this subroutine. Additional error information can be returned by calling the **cu_get_error_function**.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDOPT
The specified option is invalid.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_ENOMEM
The API could not allocate required memory.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_cancel_cmd_grp" on page 41
This subroutine cancels a command group.

mc_start_session

This subroutine establishes a session with the RMC subsystem.

Purpose

Establishes a session with the RMC subsystem.

Library

RMC Library (**libct_mc.a**)

Syntax

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_start_session(
          ct_contact_t                *contact_array,
          ct_uint32_t                 number_of_contacts,
          mc_session_opts_t           options,
          mc_sess_hndl_t              *session_hndl)
```

Parameters

INPUT

contact_array

Specifies one or more nodes that the RMC API can contact to start a session with the RMC subsystem. Specifies this using either a NULL pointer (indicating the local node on which the application is executing) or an array of *number_of_contacts* elements of type **ct_contact_t** (each array element representing a machine running the RMC subsystem).

If using a NULL pointer to specify the local node, the *number_of_contacts* parameter must be 0 (zero). The local node can be a cluster node or a single system.

If this parameter specifies an array identifying more than one contact node, all contact nodes should be in the same RSCT peer domain. It only makes sense for this parameter to specify an array of more than one element if the contact type is CT_CONTACT_IP (since the other contact types specify a single node).

The **mc_start_session** subroutine attempts to start a session with the RMC subsystem on the contact node in the array. If a session cannot be established on that node, the subroutine attempts to establish a session with the RMC subsystem on the next contact node in the array. This process continues until either a session with the RMC subsystem is established or the array is exhausted. The **ct_contact_t** data type is defined as:

```
typedef struct {
     ct_contact_type_t      type;
     ct_contact_point_t     point;
} ct_contact_t;
```

The fields of this structure contain the following items:

type Specifies one of the three supported contact types, as defined by the **ct_contact_type_t** enumeration.

```
typedef enum {
     CT_CONTACT_IP,
     CT_CONTACT_ENV_VAR,
     CT_CONTACT_LOCAL
} ct_contact_type_t;
```

The **ct_contact_type_t** enumeration defines the following values:

CT_CONTACT_IP
Indicates that the subroutine should attempt to establish a session with the RMC subsystem executing on the node identified in the *point* field of this structure.

CT_CONTACT_ENV_VAR
Indicates that the subroutine should attempt to establish a session with the RMC subsystem executing on the node identified by the CT_CONTACT environment variable. The CT_CONTACT environment variable can identify the node using a host name or an IP address. If the application specifies this contact type, the *point* field of this structure is ignored.

CT_CONTACT_LOCAL
Indicates that the subroutine should attempt to establish a session with the RMC subsystem that is executing on the local node. If the application specifies this contact type, the *point* field of this structure is ignored.

point If the contact type is specified as CT_CONTACT_ENV_VAR or CT_CONTACT_LOCAL the RMC API ignores this field. If the contact type is specified as CT_CONTACT_IP, then this field identifies the node running the RMC subsystem to which the subroutine should attempt to connect. The node can be identified by its host name or IP address. This field is a union of type **ct_contact_type_t**.

```
typedef union {
   ct_IP_contact_point_t   IP_point;
} ct_contact_point_t;
```

And the contact point itself, identified in the *IP_point* field is defined as a structure of type **ct_IP_contact_point_t**.

```
typedef struct {
     ct_int32_t         port;
     ct_char_ptr_t      name;
} ct_IP_contact_point_t;
```

The fields of this structure specify the following items:

port This field is not used by the **mc_start_session** subroutine and should be set to 0.

name Specifies the name or IP address of the contact point.

The following example shows how to initialize a contact of type CT_CONTACT_IP.

```
ct_contact_t c;


c.type = CT_CONTACT_IP;
c.point.IP_point.port = 0;
c.point.IP_point.name = "host_name_or_IP_address";
```

number_of_contacts
Specifies the number of elements in the *contact_array* array. If the *contact_array* parameter specifies a NULL pointer, this parameter must be 0 (zero).

options Either specifies the value MC_SESSION_OPTS_NONE (indicating there are no options), or else is the bitwise inclusive or of one or more of the following options. Most of these options are for setting the *session scope*. The RMC subsystem with which the session is established may be executing in a stand-alone environment, in an RSCT peer domain, in a CSM management domain, or in both an RSCT peer domain and a CSM management domain. Furthermore, if the RMC subsystem is running in a CSM management domain, it may be designated as the *distinguished daemon* (meaning the RMC subsystem is executing on the management server of the management domain). By specifying a session scope using this parameter, the application determines the scope for all commands issued in the session. The commands are issued within:

- An RSCT peer domain, which in the options that follow are identified by SR for shared resource.
- A CSM management domain, which in the options that follow are identified by DM for distributed management.
- Locally on the node that runs the RMC subsystem where the session was established.

Although this parameter can be the bitwise inclusive or of one or more of the following options, note that:

- The options for specifying the scope of the session are mutually exclusive
- The MC_SESSION_OPTS_IP_AUTHENTICATION option can only be used if the contact name is an IP address.

MC_SESSION_OPTS_LOCAL_SCOPE
The session scope is local. Commands are issued only on the node where the session was established. This is the default if no session scope is specified.

MC_SESSION_OPTS_SR_SCOPE
The session scope is an RSCT peer domain (also known as SR scope). Commands will be issued within the RSCT peer domain that contains the node where the session was established. If the execution environment of the RMC subsystem on the node at the time this subroutine is invoked does not include an RSCT peer domain, the subroutine will return an error.

MC_SESSION_OPTS_DM_SCOPE
The session scope is a CSM management domain (also know and DM scope). Commands will be issued within the CSM management domain that contains the node where the session was established. The node specified by the contact array parameter should be the CSM management server. If the node running the RMC subsystem with which the session is established is not a CSM management server, the subroutine will return an error.

MC_SESSION_OPTS_SR_LOCAL_SCOPE
The session scope is either an RSCT peer domain or local (depending on the execution environment of the RMC subsystem at the time the **mc_start_session** subroutine is invoked). If the RMC subsystem is running in an RSCT peer domain, the scope is the peer domain. If the execution environment of the RMC subsystem on the node at the time this subroutine is invoked does not include an RSCT peer domain, the scope is local.

MC_SESSION_OPTS_DM_LOCAL_SCOPE
The session scope is either a CSM management domain or local (depending on the execution environment of the RMC subsystem at the time the **mc_start_session** subroutine is invoked). If the node running the RMC subsystem with which the session is established is a CSM management server, the session scope is the CSM management domain. Otherwise the scope is local.

MC_SESSION_OPTS_SR_DM_SCOPE
The session scope is either an RSCT peer domain or a CSM management domain (depending on the execution environment of the RMC subsystem at the time the **mc_start_session** subroutine is invoked). If the RMC subsystem is running in an RSCT peer domain, the scope is the peer domain. Otherwise, if the node running the RMC

subsystem with which the session is established is a CSM management server, the session scope is the CSM management domain. If the node is neither in a peer domain, nor the management server in a management domain, the subroutine will return an error.

MC_SESSION_OPTS_DM_SR_SCOPE

The session scope is either a CSM management domain or an RSCT peer domain (depending on the execution environment of the RMC subsystem at the time the **mc_start_session** subroutine is invoked). If the node running the RMC subsystem with which the session is established is a CSM management server, the session scope is the CSM management domain. Otherwise, if the RMC subsystem is running in an RSCT peer domain, the scope is the peer domain. If the node is neither the management server in a management domain, nor in a peer domain, the subroutine will return an error.

MC_SESSION_OPTS_SR_DM_LOCAL_SCOPE

The session scope is either an RSCT peer domain, a CSM management domain, or local (depending on the execution environment of the RMC subsystem at the time the **mc_start_session** subroutine is invoked). If the RMC subsystem is running in an RSCT peer domain, the scope is the peer domain. Otherwise, if the node running the RMC subsystem with which the session is established is a CSM management server, the session scope is the CSM management domain. If the node is neither in a peer domain, nor the management server in a management domain, the scope is local.

MC_SESSION_OPTS_DM_SR_LOCAL_SCOPE

The session scope is either a CSM management domain, an RSCT peer domain, or local (depending on the execution environment of the RMC subsystem at the time the **mc_start_session** subroutine is invoked). If the node running the RMC subsystem with which the session is established is a CSM management server, the session scope is the CSM management domain. Otherwise, if the RMC subsystem is running in an RSCT peer domain, the scope is the peer domain. If the node is neither the management server in a management domain, nor in a peer domain, the scope is local.

MC_SESSION_OPTS_IP_AUTHENTICATION

Specifies that one of the IP addresses configured on the host where the application is executing should be included in the user's network credentials. If this option is not specified, the host name is either a fully qualified host domain name or a "short" name, depending on the host's DNS configuration. The application can specify this option to avoid a dependency on DNS.

OUTPUT

session_hndl

The address, allocated by the application, for the session handle. The subroutine, if successful, returns the session handle at this address.

Description

The **mc_start_session** subroutine can be used by the application to establish a session with the RMC subsystem on a node identified by the *contact_array* parameter. Depending on the execution environment of the RMC subsystem with which the session is established, and any session scope option specified by the *options* parameter, the scope of the RMC session will be an RSCT peer domain, a CSM management domain, or the local node.

An error is returned if the requested scope option specified by the options parameter cannot be supported by the current execution environment of the RMC subsystem with which the session is established. The following table illustrates what the resulting session scope would be depending on the execution environment of the RMC subsystem and the session scope option specified by the *options* parameter.

Table 24. Session scope depending on the execution environment of the RMC subsystem and the session scope option

Session scope option specified by the *options* parameter	Stand-alone environment	Peer domain	Management server in a management domain	Peer domain and on the management server in a management domain	Managed node in a management domain *	Peer domain and a managed node in a management domain *
MC_SESSION_OPTS_LOCAL_SCOPE or if no session scope is specified	local	local	local	local	local	local
MC_SESSION_OPTS_SR_SCOPE	error	peer domain	error	peer domain	error	peer domain
MC_SESSION_OPTS_DM_SCOPE	error	error	management domain	management domain	error	error
MC_SESSION_OPTS_SR_LOCAL_SCOPE	local	peer domain	local	peer domain	local	peer domain
MC_SESSION_OPTS_DM_LOCAL_SCOPE	local	local	management domain	management domain	local	local
MC_SESSION_OPTS_SR_DM_SCOPE	error	peer domain	management domain	peer domain	error	peer domain
MC_SESSION_OPTS_DM_SR_SCOPE	error	peer domain	management domain	management domain	error	peer domain
MC_SESSION_OPTS_SR_DM_LOCAL_SCOPE	local	peer domain	management domain	peer domain	local	peer domain
MC_SESSION_OPTS_DM_SR_LOCAL_SCOPE	local	peer domain	management domain	management domain	local	peer domain

* If a node is a managed node in one CSM management domain and also the management server in another management domain, see the management server columns above for the resulting session scope.

The subroutine, if successful, returns the session handle at the address specified in the *session_hndl* parameter. Subsequent calls to the RMC API can identify this session using the returned session handle. The session scope established when the **mc_start_session** subroutine is called determines where subsequent commands issued in the session using other RMC API calls are executed, and where any associated selection string is evaluated.

After a session is established successfully, if the *options* parameter is set to the **MC_SESSION_OPTS_SR_SCOPE** session scope and if the execution environment is going out of SR scope, an error message, which indicates that the session is ended, is returned to the application. Similarly, a session is ended if the present execution environment is going out of SR scope, and if the session was established with a scope option that resolved to SR scope. For example, if the contact node or peer domain is brought offline, a session that specified the **MC_SESSION_OPTS_SR_LOCAL_SCOPE** session scope on an online node in a peer domain is ended. If the scope option is not specified, or if the **MC_SESSION_OPTS_LOCAL_SCOPE** session scope is specified, a change in execution environment has no impact on the session.

Any error messages or descriptive text returned in responses resulting from commands issued in this session, are returned in the locale that was in effect at the time the **mc_start_session** subroutine was invoked. By calling the **setlocale** subroutine and then the **mc_start_session** subroutine, the application can change the locale used by the RMC subsystem when returning messages or descriptive text.

It is expected that all subroutines that specify the session handle returned by a successful call to the **mc_start_session** subroutine are compiled using the same value of the MC_VERSION macro used when compiling the call to the **mc_start_session** subroutine. Otherwise, the MC_EVERSIONMISMATCH error is returned by the other subroutines.

When the RMC API detects the interruption of a session with the RMC subsystem, it completes the processing of any responses and event notifications that have been received before the session interruption. Any application threads that are blocked in API subroutines, including the **mc_dispatch** subroutine, return with an error indicating the session was interrupted. If the application has obtained a

descriptor using the **mc_get_descriptor** subroutine, the API makes the descriptor ready for read to ensure that the application has an opportunity to call the **mc_dispatch** subroutine and receive the error. If any subroutine, other than the **mc_end_session** subroutine is invoked, then the subroutine call returns the error.

Once the application has received the error indicating the session was interrupted, it must call the **mc_end_session** subroutine. The application can then establish a new session by once again calling the **mc_start_session** subroutine. If the subroutine returns the MC_ESESSREFUSED error, the application should call the subroutine again after a short time, since the RMC subsystem may not yet have recovered. This can be repeated as many times as appropriate for the application until a new session is established. Once the new session is established, the application must register any events that were registered in the old session (if the event notifications are still required).

The managed nodes in a CSM management domain may be configured into one or more RSCT peer domains. In such a case, any application with a management domain session scope can access peer domain resources by specifying the peer domain name (known as the Active Peer Domain name).

Security

When establishing a session, the RMC subsystem authenticates the user of the calling application. The RMC API, if appropriate, authenticates the RMC subsystem. Successful authentication is a necessary condition for the session to be established. If the session is established, the network credentials of the user are saved by the RMC subsystem to perform authorization checks as various command interfaces are executed. In an RSCT peer domain or a CSM management domain, the cluster the authorization checks are performed on the node where the command is actually executed. This is typically, but not always, the node where the resource is located. For this reason, ACLs specifying the user's network credentials should be replicated throughout the cluster.

Return values

A return value of 0 indicates that a session was established with the RMC subsystem, and a session handle is available at the location that is specified by the *session_hndl* parameter. The session handle can now be used in subsequent RMC API calls to send commands to, and receive responses from, the RMC subsystem for the session scope. Any nonzero value returned by this subroutine is an error, and indicates that the session was not established. The following errors can be returned by this subroutine. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
: A severe library or system error occurred.

MC_ELIBNOMEM
: A severe library memory allocation error occurred.

MC_ESESSREFUSED
: A session could not be established with the RMC subsystem in the specified cluster. The application should try again later.

MC_EINVALIDNAME
: Invalid contact name specified.

MC_EINVALSCOPE
: The specified session scope is not currently supported.

MC_EMULTISCOPE
: Multiple session scope options are specified.

MC_ENOTPRIVATESCOPE
: The specified session scope cannot be combined with the private option.

MC_ELIBSECURITY
The RMC API detected an error in security services.

MC_ESUBSECURITY
The RMC subsystem detected an error in security services.

MC_EAUTHENTICATE
Could not authenticate the user of the calling application.

MC_EMUTUALAUTHENT
The RMC API could not authenticate the RMC subsystem.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDCONTACT
Invalid contact type specified.

MC_ENOMEM
The API could not allocate required memory.

Location

/usr/lib/libct_mc.a

Related concepts:

"RMC subsystem session" on page 2
A resource monitoring and control (RMC) subsystem session is a connection with the RMC subsystem that the application establishes through an RMC daemon that runs on a particular node.

Related reference:

"mc_end_session" on page 64
This subroutine ends a session with the RMC subsystem.

mc_timed_start_session

This subroutine establishes a session with the RMC subsystem.

Purpose

Establishes a session with the RMC subsystem. This subroutine is identical to the **mc_start_session** subroutine except that it also enables the calling application to specify time limits for establishing a session and, once a session is established, for completion of blocking operations.

Library

RMC Library (**libct_mc.a**)

Syntax

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_start_session(
          ct_contact_t                *contact_array,
          ct_uint32_t                 number_of_contacts,
          mc_session_opts_t           options,
          ct_uint32_t                 start_timeout,
          ct_uint32_t                 cmd_timeout,
          mc_sess_hndl_t              *session_hndl)
```

Parameters

INPUT

contact_array

Specifies one or more nodes the RMC API may contact to start a session with the RMC subsystem. Specifies this using either a NULL pointer (indicating the local node on which the application is executing) or an array of *number_of_contacts* elements of type **ct_contact_t** (each array element representing a machine running the RMC subsystem).

If using a NULL pointer to specify the local node, the *number_of_contacts* parameter must be 0 (zero). The local node can be a cluster node or a single system.

If this parameter specifies an array identifying more than one contact node, all contact nodes should be in the same RSCT peer domain. It only makes sense for this parameter to specify an array of more than one element if the contact type (described below) is CT_CONTACT_IP (since the other contact types specify a single node).

The **mc_timed_start_session** subroutine will attempt to start a session with the RMC subsystem on the contact node in the array. If a session cannot be established on that node, then the subroutine will attempt to establish a session with the RMC subsystem on the next contact node in the array. This process will continue until either a session with the RMC subsystem is established or the array is exhausted. The **ct_contact_t** data type is defined as:

```
typedef struct {
     ct_contact_type_t      type;
     ct_contact_point_t     point;
} ct_contact_t;
```

The fields of this structure contain the following:

type Specifies one of the three supported contact types, as defined by the **ct_contact_type_t** enumeration.

```
typedef enum {
     CT_CONTACT_IP,
     CT_CONTACT_ENV_VAR,
     CT_CONTACT_LOCAL
} ct_contact_type_t;
```

The **ct_contact_type_t** enumeration defines the following values:

CT_CONTACT_IP
Indicates that the subroutine should attempt to establish a session with the RMC subsystem executing on the node identified in the *point* field of this structure.

CT_CONTACT_ENV_VAR
Indicates that the subroutine should attempt to establish a session with the RMC subsystem executing on the node identified by the CT_CONTACT environment variable. The CT_CONTACT environment variable can identify the node using a host name or an IP address. If the application specifies this contact type, the *point* field of this structure is ignored.

CT_CONTACT_LOCAL
Indicates that the subroutine should attempt to establish a session with the RMC subsystem that is executing on the local node. If the application specifies this contact type, the *point* field of this structure is ignored.

point If the contact type is specified as CT_CONTACT_ENV_VAR or CT_CONTACT_LOCAL the RMC API ignores this field. If the contact type is specified as CT_CONTACT_IP, then this field identifies the node running the RMC subsystem to which the subroutine should attempt to connect. The node can be identified by its host name or IP address. This field is a union of type ct_contact_point_t.

```
typedef union {
   ct_IP_contact_point_t   IP_point;
} ct_contact_point_t;
```

And the contact point itself, identified in the *IP_point* field is defined as a structure of type **ct_IP_contact_point_t**.

```
typedef struct {
     ct_int32_t         port;
     ct_char_ptr_t      name;
} ct_IP_contact_point_t;
```

The fields of this structure specify the following:

port This field is not used by the **mc_timed_start_session** subroutine and should be set to 0.

name Specifies the name or IP address of the contact point.

The following example shows how to initialize a contact of type CT_CONTACT_IP.

```
ct_contact_t c;


c.type = CT_CONTACT_IP;
c.point.IP_point.port = 0;
c.point.IP_point.name = "host_name_or_IP_address";
```

number_of_contacts
Specifies the number of elements in the *contact_array* array. If the *contact_array* parameter specifies a NULL pointer, this parameter must be 0 (zero).

options Either specifies the value MC_SESSION_OPTS_NONE (indicating there are no options), or else is the bitwise inclusive OR of one or more of the following options. Most of these options are for setting the *session scope*. The RMC subsystem with which the session is established may be executing in a stand-alone environment, in an RSCT peer domain, in a CSM management domain, or in both an RSCT peer domain and a CSM management domain. Furthermore, if the RMC subsystem is running in a CSM management domain, it may be designated as the *distinguished daemon* (meaning the RMC subsystem is executing on the management server of the management domain). By specifying a session scope using this parameter, the application determines the scope for all commands issued in the session. The commands will be issued within either

- an RSCT peer domain (which in the options that follow are identified by **SR** for *shared resource*)
- a CSM management domain (which in the options that follow are identified by **DM** for *distributed management*)
- locally on the node running the RMC subsystem where the session was established.

Although this parameter can be the bitwise inclusive OR of the following options, please note that:

- the nine options for specifying the scope of the session are mutually exclusive
- the MC_SESSION_OPTS_IP_AUTHENTICATION option may only be used if the contact name is an IP address.

MC_SESSION_OPTS_LOCAL_SCOPE
The session scope is local. Commands will be issued only on the node where the session was established. This is the default if no session scope is specified.

MC_SESSION_OPTS_SR_SCOPE
The session scope is an RSCT peer domain (also known as SR scope). Commands will be issued within the RSCT peer domain that contains the node where the session was

established. If the execution environment of the RMC subsystem on the node at the time this subroutine is invoked does not include an RSCT peer domain, the subroutine will return an error.

MC_SESSION_OPTS_DM_SCOPE
The session scope is a CSM management domain (also know and DM scope). Commands will be issued within the CSM management domain that contains the node where the session was established. The node specified by the contact array parameter should be the CSM management server. If the node running the RMC subsystem with which the session is established is not an CSM management server, the subroutine will return an error.

MC_SESSION_OPTS_SR_LOCAL_SCOPE
The session scope is either an RSCT peer domain or local (depending on the execution environment of the RMC subsystem at the time the **mc_start_session** subroutine is invoked). If the RMC subsystem is running in an RSCT peer domain, the scope is the peer domain. If the execution environment of the RMC subsystem on the node at the time this subroutine is invoked does not include an RSCT peer domain, the scope is local.

MC_SESSION_OPTS_DM_LOCAL_SCOPE
The session scope is either a CSM management domain or local (depending on the execution environment of the RMC subsystem at the time the **mc_start_session** subroutine is invoked). If the node running the RMC subsystem with which the session is established is a CSM management server, the session scope is the CSM management domain. Otherwise the scope is local.

MC_SESSION_OPTS_SR_DM_SCOPE
The session scope is either an RSCT peer domain or a CSM management domain (depending on the execution environment of the RMC subsystem at the time the **mc_start_session** subroutine is invoked). If the RMC subsystem is running in an RSCT peer domain, the scope is the peer domain. Otherwise, if the node running the RMC subsystem with which the session is established is a CSM management server, the session scope is the CSM management domain. If the node is neither in a peer domain, nor the management server in a management domain, the subroutine will return an error.

MC_SESSION_OPTS_DM_SR_SCOPE

The session scope is either a CSM management domain or an RSCT peer domain (depending on the execution environment of the RMC subsystem at the time the **mc_start_session** subroutine is invoked). If the node running the RMC subsystem with which the session is established is a CSM management server, the session scope is the CSM management domain. Otherwise, if the RMC subsystem is running in an RSCT peer domain, the scope is the peer domain. If the node is neither the management server in a management domain, nor in a peer domain, the subroutine will return an error.

MC_SESSION_OPTS_SR_DM_LOCAL_SCOPE
The session scope is either an RSCT peer domain, a CSM management domain, or local (depending on the execution environment of the RMC subsystem at the time the **mc_start_session** subroutine is invoked). If the RMC subsystem is running in an RSCT peer domain, the scope is the peer domain. Otherwise, if the node running the RMC subsystem with which the session is established is a CSM management server, the session scope is the CSM management domain. If the node is neither in a peer domain, nor the management server in a management domain, the scope is local.

MC_SESSION_OPTS_DM_SR_LOCAL_SCOPE

The session scope is either a CSM management domain, an RSCT peer domain, or local (depending on the execution environment of the RMC subsystem at the time the **mc_start_session** subroutine is invoked). If the node running the RMC subsystem with which the session is established is a CSM management server, the session scope is the CSM management domain. Otherwise, if the RMC subsystem is running in an RSCT peer

domain, the scope is the peer domain. If the node is neither the management server in a management domain, nor in a peer domain, the scope is local.

MC_SESSION_OPTS_IP_AUTHENTICATION
Specifies that one of the IP addresses configured on the host where the application is executing should be included in the user's network credentials. If this option is not specified, the host name is either a fully qualified host domain name or a "short" name, depending on the host's DNS configuration. The application can specify this option in order to avoid a dependency on DNS.

start_timeout
If non-zero, specifies a time limit (in seconds) for establishing a session with the RMC subsystem. This limits the amount of time the subroutine waits for responses from the RMC subsystem running on the nodes specified by the *contact_array* parameter. If this limit is exceeded, the error MC_ETIMEDOUT is returned. Due to time limits used in authentication processing, the actual subroutine timeout may be longer than the value specified.

If zero, specifies that there is no time limit for establishing an RMC subsystem.

cmd_timeout
If non-zero, specifies the number of seconds that any blocking subroutine (including the **mc_send_cmd_grp_wait** subroutine) waits for a completion response from the RMC subsystem. If a completion response is not received within the specified time limit, the error MC_ETIMEDOUT is returned. This error does not imply that the subroutine command did not complete; it indicates only that the timeout value was reached before a completion response could be received.

When the **mc_send_cmd_grp_wait** subroutine returns the MC_ETIMEDOUT error, this means that the command group has been sent to the RMC subsystem, but none or only some of the command group responses have been delivered to the application.

If the command group contains one or more event registration commands, the RMC API unregisters the events. Some event notifications, however, may be delivered to the application before the events are unregistered.

When a blocking subroutine returns the MC_ETIMEDOUT error, this means that the command has been sent to the RMC subsystem, but none or only some of the command responses have been delivered to the application. If the subroutine was an event registration command, the RMC API unregisters the event. Some event notifications, however, may be delivered to the application before the event is unregistered.

If the value of this parameter is zero, this specifies that there is no time limit for the blocking subroutine calls.

OUTPUT

session_hndl
The address, allocated by the application, for the session handle. The subroutine, if successful, returns the session handle at this address.

Description

The **mc_timed_start_session** subroutine can be used by the application to establish a session with the RMC subsystem on a node identified by the *contact_array* parameter.

This subroutine behaves exactly that same way as the **mc_start_session** subroutine, except for the addition of the *start_timeout* and *cmd_timeout* parameters. If the *start_timeout* parameter specifies a non-zero value, it indicates a time limit (in seconds) for establishing a session with the RMC subsystem. If the *cmd_timeout* parameter specifies a non-zero value, it indicates a time limit (in seconds) that a blocking subroutine (including the **mc_send_cmd_grp_wait** subroutine) waits for a completion callback.

Depending on the execution environment of the RMC subsystem with which the session is established, and any session scope option specified by the *options* parameter, the scope of the RMC session will be an RSCT peer domain, a CSM management domain, or the local node.

An error is returned if the requested scope option specified by the options parameter cannot be supported by the current execution environment of the RMC subsystem with which the session is established. The following table illustrates what the resulting session scope would be depending on the execution environment of the RMC subsystem and the session scope option specified by the *options* parameter.

Table 25. Session scope depending on the execution environment of the RMC subsystem and the session scope option

Session scope option specified by the *options* parameter	**Stand-alone environment**	**Peer domain**	**Management server in a management domain**	**Peer domain and on the management server in a management domain**	**Managed node in a management domain ***	**Peer domain and a managed node in a management domain ***
MC_SESSION_OPTS_LOCAL_SCOPE or if no session scope is specified	local	local	local	local	local	local
MC_SESSION_OPTS_SR_SCOPE	error	peer domain	error	peer domain	error	peer domain
MC_SESSION_OPTS_DM_SCOPE	error	error	management domain	management domain	error	error
MC_SESSION_OPTS_SR_LOCAL_SCOPE	local	peer domain	local	peer domain	local	peer domain
MC_SESSION_OPTS_DM_LOCAL_SCOPE	local	local	management domain	management domain	local	local
MC_SESSION_OPTS_SR_DM_SCOPE	error	peer domain	management domain	peer domain	error	peer domain
MC_SESSION_OPTS_DM_SR_SCOPE	error	peer domain	management domain	management domain	error	peer domain
MC_SESSION_OPTS_SR_DM_LOCAL_SCOPE	local	peer domain	management domain	peer domain	local	peer domain
MC_SESSION_OPTS_DM_SR_LOCAL_SCOPE	local	peer domain	management domain	management domain	local	peer domain

* If a node is a managed node in one CSM management domain and also the management server in another management domain, see the management server columns above for the resulting session scope.

The subroutine, if successful, returns the session handle at the address specified in the *session_hndl* parameter. Subsequent calls to the RMC API can identify this session using the returned session handle. The session scope established when the **mc_timed_start_session** subroutine is called determines where subsequent commands issued in the session using other RMC API calls are executed, and where any associated selection string is evaluated.

After a session is established successfully, if the *options* parameter is set to the **MC_SESSION_OPTS_SR_SCOPE** session scope and if the execution environment is going out of SR scope, an error message, which indicates that the session is ended, is returned to the application. Similarly, a session is ended if the present execution environment is going out of SR scope, and if the session was established with a scope option that resolved to SR scope. For example, if the contact node or peer domain is brought offline, a session that specified the **MC_SESSION_OPTS_SR_LOCAL_SCOPE** session scope on an online node in a peer domain is ended. If the scope option is not specified, or if the **MC_SESSION_OPTS_LOCAL_SCOPE** session scope is specified, a change in execution environment has no impact on the session.

Any error messages or descriptive text returned in responses resulting from commands issued in this session, are returned in the locale that was in effect at the time the **mc_timed_start_session** subroutine was invoked. By calling the **setlocale** subroutine and then the **mc_timed_start_session** subroutine, the application can change the locale used by the RMC subsystem when returning messages or descriptive text.

It is expected that all subroutines that specify the session handle returned by a successful call to the **mc_timed_start_session** subroutine are compiled using the same value of the MC_VERSION macro used when compiling the call to the **mc_timed_start_session** subroutine. Otherwise, the MC_EVERSIONMISMATCH error is returned by the other subroutines.

When the RMC API detects the interruption of a session with the RMC subsystem, it completes the processing of any responses and event notifications that have been received prior to the session interruption. Any application threads that are blocked in API subroutines, including the **mc_dispatch** subroutine, return with an error indicating the session was interrupted. If the application has obtained a descriptor using the **mc_get_descriptor** subroutine, the API makes the descriptor ready for read to ensure that the application has an opportunity to call the **mc_dispatch** subroutine and receive the error. If any subroutine, other than the **mc_end_session** subroutine is invoked, then the subroutine call returns the error.

Once the application has received the error indicating the session was interrupted, it must call the **mc_end_session** subroutine. The application can then establish a new session by once again calling the **mc_timed_start_session** subroutine. If the subroutine returns the MC_ESESSREFUSED error, the application should call the subroutine again after a short time, since the RMC subsystem may not yet have recovered. This can be repeated as many time as appropriate for the application until a new session is established. Once the new session is established, the application must register any events that were registered in the old session (if the event notifications are still desired).

The managed nodes in a CSM management domain may be configured into one or more RSCT peer domains. In such a case, any application with a management domain session scope can access peer domain resources by specifying the peer domain name (known as the Active Peer Domain name).

Security

When establishing a session, the RMC subsystem authenticates the user of the calling application. The RMC API, if appropriate, authenticates the RMC subsystem. Successful authentication is a necessary condition for the session to be established. If the session is established, the network credentials of the user are saved by the RMC subsystem in order to perform authorization checks as various command interfaces are executed. In an RSCT peer domain or a CSM management domain, the cluster the authorization checks are performed on the node where the command is actually executed. This is typically, but not always, the node where the resource is located. For this reason, ACLs specifying the user's network credentials should be replicated throughout the cluster.

Return values

A return value of 0 indicates that a session was established with the RMC subsystem, and a session handle is available at the location specified by the *session_hndl* parameter. The session handle can now be used in subsequent RMC API calls to send commands to, and receive responses from, the RMC subsystem for the session scope. Any non-zero value returned by this subroutine is an error, and indicates that the session was not established. The following errors can be returned by this subroutine. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_ESESSREFUSED
A session could not be established with the RMC subsystem in the specified cluster. The application should try again later.

MC_EINVALIDNAME
Invalid contact name specified.

MC_EINVALSCOPE
The specified session scope is not currently supported.

MC_EMULTISCOPE
Multiple session scope options are specified.

MC_ENOTPRIVATESCOPE
The specified session scope cannot be combined with the private option.

MC_ELIBSECURITY
The RMC API detected an error in security services.

MC_ESUBSECURITY
The RMC subsystem detected an error in security services.

MC_EAUTHENTICATE
Could not authenticate the user of the calling application.

MC_EMUTUALAUTHENT
The RMC API could not authenticate the RMC subsystem.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDCONTACT
Invalid contact type specified.

MC_ETIMEDOUT
The session could not be established prior to expiration of the timeout.

MC_ENOMEM
The API could not allocate required memory.

Location

/usr/lib/libct_mc.a

Related concepts:

"RMC subsystem session" on page 2
A resource monitoring and control (RMC) subsystem session is a connection with the RMC subsystem that the application establishes through an RMC daemon that runs on a particular node.

Related reference:

"mc_session_info" on page 193
This subroutine gets information about a session.

mc_undefine_resource_*

This subroutine removes a resource from the RMC subsystem.

Purpose

Removes a resource from the RMC subsystem.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_undefine_resource_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_undefine_resource_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_undefine_resource_bp(
          mc_sess_hndl_t            sess_hndl,
          mc_undefine_rsrc_rsp_t    **response,
          ct_resource_handle_t      rsrc_hndl,
          ct_structured_data_t      *data)
```

- The **mc_undefine_resource_ap** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_undefine_resource_ap(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_undefine_rsrc_rsp_t    **response,
          ct_resource_handle_t      rsrc_hndl,
          ct_structured_data_t      *data)
```

- The **mc_undefine_resource_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_undefine_resource_bc(
          mc_sess_hndl_t            sess_hndl,
          mc_undef_rsrc_cb_t        *undef_rsrc_cb,
          void                      *undef_rsrc_cb_arg,
          ct_resource_handle_t      rsrc_hndl,
          ct_structured_data_t      *data)
```

  The definition for the response callback is:

```
typedef void (mc_undef_rsrc_cb_t)(mc_sess_hndl_t,
mc_undefine_rsrc_rsp_t *,
void *);
```

- The **mc_undefine_resource_ac** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_undefine_resource_ac(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_undef_rsrc_cb_t        *undef_rsrc_cb,
          void                      *undef_rsrc_cb_arg,
          ct_resource_handle_t      rsrc_hndl,
          ct_structured_data_t      *data)
```

  The definition for the response callback is:

```
typedef void (mc_undef_rsrc_cb_t)(mc_sess_hndl_t,
mc_undefine_rsrc_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

undef_rsrc_cb
Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

undef_rsrc_cb_arg
Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

rsrc_hndl
The resource handle that identifies the resource to be undefined. A resource handle is returned in the response structure for many RMC API subroutines including the **mc_define_resource_*** subroutine. An array of resource handles for resources of a particular resource class is returned in the response structure for the **mc_enumerate_resources_*** and **mc_enumerate_permitted_rsrcs_*** subroutines. To validate the resource handle before calling this subroutine, the application can call one of the **mc_validate_rsrc_hndl_*** subroutines.

data A pointer to structured data containing resource-class specific options for undefining a resource. To accept the default values (or if the resource class does not define options) for undefining a resource, the *data* parameter should be a NULL pointer.

To obtain the syntax and semantics for the structured data required by the resource class for specifying undefine resource options, the application can use the **mc_qdef_sd_*** subroutines.

OUTPUT

response
A pointer to a location in which the RMC API will return a pointer to the response. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_undefine_resource_*** subroutines can be used by the application to undefine a resource (identified by the *rsrc_hndl* parameter). The resource manager associated with the resource will remove the actual instance. If the resource manager accepts structured data as options for undefining a resource, the application can provide this using the *data* parameter. Once a resource is undefined, the resource handle (specified by the *rsrc_hndl* parameter) is invalid and cannot be used in subsequent RMC API calls.

The response for these subroutines is a structure of type **mc_undefine_rsrc_rsp_t**, and is described in Response structure.

This command cannot be used in an ordered command group.

Security

To undefine a resource, the user of the calling application must have either the **d** or **w** permission specified in an ACL entry for the associated resource class.

Return values

For the **mc_undefine_resource_bp** and **mc_undefine_resource_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and a response has been received and processed.

For the **mc_undefine_resource_ap** and **mc_undefine_resource_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_EORDERGROUP
An attempt was made to add the command to an ordered command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before the response could be received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before the response could be received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_EINVALIDDATATYPE
Invalid attribute data type specified.

MC_EINVALIDVALUEPTR
Invalid attribute value pointer specified.

MC_EINVALIDSDTYPE
Invalid structured data subtype specified.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before the response could be received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_undefine_rsrc_rsp_t**. The response contains the resource handle of the resource that was undefined (provided the *mc_error* field of the **mc_undefine_rsrc_rsp_t** structure indicates there was no error). If the *mc_error* field indicates an error, the response contains the resource handle in error. This command results in only one response.

The response structure definition is:

```
typedef struct mc_undefine_rsrc_rsp     mc_undefine_rsrc_rsp_t;
struct mc_undefine_rsrc_rsp {
     mc_errnum_t                        mc_error;
     ct_char_t                          *mc_class_name;
     ct_resource_handle_t               mc_rsrc_hndl;
};
```

The fields of this structure contain the following:

mc_error
If 0 (zero), this field indicates that the command was successful. Any other value is an error, and indicates that the RMC subsystem or a resource manager could not complete the command. The error may also indicate that the command arguments were in error. The error codes imply which of the remaining fields in the structure are defined.

mc_class_name
The name of the resource class in which the resource instance is deleted.

mc_rsrc_hndl
The resource handle of the deleted resource instance.

Location

/usr/lib/libct_mc.a

Related reference:

"mc_define_resource_*" on page 58
This subroutine defines a new resource.

mc_unreg_event_*

This subroutine unregisters an event with the RMC subsystem.

Purpose

Unregisters an event with the RMC subsystem.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_unreg_event_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_unreg_event_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_unreg_event_bp(
          mc_sess_hndl_t            sess_hndl,
          mc_unreg_rsp_t            **response,
          mc_registration_id_t      registration_id)
```

- The **mc_unreg_event_ap** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>
ct_int32_t
     mc_unreg_event_ap(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_unreg_rsp_t            **response,
          mc_registration_id_t      registration_id)
```

- The **mc_unreg_event_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_unreg_event_bc(
          mc_sess_hndl_t            sess_hndl,
          mc_unreg_cb_t             *unreg_cb,
          void                      *unreg_cb_arg,
          mc_registration_id_t      registration_id)
```

  The definition for the response callback is:

```
typedef void (mc_unreg_cb_t)(mc_sess_hndl_t,
mc_unreg_rsp_t *,
void *);
```

- The **mc_unreg_event_ac** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_unreg_event_ac(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_unreg_cb_t             *unreg_cb,
          void                      *unreg_cb_arg,
          mc_registration_id_t      registration_id)
```

  The definition for the response callback is:

```
typedef void (mc_unreg_cb_t)(mc_sess_hndl_t,
mc_unreg_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_ start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

unreg_cb
Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

unreg_cb_arg
Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

registration_id
The event registration ID that identifies the event to be unregistered. A valid event registration ID is one returned by the **rm_reg_event_select_***, **mc_reg_event_handle_***, or **mc_reg_class_event_*** subroutines. In order for the registration ID to be valid, the response from the **rm_reg_event_select_***, **mc_reg_event_handle_***, or **mc_reg_class_event_*** subroutine must have indicated that the event was successfully registered.

If an event registration ID is specified in a call to an **mc_query_event_*** subroutine, and the command has not yet completed, you cannot specify the registration ID when calling an **mc_unreg_event_*** subroutine.

OUTPUT

response
A pointer to a location in which the RMC API will return a pointer to the response. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_unreg_event_*** subroutines can be used by the application to unregister a resource or resource class event that was previously registered using the **mc_reg_event_select_***, **mc_reg_event_handle_***, or **mc_reg_class_event_*** subroutines.

The response for these subroutines is a structure of type **mc_unreg_rsp_t**, and is described in Response structure. The *mc_error* field of the **mc_unreg_rsp_t** structure indicates whether or not the event has been successfully unregistered. Included in a successful response is the registration ID of the event that was unregistered. The RMC subsystem will not pass this response to the application until all event notification callbacks being processed for the event have completed. This enables the application to safely release any resources associated with the event when it receives this response.

In order to avoid deadlocks, the blocking versions of this interface (the **mc_unreg_event_bp** and **mc_unreg_event_bc** subroutines) should not be called from within an event notification callback. Also to avoid deadlocks, the **mc_send_cmd_grp_wait** subroutine should not be called from within an event notification callback if the command group it would be sending contains commands added by the **mc_unreg_event_ap** or **mc_unreg_event_ac** subroutines.

This command cannot be used in an ordered command group.

Security

To unregister a resource or resource class event, the user of the calling application must have either **s** or **r** permission specified in an ACL entry for the resource or resource class.

Return values

For the **mc_unreg_event_bp** and **mc_unreg_event_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and a response has been received and processed.

For the **mc_unreg_event_ap** and **mc_unreg_event_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_EORDERGROUP
An attempt was made to add the command to an ordered command group.

MC_EDEADLOCK
An attempt was made to invoke the command from within an event notification callback.

MC_EQEVENTACTIVE
A query event command has not yet completed.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_EINVALIDEID
The specified registration ID is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before the response could be received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before the response could be received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before the response could be received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_unreg_rsp_t**. The response structure definition is:

```
typedef struct mc_unreg_rsp             mc_unreg_rsp_t;
struct mc_unreg_rsp {
     mc_errnum_t                        mc_error;
     mc_registration_id_t              mc_registration_id;
};
```

The fields of this structure contain the following:

mc_error
This field is always 0 (zero), indicating that the event has been successfully unregistered.

mc_registration_id
The event registration ID of the event that was unregistered. This registration ID is now invalid and cannot be used in subsequent calls to the RMC API.

The RMC subsystem guarantees that this response is passed to the application only when no event notification callbacks for the specified event are being executed. Once the response has been passed to the application, no more event notifications for this event will be delivered to the application. This permits the application to release any application resources associated with the event when it receives the response.

Location

/usr/lib/libct_mc.a

mc_validate_rsrc_hndl_*

This subroutine validates one or more resource handles.

Purpose

Validates one or more resource handles.

Library

RMC Library (**libct_mc.a**)

Syntax

Like many of the RMC interfaces, there are four **mc_validate_rsrc_hndl_*** subroutines. All four subroutines issue the same command action, but enable your application to vary how the command is sent to the RMC subsystem, and how the command response is made available to the application.

- The **mc_validate_rsrc_hndl_bp** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_validate_rsrc_hndl_bp(
          mc_sess_hndl_t            sess_hndl,
          mc_rsrc_hndl_rsp_t        **rsp_array,
          ct_uint32_t               *array_cnt,
          ct_resource_handle_t      *rsrc_hndl,
          ct_uint32_t               rsrc_hndl_cnt)
```

- The **mc_validate_rsrc_hndl_ap** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the pointer response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_validate_rsrc_hndl_ap(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_rsrc_hndl_rsp_t        **rsp_array,
          ct_uint32_t               *array_cnt,
          ct_resource_handle_t      *rsrc_hndl,
          ct_uint32_t               rsrc_hndl_cnt)
```

- The **mc_validate_rsrc_hndl_bc** subroutine sends the command to the RMC subsystem and blocks execution. To receive responses, it specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_validate_rsrc_hndl_bc(
          mc_sess_hndl_t            sess_hndl,
          mc_val_rsrc_hndl_cb_t     *val_rsrc_hndl_cb,
          void                      *val_rsrc_hndl_cb_arg,
          ct_resource_handle_t      *rsrc_hndl,
          ct_uint32_t               rsrc_hndl_cnt)
```

  The definition for the response callback is:

```
typedef void (mc_val_rsrc_hndl_cb_t)(mc_sess_hndl_t,
mc_rsrc_hndl_rsp_t *,
void *);
```

- The **mc_validate_rsrc_hndl_ac** subroutine adds the command to a command group. Please note that this command cannot be used in an ordered command group. To receive responses, this subroutine specifies the callback response method. The syntax is:

```
#include <rsct/ct_mc_v6.h>

ct_int32_t
     mc_validate_rsrc_hndl_ac(
          mc_cmdgrp_hndl_t          cmdgrp_hndl,
          mc_val_rsrc_hndl_cb_t     *val_rsrc_hndl_cb,
          void                      *val_rsrc_hndl_cb_arg,
          ct_resource_handle_t      *rsrc_hndl,
          ct_uint32_t               rsrc_hndl_cnt)
```

  The definition for the response callback is:

```
typedef void (mc_val_rsrc_hndl_cb_t)(mc_sess_hndl_t,
mc_rsrc_hndl_rsp_t *,
void *);
```

Parameters

INPUT

sess_hndl
The session handle that identifies the RMC subsystem session. A session handle is returned by the **mc_start_session** or **mc_timed_start_session** subroutine when the application establishes a session with the RMC subsystem.

cmdgrp_hndl
The command group handle that identifies the command group to which this command should be added. A command group handle is returned by the **mc_start_cmd_grp** subroutine when the application allocates a command group. This parameter applies only to variations of this subroutine that add the command to a command group.

val_rsrc_hndl_cb
Identifies the callback routine that will be invoked by the RMC API to return command responses to the application. This parameter applies only to the variations of this subroutine that use the callback response method.

val_rsrc_hndl_cb_arg
Identifies the argument that the RMC API will use to pass command responses to the callback routine. This parameter applies only to the variations of this subroutine that use the callback response method.

rsrc_hndl
Identifies the resource handles to be validated using a pointer to an array of *rsrc_hndl_cnt* elements of type **ct_resource_handle_t**.

rsrc_hndl_cnt
Specifies the number of elements in the **rsrc_hndl** array.

OUTPUT

rsp_array
A pointer to a location in which the RMC API will return a pointer to a response array of *array_cnt* elements. This parameter applies only to variations of this subroutine that use the pointer response method.

array_cnt
Identifies a location in which the RMC API will return the number of elements in the response array *rsp_array*. This parameter applies only to variations of this subroutine that use the pointer response method.

Description

The **mc_validate_rsrc_hndl_*** subroutines can be used by the application to verify that one or more resource handles (identified by the *rsrc_hndl* parameter) are still valid. This enables the application to ensure that a resource handle previously obtained by the RMC subsystem represents a resource that is still defined by its resource manager.

The response for these subroutines is a structure of type **mc_rsrc_hndl_rsp_t**, and is described in Response structure.

This command cannot be used in an ordered command group.

Security

To validate resource handles, the user of the calling application must have either **v** or **r** permission specified in an ACL entry for each resource class implied by the resource handles.

Return values

For the **mc_validate_rsrc_hndl_bp** and **mc_validate_rsrc_hndl_bc** subroutines, a return value of 0 indicates that the command has been successfully sent to the RMC subsystem and one or more responses have been received and processed.

For the **mc_validate_rsrc_hndl_ap** and **mc_validate_rsrc_hndl_ac** subroutines, a return value of 0 indicates that the command has been successfully added to the command group.

Any non-zero value returned by these subroutines is an error. The following errors can be returned by these subroutines. Additional error information can be returned by calling the **cu_get_error** function.

MC_ELIB
A severe library or system error occurred.

MC_ELIBNOMEM
A severe library memory allocation error occurred.

MC_EINVALIDCMD
The specified command group handle is invalid.

MC_EINVALIDRSRCCNT
Invalid resource handle count.

MC_ECMDGRPLIMIT
The command group already contains the maximum number of commands, as specified by the **MC_CMD_GRP_LIMIT** macro.

MC_EORDERGROUP
An attempt was made to add the command to an ordered command group.

MC_EINVALIDSESS
The specified session handle is invalid.

MC_ESESSENDED
The session has been ended.

MC_ESESSINTRPT
The session has been interrupted.

MC_ESENTENDED
The command has been sent but the session ended before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_ESENTINTRPT
The command has been sent but the session was interrupted before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.

MC_EAGAIN
Some system resource is not available. The application should try again later.

MC_EINVALIDRSPPTR
Invalid response pointer specified.

MC_EINVALIDCB
Invalid callback specified.

MC_ECMDTOOLARGE
The command is too large to send to the RMC subsystem.

MC_ECMDGRPSLIMIT
The maximum number of command groups are active.

MC_EINVALIDCNTPTR
Invalid response count pointer specified.

MC_ETIMEDOUT
The command has been sent to the RMC subsystem, but the command timeout limit (specified by the **mc_timed_start_session** subroutine when the session was established) was reached before all responses could be received. If subroutine uses pointer response, check the appropriate array count for any responses. If the subroutine uses callback response, the callback will have been invoked for any responses received.

MC_ENOMEM
The API could not allocate required memory.

Response structure

The response for these subroutines is a structure of type **mc_rsrc_hndl_rsp_t**. If multiple resource handles were identified using the *rsrc_hndl* parameter of the subroutine, then one response will be returned for each resource handle specified.

The response structure definition is:

```
typedef struct mc_rsrc_hndl_rsp         mc_rsrc_hndl_rsp_t;
struct mc_rsrc_hndl_rsp {
     mc_errnum_t                        mc_error;
     ct_resource_handle_t               mc_rsrc_hndl;
};
```

The fields of this structure contain the following:

mc_error
If 0 (zero), this field indicates that the command was successful. Any other value is an error. If there is an error, the error codes indicate whether the resource handle contained in *mc_rsrc_hndl* is invalid or the command could not be completed for the resource specified by the resource handle. The error may also indicate that the command arguments were in error.

mc_rsrc_hndl
The resource handle that identifies the resource that was the target of the command.

Location

/usr/lib/libct_mc.a

RMC API data definitions

The RMC API includes several macros and datatypes that simplify the task of programming the API, provide more complete error checking during compilation, and hide implementation details.

This information will help you understand the subroutine parameters described in "RMC API subroutines" on page 40.

The RMC API data declarations are contained in the header file **rsct/ct_mc.h**. The definitions in this header file are prefixed by **mc_**, in appropriate case. The **rsct/ct_mc.h** header file also includes the following header files:

ct_mcerr.h
Definitions of error codes, prefixed by **RMC_**, and descriptions of error arguments that can be returned in responses and event notifications. These error codes are summarized in "RMC API error codes and return values" on page 243.

ct_rmc.h
Definitions, prefixed by **rmc_**, that are common to several cluster APIs

ct_cu.h
Definitions, prefixed by **cu_**, for the cluster common utilities API.

ct.h Definitions, prefixed by **ct_**, that are common to all cluster APIs.

In order to use the version of the RMC API described here, the **rsct/ct_mc.h** header file must be preceded by the **MC_VERSION** macro definition, as shown in the following:

```
#define MC_VERSION 6
#include <rsct/ct_mc.h>
```

You can also, instead of defining the **MC_VERSION** and including the **rsct/ct_mc.h** header file, include the header file **rsct/ct_mc_v6.h**. This header file defines the **MC_VERSION** and includes the **rsct/ct_mc.h** header file.

RMC API macro definitions

The RMC API defines two macros to represent the maximum number of commands allowed in a command group, and the maximum number of command groups allowed in a session.

The macro definitions are:

```
#define MC_CMD_GRP_LIMIT 65535 /* max number of commands
                                  in a command group */

#define MC_CMD_GRPS_LIMIT 65535 /* max number of command
                                   groups in a session */
```

RMC API datatypes

The RMC API defines a number of datatypes that are summarized in the following table.

The error structure used in all response structures and event notification structures, is described in "Obtaining error information returned in response structures or event notification structures" on page 30. The following table lists the data types in alphabetical order and refers you to where you can obtain more information.

Table 26. RMC API datatypes

Data type	Description	For more information, see
***ct_array_ptr_t**	A pointer type to the composite type **ct_array_t**. This type is provided for convenience.	"RMC API data types for values, resources, and resource attributes" on page 238
ct_array_t	A structure used to specify an array of values.	"RMC API data types for values, resources, and resource attributes" on page 238
***ct_binary_ptr_t**	A pointer type to the composite type **ct_binary_t**. This type is provided for convenience.	"RMC API data types for values, resources, and resource attributes" on page 238
ct_binary_t	A structure used to specify a binary value.	"RMC API data types for values, resources, and resource attributes" on page 238
***ct_char_ptr_t**	Specifies a pointer to scalar value of type char.	"RMC API data types for values, resources, and resource attributes" on page 238
ct_char_t	Specifies a scalar value of type char.	"RMC API data types for values, resources, and resource attributes" on page 238
ct_contact_point_t	A union that identifies a node running the RMC subsystem.	"mc_start_session" on page 206 or "mc_timed_start_session" on page 212

Table 26. RMC API datatypes (continued)

Data type	Description	For more information, see
ct_contact_t	Used to specify one or more nodes the RMC API may contact to start a session with the RMC subsystem.	"mc_start_session" on page 206 or "mc_timed_start_session" on page 212
ct_contact_type_t	Enumeration that specifies supported contact types.	"mc_start_session" on page 206 or "mc_timed_start_session" on page 212
ct_data_type_t	An enumeration used to identify the data types, or pointers to data types, of values that are used by the RMC API.	"RMC API data types for values, resources, and resource attributes" on page 238
ct_float32_t	Specifies a scalar value of type float.	"RMC API data types for values, resources, and resource attributes" on page 238
ct_float64_t	Specifies a scalar value of type double.	"RMC API data types for values, resources, and resource attributes" on page 238
ct_int8_t	Specifies a scalar value of type **int8_t**.	"RMC API data types for values, resources, and resource attributes" on page 238
ct_int16_t	Specifies a scalar value of type **int16_t**.	"RMC API data types for values, resources, and resource attributes" on page 238
ct_int32_t	Specifies a scalar value of type **int32_t**.	"RMC API data types for values, resources, and resource attributes" on page 238
ct_int64_t	Specifies a scalar value of type **int64_t**.	"RMC API data types for values, resources, and resource attributes" on page 238
ct_IP_contact_point_t	A structure that identifies a contact point.	"mc_start_session" on page 206 or "mc_timed_start_session" on page 212
ct_uint8_t	Specifies a scalar value of type **uint8_t**.	"RMC API data types for values, resources, and resource attributes" on page 238
ct_uint16_t	Specifies a scalar value of type **uint16_t**.	"RMC API data types for values, resources, and resource attributes" on page 238
ct_uint32_t	Specifies a scalar value of type **uint32_t**.	"RMC API data types for values, resources, and resource attributes" on page 238
ct_uint64_t	Specifies a scalar value of type **uint64_t**.	"RMC API data types for values, resources, and resource attributes" on page 238
***ct_resource_handle_ptr_t**	A pointer type to the composite type **ct_resource_handle_t**. This type is provided for convenience.	"RMC API data types for values, resources, and resource attributes" on page 238
ct_resource_handle_t	A structure used to contain a resource handle	"RMC API data types for values, resources, and resource attributes" on page 238
ct_resource_id_t	A structure used to contain a resource ID.	"RMC API data types for values, resources, and resource attributes" on page 238
***ct_sd_ptr_t**	A pointer type to the composite type **ct_structured_data_t**. This type is provided for convenience.	"RMC API data types for values, resources, and resource attributes" on page 238
ct_structured_data_element_t	A structure used to define an element of Structured Data.	"RMC API data types for values, resources, and resource attributes" on page 238

Table 26. RMC API datatypes (continued)

Data type	Description	For more information, see
ct_structured_data_t	A structure used to define a Structured Data value.	"RMC API data types for values, resources, and resource attributes" on page 238
ct_value_t	A union used to return resource attribute values.	"RMC API data types for values, resources, and resource attributes" on page 238
mc_action_cb_t	Callback definition for the response callback specified on the variations of the **mc_invoke_action_*** subroutines that use callback response.	"mc_invoke_action_*" on page 78
mc_action_props_t	An enumeration that defines the properties on an action. Used in the response structure returned by the RMC API in response to a call to an **mc_qdef_actions_*** subroutine.	"mc_qdef_actions_*" on page 97
mc_action_rsp_t	A response structure returned by the RMC API in response to a call to an **mc_invoke_action_*** subroutine.	"mc_invoke_action_*" on page 78
mc_action_t	A structure that describes an action. Used in the response structure returned by the RMC API in response to a call to an **mc_qdef_actions_*** subroutine.	"mc_qdef_actions_*" on page 97
mc_action_type_t	An enumeration used to indicate the type of an action. Used in the response structure returned by the RMC API in response to a call to an **mc_qdef_valid_values_*** subroutine.	"mc_qdef_valid_values_*" on page 126
mc_attribute_t	A structure used to return a resource attribute.	"RMC API data types for values, resources, and resource attributes" on page 238
mc_class_action_cb_t	Callback definition for the response callback specified on the variations of the **mc_invoke_class_action_*** subroutines that use callback response.	"mc_invoke_class_action_*" on page 82
mc_class_action_rsp_t	A response structure returned by the RMC API in response to a call to an **mc_invoke_class_action_*** subroutine.	"mc_invoke_class_action_*" on page 82
mc_class_event_cb_t	Callback definition for the resource class event notification callback specified on calls to the **mc_reg_class_event_*** subroutine.	"mc_reg_class_event_*" on page 160
mc_class_event_t	Event notification structure used to notify an application of resource class events for which it registered using the **mc_reg_class_event_*** subroutine.	"mc_reg_class_event_*" on page 160
mc_class_name_rsp_t	A response structure returned by the RMC API in response to a call to an **mc_refresh_config_*** subroutine.	"mc_refresh_config_*" on page 156
mc_class_query_cb_t	Callback definition for the response callback specified on the variations of the **mc_class_query_d_*** and **mc_class_query_p_*** subroutines that use callback response.	"mc_class_query_d_*" on page 42 and "mc_class_query_p_*" on page 47
mc_class_query_rsp_t	A response structure returned by the RMC API in response to a call to an **mc_class_query_d_*** or **mc_class_query_p_*** subroutine.	"mc_class_query_d_*" on page 42 and "mc_class_query_p_*" on page 47
mc_class_set_cb_t	Callback definition for the response callback specified on the variations of the **mc_class_set_*** subroutines that use callback response.	"mc_class_set_*" on page 53
mc_class_set_rsp_t	A response structure returned by the RMC API in response to a call to an **mc_class_set_*** subroutine.	"mc_class_set_*" on page 53
mc_cmdgrp_hndl_t	An opaque data type used as a handle to represent command groups.	"RMC API opaque data types" on page 241
mc_cmd_grp_opts_t	An enumeration that defines options for allocating a command group that can be specified on the **mc_start_cmd_grp** subroutine.	"Start command group options" on page 242 and "mc_start_cmd_grp" on page 204

Table 26. RMC API datatypes (continued)

Data type	Description	For more information, see
mc_complete_cb_t	Callback definition for the completion callback specified on calls to the **mc_send_cmd_grp** subroutine.	"mc_send_cmd_grp" on page 188
mc_dattr_props_t	An enumeration that defines properties of a dynamic attribute. Used in the response structure returned by the RMC API in response to a call to an **mc_qdef_d_attribute_*** subroutine.	"mc_qdef_d_attribute_*" on page 102
mc_define_rsrc_rsp_t	A response structure returned by the RMC API in response to a call to an **mc_define_resource_*** subroutine.	"mc_define_resource_*" on page 58
mc_def_rsrc_cb_t	Callback definition for the response callback specified on the variations of the **mc_define_resource_*** subroutines that use callback response.	"mc_define_resource_*" on page 58
mc_dispatch_opts_t	An enumeration that defines dispatch options that can be specified on the **mc_dispatch** subroutine.	"Dispatch options" on page 242 and "mc_dispatch" on page 62
mc_enumerate_cb_t	Callback definition for the response callback specified on the variations of the **mc_enumerate_resources_***, and **mc_enumerate_permitted_rsrcs_*** subroutines that use callback response.	"mc_enumerate_resources_*" on page 69 and "mc_enumerate_permitted_rsrcs_*" on page 65
mc_enumerate_rsp_t	A response structure returned by the RMC API in response to a call to an **mc_enumerate_resources_*** or **mc_enumerate_permitted_rsrcs_*** subroutine.	"mc_enumerate_resources_*" on page 69 and "mc_enumerate_permitted_rsrcs_*" on page 65
mc_errnum_t	A structure used to contain error information. A structure of this type is contained within all response structures and event notifications.	"Obtaining error information returned in response structures or event notification structures" on page 30
mc_error_attr_t	A structure that describes attribute that could not be set. Used in the response structure returned by the RMC API in response to a call to an **mc_class_set_***, **mc_set_handle_***, or **mc_set_select_*** subroutine.	"mc_class_set_*" on page 53, "mc_set_handle_*" on page 194, and "mc_set_select_*" on page 199
mc_event_cb_t	Callback definition for the resource event notification callback specified on calls to the **mc_reg_event_handle_*** and **mc_reg_event_select_*** subroutines.	"mc_reg_event_handle_*" on page 168 and "mc_reg_event_select_*" on page 176
mc_event_flags_t	An enumeration that defines event flags returned in event notification structures.	"mc_reg_class_event_*" on page 160, "mc_reg_event_handle_*" on page 168, and "mc_reg_event_select_*" on page 176
mc_event_t	Event notification structure, used to notify an application of resource events for which it registered using the **mc_reg_event_handle_*** or **mc_reg_event_select_*** subroutines.	"mc_reg_event_handle_*" on page 168 and "mc_reg_event_select_*" on page 176
mc_list_usage_t	An enumeration that defines options used by the **mc_invoke_class_action_*** subroutines.	"Name list usage" on page 243 and "mc_invoke_class_action_*" on page 82
mc_offline_cb_t	Callback definition for the response callback specified on the variations of the **mc_offline_*** subroutines that use callback response.	"mc_offline_*" on page 88
mc_offline_opts_t	An enumeration that defines options for taking a resource offline. These options are used by the **mc_offline_*** subroutines.	"Offline command options" on page 243 and "mc_offline_*" on page 88
mc_online_cb_t	Callback definition for the response callback specified on the variations of the **mc_online_*** subroutines that use callback response.	"mc_online_*" on page 92
mc_pattr_props_t	An enumeration that defines properties of a persistent attribute. Used in the response structure returned by the RMC API in response to a call to an **mc_qdef_p_attribute_*** subroutine.	"mc_qdef_p_attribute_*" on page 108

Table 26. RMC API datatypes (continued)

Data type	Description	For more information, see
mc_qdef_actions_cb_t	Callback definition for the response callback specified on the variations of the **mc_qdef_actions_*** subroutines that use callback response.	"mc_qdef_actions_*" on page 97
mc_qdef_actions_rsp_t	A response structure returned by the RMC API in response to a call to an **mc_qdef_actions_*** subroutine.	"mc_qdef_actions_*" on page 97
mc_qdef_dattr_cb_t	Callback definition for the response callback specified on the variations of the **mc_qdef_d_attribute_*** subroutines that use callback response.	"mc_qdef_d_attribute_*" on page 102
mc_qdef_dattr_rsp_t	A response structure returned by the RMC API in response to a call to an **mc_qdef_d_attribute_*** subroutine.	"mc_qdef_d_attribute_*" on page 102
mc_qdef_opts_t	An enumeration that defines query definition options for the **mc_qdef_actions_***, **mc_qdef_d_attribute_***, **mc_qdef_p_attribute_***, **mc_qdef_resource_class_***, **mc_qdef_sd_***, and **mc_qdef_valid_values_*** subroutines.	"Query definition command options" on page 243
mc_qdef_pattr_cb_t	Callback definition for the response callback specified on the variations of the **mc_qdef_p_attribute_*** subroutines that use callback response.	"mc_qdef_p_attribute_*" on page 108
mc_qdef_pattr_rsp_t	A response structure returned by the RMC API in response to a call to an **mc_qdef_p_attribute_*** subroutine.	"mc_qdef_p_attribute_*" on page 108
mc_qdef_rsrc_class_cb_t	Callback definition for the response callback specified on the variations of the **mc_qdef_resource_class_*** subroutines that use callback response.	"mc_qdef_resource_class_*" on page 114
mc_qdef_rsrc_class_rsp_t	A response structure returned by the RMC API in response to a call to an **mc_qdef_resource_class_*** subroutine.	"mc_qdef_resource_class_*" on page 114
mc_qdef_sd_cb_t	Callback definition for the response callback specified on the variations of the **mc_qdef_sd_*** subroutines that use callback response.	"mc_qdef_sd_*" on page 120
mc_qdef_sd_rsp_t	A response structure returned by the RMC API in response to a call to an **mc_qdef_sd_*** subroutine.	"mc_qdef_sd_*" on page 120
mc_qdef_valid_vals_cb_t	Callback definition for the response callback specified on the variations of the **mc_qdef_valid_values_*** subroutines that use callback response.	"mc_qdef_valid_values_*" on page 126
mc_qdef_valid_vals_rsp_t	A response structure returned by the RMC API in response to a call to an **mc_qdef_valid_values_*** subroutine.	"mc_qdef_valid_values_*" on page 126
mc_query_cb_t	Callback definition for the response callback specified on the variations of the **mc_query_d_handle_***, **mc_query_d_select_***, **mc_query_p_handle_***, and **mc_query_p_select_*** subroutines that use callback response.	"mc_query_d_handle_*" on page 133, "mc_query_d_select_*" on page 138, "mc_query_p_handle_*" on page 147, and "mc_query_p_select_*" on page 151
mc_query_event_cb_t	Callback definition for the response callback specified on the variations of the **mc_query_event_*** subroutines that use callback response.	"mc_query_event_*" on page 143
mc_query_event_rsp_t	A response structure returned by the RMC API in response to a call to an **mc_query_event_*** subroutine.	"mc_query_event_*" on page 143
mc_query_rsp_t	A response structure returned by the RMC API in response to a call to an **mc_query_d_handle_***, **mc_query_d_select_***, **mc_query_p_handle_***, or **mc_query_p_select_*** subroutine.	"mc_query_d_handle_*" on page 133, "mc_query_d_select_*" on page 138, "mc_query_p_handle_*" on page 147, and "mc_query_p_select_*" on page 151
mc_refresh_cfg_cb_t	Callback definition for the response callback specified on the variations of the **mc_refresh_config_*** subroutines that use callback response.	"mc_refresh_config_*" on page 156

Table 26. RMC API datatypes (continued)

Data type	Description	For more information, see
mc_reg_cb_t	Callback definition for the response callback specified on the variations of the **mc_reg_class_event_***, **mc_reg_event_handle_***, and **mc_reg_event_select_*** subroutines that use callback response.	"mc_reg_class_event_*" on page 160, "mc_reg_event_handle_*" on page 168, and "mc_reg_event_select_*" on page 176
mc_registration_id_t	An opaque data type. The data type of an event registration ID.	"RMC API opaque data types" on page 241
mc_reg_opts_t	An enumeration that defines event registration options for the **mc_reg_class_event_***, **mc_reg_event_handle_***, and **mc_reg_event_select_*** subroutines.	"Event registration command options" on page 242, "mc_reg_class_event_*" on page 160, "mc_reg_event_handle_*" on page 168, and "mc_reg_event_select_*" on page 176
mc_reg_rsp_t	A response structure returned by the RMC API in response to a call to an **mc_reg_class_event_***, **mc_reg_event_handle_***, or **mc_reg_event_select_*** subroutine.	"mc_reg_class_event_*" on page 160, "mc_reg_event_handle_*" on page 168, and "mc_reg_event_select_*" on page 176
mc_reset_cb_t	Callback definition for the response callback specified on the variations of the **mc_reset_*** subroutines that use callback response.	"mc_reset_*" on page 184
mc_rsrc_class_props_t	An enumeration that defines properties of a resource class. Used in the response structure returned by the RMC API in response to a call to an **mc_qdef_resource_class_*** subroutine.	"mc_qdef_resource_class_*" on page 114
mc_rsrc_hndl_rsp_t	A response structure returned by the RMC API in response to a call to an **mc_offline_***, **mc_online_***, **mc_reset_***, or **mc_validate_rsrc_hndl_*** subroutine.	"mc_offline_*" on page 88, "mc_online_*" on page 92, "mc_reset_*" on page 184, "mc_validate_rsrc_hndl_*" on page 227
mc_rsrc_mgr_t	A structure used to specify a resource manager that supports a particular resource class. Used in the response structure returned by the RMC API in response to a call to an **mc_qdef_resource_class_*** subroutine.	"mc_qdef_resource_class_*" on page 114
mc_sd_element_t	A structure that describes an individual Structured Data element. Used in the response structure returned by the RMC API in response to a call to an **mc_qdef_sd_*** subroutine.	"mc_qdef_sd_*" on page 120
mc_sd_usage_t	An enumeration that defines how Structured Data is used. Used as input to the **mc_qdef_sd_*** subroutines to specify the kind of Structured Data information to be returned. Also used in the response structure returned by the RMC API in response to a call to an **mc_qdef_sd_*** subroutine.	"mc_qdef_sd_*" on page 120
mc_sess_info_t	An enumeration that specifies what type of session information is requested: **MC_SESS_INFO_LOCAL_IP_ADDRESS**.	"mc_session_info" on page 193
mc_session_hndl_t	An opaque datatype used as a handle to represent RMC sessions.	"RMC API opaque data types" on page 241
mc_session_opts_t	An enumeration that defines session options that can be specified on the **mc_start_session** or **mc_timed_start_session** subroutine.	"Start session options" on page 241. "mc_start_session" on page 206 or "mc_timed_start_session" on page 212
mc_set_cb_t	Callback definition for the response callback specified on the variations of the **mc_set_handle_*** and **mc_set_select_*** subroutines that use callback response.	"mc_set_handle_*" on page 194 and "mc_set_select_*" on page 199

Table 26. RMC API datatypes (continued)

Data type	Description	For more information, see
mc_set_rsp_t	A response structure returned by the RMC API in response to a call to an **mc_set_handle_*** or **mc_set_select_*** subroutine.	"mc_set_handle_*" on page 194 and "mc_set_select_*" on page 199
mc_undefine_rsrc_rsp_t	A response structure returned by the RMC API in response to a call to an **mc_undefine_resource_*** subroutine.	"mc_undefine_resource_*" on page 219
mc_undef_rsrc_cb_t	Callback definition for the response callback specified on the variations of the **mc_undefine_resource_*** subroutines that use callback response.	"mc_undefine_resource_*" on page 219
mc_unreg_cb_t	Callback definition for the response callback specified on the variations of the **mc_unreg_event_*** subroutines that use callback response.	"mc_unreg_event_*" on page 223
mc_unreg_rsp_t	A response structure returned by the RMC API in response to a call to an **mc_unreg_event_*** subroutine.	"mc_unreg_event_*" on page 223
mc_valid_value_t	A structure used to describe an individual valid value. Used in the response structure returned by the RMC API in response to a call to an **mc_qdef_valid_values_*** subroutine.	"mc_qdef_valid_values_*" on page 126
mc_val_rsrc_hndl_cb_t	Callback definition for the response callback specified on the variations of the **mc_validate_rsrc_hndl_*** subroutines that use callback response.	"mc_define_resource_*" on page 58
mc_vv_usage_t	An enumeration that defines the possible applications of valid values. Used as input to the **mc_qdef_valid_values_*** subroutines to specify the kind of valid value information to be returned. Also used in the response structure returned by the RMC API in response to a call to an **mc_qdef_valid_values_*** subroutine.	"mc_qdef_valid_values_*" on page 126
rmc_attribute_id_t	A datatype for attribute IDs.	"RMC API data types for values, resources, and resource attributes"
rmc_opstate_t	An enumeration that is used to specify the possible values of the OpState resource dynamic attribute defined in many resource classes.	"RMC API data types for values, resources, and resource attributes"
rmc_variable_type_t	Enumeration that identifies the variable types of dynamic resource attributes that are returned by the RMC API.	"RMC API data types for values, resources, and resource attributes"

RMC API data types for values, resources, and resource attributes:

This topic addresses RMC API data types for values, resources, and resource attributes.

The **ct_data_type_t** enumeration is used to identify the data types, or pointers to data types, of values that are used by the RMC API.

```
typedef enum {
     CT_UNKNOWN = 0,
     CT_NONE,                        /* for Quantum variables only */
     CT_INT32,
     CT_UINT32,
     CT_INT64,
     CT_UINT64,
     CT_FLOAT32,
     CT_FLOAT64,
     CT_CHAR_PTR,                    /* pointer to NULL terminated string*/
     CT_BINARY_PTR,                  /* pointer to ct_binary_t */
     CT_RSRC_HANDLE_PTR,             /* pointer to ct_resource_handle_t */
     CT_SD_PTR,                      /* pointer to ct_structured_data_t */
     CT_INT32_ARRAY,                 /* CT_INT32 array */
     CT_UINT32_ARRAY,                /* CT_UINT32 array */
     CT_INT64_ARRAY,                 /* CT_INT64 array */
     CT_UINT64_ARRAY,                /* CT_UINT64 array */
```

```
     CT_FLOAT32_ARRAY,              /* CT_FLOAT32 array */
     CT_FLOAT64_ARRAY,              /* CT_FLOAT64 array */
     CT_CHAR_PTR_ARRAY,             /* CT_CHAR_PTR array */
     CT_BINARY_PTR_ARRAY,           /* CT_BINARY_PTR array */
     CT_RSRC_HANDLE_PTR_ARRAY       /* CT_RSRC_HANLE_PTR array */
     CT_SD_PTR_ARRAY,               /* CT_SD_PTR array */
} ct_data_type_t;
```

The **rmc_variable_type_t** enumeration is used to identify the variable types of dynamic resource attributes that are returned by the RMC API:

```
typedef enum {
     RMC_COUNTER,
     RMC_QUANTITY,
     RMC_STATE,
     RMC_QUANTUM
} rmc_variable_type_t;
```

The **rmc_opstate_t** enumeration is used to specify the possible values of the **OpState** resource dynamic attribute, defined in many resource classes:

```
typedef enum {
     RMC_OPSTATE_UNKNOWN = 0,
     RMC_OPSTATE_ONLINE,
     RMC_OPSTATE_OFFLINE,
     RMC_OPSTATE_FAILED_OFFLINE,
     RMC_OPSTATE_STUCK_ONLINE,
     RMC_OPSTATE_PENDING_ONLINE,
     RMC_OPSTATE_PENDING_OFFLINE,
     RMC_OPSTATE_MIXED
} rmc_opstate_t;
```

Scalar values are specified using one of the following types:

```
typedef int8_t              ct_int8_t;
typedef uint8_t             ct_uint8_t;
typedef int16_t             ct_int16_t;
typedef uint16_t            ct_uint16_t;
typedef int32_t             ct_int32_t;
typedef uint32_t            ct_uint32_t;
typedef int64_t             ct_int64_t;
typedef uint64_t            ct_uint64_t;
typedef float               ct_float32_t;
typedef double              ct_float64_t;
typedef char                ct_char_t;
typedef char                *ct_char_ptr_t;
```

A **ct_binary_t** type is used to specify a binary value. Note that the structure only defines a binary type of length 1. Otherwise, the type is used to overlay a buffer of appropriate size to contain the desired binary value. If the length field is zero, a NULL binary value is indicated. If such is the case, the first element of the data array should **not** be assumed to be addressable.

```
typedef struct {
     ct_uint32_t     length;
     ct_char_t       data[1];
} ct_binary_t;
```

A **ct_structured_data_element_t** type is used to define an element of Structured Data.

```
typedef struct {
     ct_data_type_t    data_type;
     ct_value_t        value;
} ct_structured_data_element_t;
```

Note that *data_type* can be any value taken from **ct_data_type_t** except **CT_SD_PTR** and **CT_SD_PTR_ARRAY**. In other words, a SD element can be any type other than Structured Data or Structured Data Array.

A **ct_structured_data_t** type is used to define a Structured Data value.

```
typedef struct {
     ct_uint32_t                     element_count;
     ct_structured_data_element_t    elements[1];
} ct_structured_data_t;
```

Note that the structure only defines an array with one array element. Otherwise, the type is used to overlay a buffer of appropriate size to contain the desired number of elements. If the *element_count* field is **0**, a Null array is indicated. If such is the case, the first element of the array should **not** be assumed to be addressable.

A **ct_resource_id_t** structure is used to contain a resource ID. A resource ID is a resource class specific identifier of a resource. The resource ID is considered an opaque type to the user of the RMC API.

```
typedef struct {
     ct_uint32_t id1;
     ct_uint32_t id2;
     ct_uint32_t id3;
     ct_uint32_t id4;
} ct_resource_id_t;
```

A **ct_resource_handle_t** structure is used to contain a resource handle. A resource handle is a cluster unique, persistent identifier of a resource. The resource handle is considered an opaque type.

```
typedef struct {
     ct_uint32_t             header;
     ct_resource_id_t        id;
} ct_resource_handle_t;
```

For convenience, pointer types to the composite types are also defined:

```
typedef ct_binary_t             *ct_binary_ptr_t;
typedef ct_resource_handle_t    *ct_resource_handle_ptr_t;
typedef ct_structured_data_t    *ct_sd_ptr_t;
typedef ct_array_t              *ct_array_ptr_t;
```

Resource attribute values are returned in the **ct_value_t** type. This type provides a uniform representation of the preceding value types. The actual type of the value contained in the **ct_value_t** is specified by a value from the **ct_data_type_t** enumeration.

```
typedef union {
     ct_int32_t                      val_int32;
     ct_uint32_t                     val_uint32;
     ct_int64_t                      val_int64;
     ct_uint64_t                     val_uint64;
     ct_float32_t                    val_float32;
     ct_float64_t                    val_float64;
     ct_char_ptr_t                   ptr_char;
     ct_binary_ptr_t                 ptr_binary;
     ct_resource_handle_ptr_t        ptr_rsrc_handle;
     ct_sd_ptr_t                     ptr_sd;
     ct_array_ptr_t                  ptr_array;
} ct_value_t;
```

Note that for the character string, binary, resource handle, Structured Data and array data types, the value returned is actually a pointer to the value.

A **ct_array_t** type is used to specify an array of values.

```
typedef struct {
     ct_uint32_t element_count;
     ct_value_t elements[1];
} ct_array_t;
```

Note that the structure only defines an array with one array element. Otherwise, the type is used to overlay a buffer of appropriate size to contain the desired number of elements. If the *element_count* field is zero, a NULL array is indicated. If such is the case, the first element of the array should NOT be assumed to be addressable.

When a **ct_value_t** has a type from the **ct_data_type_t** enumeration of the form **CT_ type_ARRAY**, then the *ptr_array* field is to be used; the type of the elements in the **ct_array_t** are then **CT_ type**.

A resource attribute is returned in a **mc_attribute_t** type. his type includes the (programmatic) attribute name, ID and data type. This type is also used to provide attributes to the RMC API. When provided to the RMC API, the *mc_at_id* field is not defined and the *mc_at_dtype* field must be set appropriately.

```
typedef struct mc_attribute mc_attribute_t;
struct mc_attribute {
     ct_char_t               *mc_at_name;
     rmc_attribute_id_t      mc_at_id;
     ct_data_type_t          mc_at_dtype;
     ct_value_t              mc_at_value;
};
```

A response or event notification contains a pointer to an array of type **mc_attribute_t** and a count of elements in the array. The application can examine this array directly.

The **rmc_attribute_id_t** datatype is defined as follows:

```
typedef ct_int32_t rmc_attribute_id_t;
```

RMC API opaque data types:

The RMC API defines two opaque data types used as handles.

One is used to identify a session and the other used to identify a command group.

```
typedef void *mc_sess_hndl_t;         /* session handle */

typedef void *mc_cmdgrp_hndl_t;       /* command group handle */
```

The command subroutines take either a session handle or command group handle as their first argument.

The RMC API defines a registration ID as an opaque data type.

```
typedef void *mc_registration_id_t;    /* registration ID */
```

RMC API enumerations to define subroutine options:

Enumerations define options to several of the RMC API subroutines.

These options are described in the corresponding subroutine descriptions in “RMC API subroutines” on page 40.

Start session options:

The **mc_session_opts_t** enumeration defines session options that can be specified on the **mc_start_session** or **mc_timed_start_session** subroutine.

See “mc_start_session” on page 206 or “mc_timed_start_session” on page 212 for a description of each of these options.

```
typedef enum mc_session_opts    mc_session_opts_t;
enum mc_session_opts {
     MC_SESSION_OPTS_NONE                = 0x0000,
     MC_SESSION_OPTS_LOCAL_SCOPE         = 0x0001,
     MC_SESSION_OPTS_SR_SCOPE            = 0x0002,
     MC_SESSION_OPTS_DM_SCOPE            = 0x0004,
     MC_SESSION_OPTS_PRIVATE             = 0x0008,
     MC_SESSION_OPTS_SR_LOCAL_SCOPE      = 0x0010,
     MC_SESSION_OPTS_DM_LOCAL_SCOPE      = 0x0020,
     MC_SESSION_OPTS_SR_DM_SCOPE         = 0x0040,
     MC_SESSION_OPTS_DM_SR_SCOPE         = 0x0080,
     MC_SESSION_OPTS_SR_DM_LOCAL_SCOPE   = 0x0100,
     MC_SESSION_OPTS_DM_SR_LOCAL_SCOPE   = 0x0200,
     MC_SESSION_OPTS_IP_AUTHENTICATION   = 0x0400
};
```

Dispatch options:

The **mc_dispatch_opt_t** enumeration defines dispatch options that can be specified on the **mc_dispatch** subroutine.

See “mc_dispatch” on page 62 for a description of each of these options

```
typedef enum mc_dispatch_opts  mc_dispatch_opts_t;
enum mc_dispatch_opts {
     MC_DISPATCH_OPTS_WAIT,
     MC_DISPTACH_OPTS_ASSIGN
};
```

Start command group options:

The **mc_cmd_grp_opts_t** enumeration defines options for allocating a command group that can be specified on the **mc_start_cmd_grp** subroutine.

See “mc_start_cmd_grp” on page 204 for a description of each of these options.

```
typedef enum mc_cmd_grp_opts   mc_cmd_grp_opts_t;
enum mc_cmd_grp_opts {
     MC_CMD_GRP_OPTS_NONE              = 0x0000,
     MC_CMD_GRP_OPTS_ORDERED           = 0x0001,
};
```

Event registration command options:

The **mc_reg_opts_t** enumeration defines event registration options for the **mc_reg_class_event_***, **mc_reg_event_handle_***, and **mc_reg_event_select_*** subroutines. The MC_REG_OPTS_KEEP_REG constant is used only with the **mc_reg_event_select_*** subroutines.

See “mc_reg_class_event_*” on page 160, “mc_reg_event_handle_*” on page 168, or “mc_reg_event_select_*” on page 176 for a description of these options.

```
typedef enum mc_reg_opts mc_reg_opts_t;
enum mc_reg_opts {
     MC_REG_OPTS_NONE                  = 0x0000,
     MC_REG_OPTS_NO_REG                = 0x0001,
     MC_REG_OPTS_IMMED_EVAL            = 0x0002,
     MC_REG_OPTS_REARM_EVENT           = 0x0004,
     MC_REG_OPTS_KEEP_REG              = 0x0008
     MC_REG_NO_TOGGLE                  = 0x0010
};
```

Query definition command options:

The **mc_qdef_opts_t** enumeration defines query definition options for the **mc_qdef_actions_***, **mc_qdef_d_attribute_***, **mc_qdef_p_attribute_**, **mc_qdef_resource_class_***, **mc_qdef_sd_***, and **mc_qdef_valid_values_*** subroutines.

For a description of these options, see the reference information on any of these subroutines in "RMC API subroutines" on page 40.

```
typedef enum mc_qdef_opts mc_qdef_opts_t;
enum mc_qdef_opts {
     MC_QDEF_OPTS_NONE               = 0x0000,
     MC_QDEF_OPTS_NODSCRP            = 0x0001
};
```

Offline command options:

The **mc_offline_opts_t** enumeration defines options for taking a resource offline. These options are used by the **mc_offline_*** subroutines.

See "mc_offline_*" on page 88 for a description of each of these options.

```
typedef enum mc_offline_opts mc_offline_opts_t;
enum mc_offline_opts {
     MC_OFFLINE_OPTS_NONE            = 0x0000,
     MC_OFFLINE_OPTS_FAILED          = 0x0001
};
```

Name list usage:

The **mc_list_usage_t** enumeration defines options used by the **mc_invoke_class_action_*** subroutines.

See "mc_invoke_class_action_*" on page 82 for a description of each of these options.

```
typedef enum mc_list_usage mc_list_usage_t;
enum mc_list_usage {
     MC_LIST_USAGE_NODES             = 0x0000,
     MC_LIST_USAGE_PEER_DOMAINS      = 0x0001
};
```

RMC API error codes and return values

Errors can be detected by an RMC API subroutine, an RMC subsystem daemon, or a resource manager.

As described in "Notifying the application of errors" on page 29, errors can be detected by an RMC API subroutine, an RMC subsystem daemon, or a resource manager.

- When an RMC API subroutine detects an error, the subroutine returns immediately with an error code. These error codes are listed in "RMC API errors" on page 244.
- When an RMC subsystem daemon or a resource manager detects an error, the response structure or event notification structure will contain an error structure that provides an error code. These error codes are listed in "Response and event structure error codes" on page 247.

Related concepts:

"Notifying the application of errors" on page 29
Errors can be detected in various ways when using the RMC API.

"Obtaining error information returned by the RMC API subroutines" on page 29
All RMC API subroutines return a value of type **ct_int32_t**. A return value of 0 indicates that the subroutine completed successfully. Any non-zero value is an error value.

"Obtaining error information returned in response structures or event notification structures" on page 30
All response structures and event notification structures contain error information.

RMC API errors

The following two tables describe the error codes that may be returned by an RMC API subroutine.

This first table lists the errors by error code. These same errors are also listed alphabetically by return value in Table 28 on page 245. As described in "Obtaining error information returned by the RMC API subroutines" on page 29, an application can also use the cluster common subroutines **cu_get_error** and **cu_get_errmsg** to obtain more information about an error. For complete syntax information on these subroutines, see "Cluster utilities: error-related subroutines" on page 264.

Table 27. RMC API errors (listed by error code)

Error code	Return value	Description
1	MC_ELIB	A severe library or system error occurred.
2	MC_ESESSREFUSED	A session could not be established with the RMC subsystem in the specified cluster. The application should try again later.
3	MC_ESESSINTRPT	The session has been interrupted.
4	MC_ESESSENDED	The session has been ended.
5	MC_EINVALIDSESS	The specified session handle is invalid.
6	MC_ENODSCRP	A descriptor cannot be allocated by the API.
7	MC_EINVALIDDSCRP	The specified descriptor is invalid.
8	MC_EINVALIDCMD	The specified command group handle is invalid.
9	MC_ENOCMDS	The command group contains no commands.
10	MC_ESENTENDED	The command has been sent but the session ended before all responses could be received. If pointer response was selected for the command, check the appropriate pointer or array count, if defined, for any responses. If callback response was selected for the command, the callback was invoked for any responses received.
11	MC_ESENTINTRPT	The command has been sent but the session was interrupted before all responses could be received. If pointer response was selected for the command, check the appropriate pointer or array count, if defined, for any responses. If callback response was selected for the command, the callback was invoked for any responses received.
12	MC_EINVALIDDATA	Invalid response or event notification structure.
13	MC_ECMDGRPSLIMIT	The maximum number of command groups are active for the session.
14	MC_ECMDGRPLIMIT	The command group already contains the maximum number of commands, as specified by the MC_CMD_GRP_LIMIT macro.
15	MC_EINVALIDEID	The specified registration ID is invalid.
16	MC_EINVALIDOPT	The specified option is invalid.
17	MC_EBUSY	Function busy. When returned by the **mc_get_descriptor** subroutine, indicates that a descriptor has already been allocated to the specified session.
18	MC_ENOMEM	The API could not allocate required memory.
19	MC_EAGAIN	Some system resource is not available. The application should try again later.
20	MC_EINVALIDCB	Invalid callback routine specified.
21	MC_EINVALIDRSPPTR	Invalid response pointer specified.
22	MC_EINVALIDCNTPTR	Invalid response count pointer specified.
23	MC_EINVALIDRSRCCNT	Invalid resource handle count.
24	MC_ECMDTOOLARGE	The command is too large to send to the RMC subsystem.
25	MC_ELIBNOMEM	A severe library memory allocation error occurred.
26	MC_EORDERGROUP	An attempt was made to add the command to an ordered command group.

Table 27. RMC API errors (listed by error code) (continued)

Error code	Return value	Description
27	MC_ETARGETMISMATCH	The target specified for the command does not match the target of the command group.
28	MC_EINVALIDDATATYPE	Invalid attribute data type specified.
29	MC_EINVALIDVALUEPTR	Invalid attribute value pointer specified.
30	MC_EINVALIDSBSLEN	Invalid structured byte string len.
31	MC_EINVALIDSDTYPE	Invalid structured data subtype specified.
32	MC_EINVALIDCONTACT	Invalid contact type specified.
33	MC_ENOTSUPPORTED	Called function is not supported.
34	MC_EDEADLOCK	An attempt was made to invoke the command from within an event notification callback.
35	MC_EINVALIDNAME	Invalid contact point name.
36	MC_EINVALIDSTRING	Invalid string
37	MC_ELIBSECURITY	The RMC API detected an error in security services.
38	MC_ESUBSECURITY	The RMC subsystem detected an error in security services.
39	MC_EAUTHENTICATE	Could not authenticate the user of the calling application.
40	MC_EAUTHORIZATION	User not authorized to use RMC.
41	MC_EMUTUALAUTHENT	The RMC API could not authenticate the RMC subsystem.
42	MC_EVERSIONMISMATCH	RMC API version mismatch.
43	MC_EINVALSCOPE	The specified session scope is not currently supported.
44	MC_EMULTISCOPE	Multiple session scope options are specified.
45	MC_ENOTPRIVATESCOPE	The specified session scope cannot be combined with the private option.
46	MC_EQEVENTACTIVE	A query event command has not yet completed.
47	MC_ENOSUPPORTRTNVER	No support for routine in this version of RMC.
48	MC_ENOSUPPORTARGVER	No support for arguments in this version of RMC.
49	MC_ECLASSEID	The registration ID specified a class event.
50	MC_ENOIPADDRCONTACT	Contact must be an IP address.
51	MC_ENOIPAUTHENTSUPPORT	IP authentication not supported by session.
52	MC_ETIMEDOUT	Start session, or command, timed out.

The following table lists the error codes that may be returned by an RMC API subroutine alphabetically by return value. These same errors are also listed numerically by error code in Table 27 on page 244.

Table 28. RMC API errors (listed by return value)

Return value	Error code	Description
MC_EAGAIN	19	Some system resource is not available. The application should try again later.
MC_EAUTHENTICATE	39	Could not authenticate the user of the calling application.
MC_EAUTHORIZATION	40	User not authorized to use RMC.
MC_EBUSY	17	Function busy. When returned by the **mc_get_descriptor** subroutine, indicates that a descriptor has already been allocated to the specified session.
MC_ECLASSEID	49	The registration ID specified a class event.
MC_ECMDGRPLIMIT	14	The command group already contains the maximum number of commands, as specified by the MC_CMD_GRP_LIMIT macro.
MC_ECMDGRPSLIMIT	13	The maximum number of command groups are active for the session.
MC_ECMDTOOLARGE	24	The command is too large to send to the RMC subsystem.

Table 28. RMC API errors (listed by return value) (continued)

Return value	Error code	Description
MC_EDEADLOCK	34	An attempt was made to invoke the command from within an event notification callback.
MC_EINVALIDCB	20	Invalid callback routine specified.
MC_EINVALIDCMD	8	The specified command group handle is invalid.
MC_EINVALIDCONTACT	32	Invalid contact type specified.
MC_EINVALIDDATA	12	Invalid response or event notification structure.
MC_EINVALIDDATATYPE	28	Invalid attribute data type specified.
MC_EINVALIDDSCRP	7	The specified descriptor is invalid.
MC_EINVALIDEID	15	The specified registration ID is invalid.
MC_EINVALIDNAME	35	Invalid contact point name.
MC_EINVALIDOPT	16	The specified option is invalid.
MC_EINVALIDRSPPTR	21	Invalid response pointer specified.
MC_EINVALIDRSRCCNT	23	Invalid resource handle count.
MC_EINVALIDSBSLEN	30	Invalid structured byte string len.
MC_EINVALIDSDTYPE	31	Invalid structured data subtype specified.
MC_EINVALIDSESS	5	The specified session handle is invalid.
MC_EINVALIDSTRING	36	Invalid string
MC_EINVALIDVALUEPTR	29	Invalid attribute value pointer specified.
MC_EINVALSCOPE	43	The specified session scope is not currently supported.
MC_ELIB	1	A severe library or system error occurred.
MC_ELIBNOMEM	25	A severe library memory allocation error occurred.
MC_ELIBSECURITY	37	The RMC API detected an error in security services.
MC_EMULTISCOPE	44	Multiple session scope options are specified.
MC_EMUTUALAUTHENT	41	The RMC API could not authenticate the RMC subsystem.
MC_ENOCMDS	9	The command group contains no commands.
MC_ENODSCRP	6	A descriptor cannot be allocated by the API.
MC_ENOIPADDRCONTACT	50	Contact must be an IP address.
MC_ENOIPAUTHENTSUPPORT	51	IP authentication not supported by session.
MC_ENOMEM	18	The API could not allocate required memory.
MC_ENOSUPPORTARGVER	48	No support for arguments in this version of RMC.
MC_ENOSUPPORTRTNVER	47	No support for routine in this version of RMC.
MC_ENOTPRIVATESCOPE	45	The specified session scope cannot be combined with the private option.
MC_ENOTSUPPORTED	33	Called function is not supported.
MC_EORDERGROUP	26	An attempt was made to add the command to an ordered command group.
MC_EQEVENTACTIVE	46	A query event command has not yet completed.
MC_ESENTENDED	10	The command has been sent but the session ended before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.
MC_ESENTINTRPT	11	The command has been sent but the session was interrupted before all responses could be received. If pointer response was selected for the command, check the appropriate array count for any responses. If callback response was selected for the command, the callback was invoked for any responses received.
MC_ESESSENDED	4	The session has been ended.

Table 28. RMC API errors (listed by return value) (continued)

Return value	Error code	Description
MC_ESESSINTRPT	3	The session has been interrupted.
MC_ESESSREFUSED	2	A session could not be established with the RMC subsystem in the specified cluster. The application should try again later.
MC_ESUBSECURITY	38	The RMC subsystem detected an error in security services.
MC_ETARGETMISMATCH	27	The target specified for the command does not match the target of the command group.
MC_ETIMEDOUT	52	Start session, or command, timed out.
MC_EVERSIONMISMATCH	42	RMC API version mismatch.

Response and event structure error codes

All response structures and event notification structures contain error information in a structure of type **mc_errnum_t**.

As described in "Obtaining error information returned in response structures or event notification structures" on page 30, all response structures and event notification structures contain error information in a structure of type **mc_errnum_t**. Error codes that may be returned are grouped in ranges of 64K, starting with the lowest values in each group. Each range represents errors of a similar nature. The high-order sixteen bits of the error code identify the error group or general class of error, and the lower sixteen bits indicate a specific error within the group. The application can identify the error group using the MC_GET_GENERR macro.

The error groups are:

Table 29. Error groups

Error code value	Error group	For information of the errors in this group, see:
0x10000 thru 0x1ffff	Errors common to all resource managers	The error message returned in the **mc_errnum_t** structure.
0x30000 thru 0x3ffff	Command specification	"Command specification errors"
0x40000 thru 0x4ffff	Resource access or usage	"Resource access or usage errors" on page 255
0x50000 thru 0x5ffff	Expression specification	"Expression specification errors" on page 262
0x60000 thru 0x6ffff	Select string specification	"Select string specification errors" on page 263

Related concepts:

"Obtaining error information returned in response structures or event notification structures" on page 30
All response structures and event notification structures contain error information.

Command specification errors:

The following two tables describe the command specification errors that may be returned to the application in response structures.

This first table lists the command specification errors by error code. These same errors are also listed alphabetically by return value in Table 31 on page 251.

Table 30. Command specification errors (listed by error code)

Error code	Return value	Description	The *mc_args* field of the mc_errnum structure will be a pointer to an array containing, in order, the following argument(s):
0x30001	RMC_EOKBUTOTHERERROR	This command was correctly specified but another command in the ordered command group was not.	No arguments
0x30002	RMC_ECLASSNOTDEFINED	The specified class name is not defined.	class name
0x30011	RMC_ESELSTRBADNODNUM	select string does not contain any node numbers within the range 1-N, where N is the highest configured cluster node number	• highest configured node number • highest configured node number • select string
0x30012	RMC_ESELSTRBADNODID	select string does not contain any configured node IDs	select string
0x30014	RMC_ESELSTRBADNEXPR	select string contains an improper expression involving a node number, i.e. the node number is not an integral value	select string
0x30018	RMC_GENERR_CMD_SPEC	The class name has not been specified	No arguments
0x3001e	RMC_ECMDNOTSUPPORTED	the command is not supported by the specified class	class name
0x3001f	RMC_EDUPATTRNAM	attribute name is a duplicate	• array index of duplicate attr • attribute name
0x30020	RMC_EEXPRNODATTRNAME	expression does not contain a dynamic attribute name	expression
0x30021	RMC_GENERR_CMD_SPEC	expression is missing from command	No arguments
0x30022	RMC_ERAEXPRISNULL	rearm expression is a NULL string	No arguments
0x30023	RMC_EEXPRDIFFATTRS	expression and rearm expression contain different attribute names	• attribute name • attribute name
0x30024	RMC_EATTRMISSING	an attribute name is missing from the attribute array	array index of missing attr
0x30025	RMC_EBADATTRNAM	attribute name is not defined	• array index of invalid attr • attribute name
0x30026	RMC_ENOREGEVENT	event not registered; validity check only requested	No arguments
0x30027	RMC_EBADREGID	specified registration ID does not match a registered event for the session	No arguments
0x30028	RMC_ENOPATTRSDEFINED	no persistent attributes are defined for the specified class.	class name
0x30029	RMC_ENODATTRSDEFINED	no dynamic attributes are defined for the specified class.	class name
0x3002a	RMC_ENOCPATTRSDEFINED	no class persistent attributes are defined for the specified class.	class name
0x3002b	RMC_ENOCDATTRSDEFINED	no class dynamic attributes are defined for the specified class.	class name
0x3002c	RMC_EBADRSRCHANDLE	the specified resource handle is invalid	no arguments
0x3002d	RMC_ENOLOCATORATTR	the locator attribute was not included in the define resource command for the specified class.	• locator attribute name • class name

Table 30. Command specification errors (listed by error code) (continued)

Error code	Return value	Description	The *mc_args* field of the mc_errnum structure will be a pointer to an array containing, in order, the following argument(s):
0x3002e	RMC_EBADATTRTYPE	specified attribute type does not match defined type for specified attribute	• array index of invalid attr • attribute name
0x3002f	RMC_ENORSRCFORCMD	the command specified one or more resources but the specified class does not support resources	class name
0x30030	RMC_EEXPRNOTFORQUANTUM	expression contains more than just a single attribute name token and the named attribute is a quantum variable type	expression
0x30031	RMC_ERAEXPRNOTALLOWED	a re-arm expression was specified with a dynamic attribute that is a quantum variable type	expression
0x30032	RMC_ENOQUERYQUANTUM	a query dynamic attribute command specified an attribute that is of quantum variable type. Quantum dynamic attributes cannot be queried	• array index of quantum attr • attribute name
0x30033	RMC_EALLQUANTUM	a query dynamic attribute command indicated that all attributes were to be returned, but all dynamic attributes are of quantum variable type. Quantum dynamic attributes cannot be queried	name of associated class
0x30034	RMC_ENOATTRIBUTES	the command specified no attributes but this command requires that at least one attribute be specified.	name of associated class
0x30035	RMC_EMAYNOTSETPATTRS	persistent attributes of a MtypDivided class itself cannot be set	name of class
0x30036	RMC_ENOVALOTHERERROR	This command could not be validated since a previous command in the ordered command group was incorrectly specified	no arguments
0x30037	RMC_EBADSDUSEARG	an invalid sd_use argument was specified	no arguments
0x30038	RMC_EBADNAMECOUNT	a non-zero name count was specified but the remaining command arguments require that it be zero	no arguments
0x30039	RMC_ESDNOTDEFINED	No SDs for the use specified in the command are defined for this class	class name
0x3003a	RMC_EATTRNOTSD	attribute name is not an SD data type	• array index of invalid attr • attribute name
0x3003b	RMC_EACTIONMISSING	an action name is missing from the action array	array index of missing action
0x3003c	RMC_EBADACTIONNAM	action name is not defined	• array index of invalid action • action name
0x3003d	RMC_ENOACTIONSDINPUT	the specified action does not have a defined SD input	• array index of invalid action • action name
0x3003e	RMC_ENOACTIONSDRESPONSE	the specified action does not have a defined SD response	• array index of invalid action • action name
0x3003f	RMC_EBADVVUSEARG	an invalid vv_use argument was specified	no arguments

Table 30. Command specification errors (listed by error code) (continued)

Error code	Return value	Description	The *mc_args* field of the mc_errnum structure will be a pointer to an array containing, in order, the following argument(s):
0x30040	RMC_EAACNOTDEFINED	No attributes, actions with defined inputs or commands with defined inputs for the use specified in the command are defined for this class	class name
0x30041	RMC_EBADLOCATORVALUE	the locator attribute is of array type and does not contain just one element	locator attribute name
0x30042	RMC_ESELSTRNOSELECT	select string contains attributes that are not all defined for any variety of the resource; no resources can be selected	select string
0x30043	RMC_ESELSTRBADNAMEXPR	select string contains an improper expression involving a node name, i.e. the node name is not a string value	select string
0x30044	RMC_EINVALSCOPE	Requested scope does not match current daemon cluster configuration	no arguments
0x30045	RMC_EINVALNODENAMEARG	a command argument specified a node name but the node name is not a configured node name	invalid node name
0x30046	RMC_ENOCLUSTERNODES	The command specified a resource class that is not supported by any of the currently configured cluster nodes or there are no configured cluster nodes.	no arguments
0x30047	RMC_ENOTSUBDIVIDED	The command specified a node name list argument but the specified resource class is not of the Subdivided Management style. The node name list argument may only be used with resource classes that are managed using the Subdivided Management style.	no arguments
0x30048	RMC_EINVALLOCATORVALUE	the locator attribute value is not a configured Node ID.	locator attribute name
0x30049	RMC_EACCESSFORSESSION	the specified resource class cannot be accessed from a session in Distributed Management scope	resource class name
0x3004a	RMC_ENOAPDATTR	the ActivePeerDomain attribute was not included in the define resource command for the specified class.	• APD attribute name • class name
0x3004b	RMC_EBADAPDATTRVAL	the value of the ActivePeerDomain attribute is not a configured Peer Domain name.	APD attribute name
0x3004c	RMC_ENOTGLOBALIZED	The command specified a peer domain name list argument but the specified resource class is not of the Globalized Management style. The peer domain name list argument may only be used with resource classes that are managed using the Globalized Management style.	no arguments
0x3004d	RMC_EINVALAPDNAMEARG	a command argument specified a peer domain name but the peer domain name is not an active peer domain name	invalid peer domain name
0x3004e	RMC_ENOTDMSCOPE	The command specified a peer domain name list argument but the session is not in Distributed Management session scope. The peer domain name list argument may only be used when the session scope is Distributed Management.	no arguments

Table 30. Command specification errors (listed by error code) (continued)

Error code	Return value	Description	The *mc_args* field of the mc_errnum structure will be a pointer to an array containing, in order, the following argument(s):
0x3004f	RMC_EEXPRNOATTRNAME	expression does not contain an attribute name	expression
0x30050	RMC_EEXPRPATTRNOSUP	expression contains a persistent attribute not supported in expressions	expression
0x30051	RMC_EEXPRPTYPENOSUP	expression contains a persistent attribute of type resource handle	expression
0x30052	RMC_EINVALPERM	the permission argument contains no valid permission bits	permission argument
0x30053	RMC_EINVALLISTUSAGE	the list usage argument is not valid	list usage argument
0x30054	RMC_EACLTYPETARGET	the ACL type argument is not valid for the target of the command	ACL type argument
0x30055	RMC_EINVALACLTYPE	the ACL type argument is not valid	ACL type argument
0x30056	RMC_EACLFLAGTYPE	the ACL flag argument specifies to use the Resource Shared ACL, but the specified ACL type is not Resource nor Resource Initial	no arguments
0x30057	RMC_ENOACLALLOWED	the ACL flag argument indicates to use the Resource Shared ACL, but an ACL was also specified	no arguments
0x30058	RMC_EINVALACL	the specified ACL is not valid	no arguments

The following table lists the command specification errors by return value.

Table 31. Command specification errors (listed by return value)

Return value	Error code	Description	The *mc_args* field of the mc_errnum structure will be a pointer to an array containing, in order, the following argument(s):
RMC_EAACNOTDEFINED	0x30040	No attributes, actions with defined inputs or commands with defined inputs for the use specified in the command are defined for this class	class name
RMC_EACCESSFORSESSION	0x30049	the specified resource class cannot be accessed from a session in Distributed Management scope	resource class name
RMC_EACLFLAGTYPE	0x30056	the ACL flag argument specifies to use the Resource Shared ACL, but the specified ACL type is not Resource nor Resource Initial	no arguments
RMC_EACLTYPETARGET	0x30054	the ACL type argument is not valid for the target of the command	ACL type argument
RMC_EACTIONMISSING	0x3003b	an action name is missing from the action array	array index of missing action
RMC_EALLQUANTUM	0x30033	a query dynamic attribute command indicated that all attributes were to be returned, but all dynamic attributes are of quantum variable type. Quantum dynamic attributes cannot be queried	name of associated class
RMC_EATTRMISSING	0x30024	an attribute name is missing from the attribute array	array index of missing attr
RMC_EATTRNOTSD	0x3003a	attribute name is not an SD data type	• array index of invalid attr • attribute name

Table 31. Command specification errors (listed by return value) (continued)

Return value	Error code	Description	The *mc_args* field of the mc_errnum structure will be a pointer to an array containing, in order, the following argument(s):
RMC_EBADACTIONNAM	0x3003c	action name is not defined	• array index of invalid action • action name
RMC_EBADAPDATTRVAL	0x3004b	the value of the ActivePeerDomain attribute is not a configured Peer Domain name.	APD attribute name
RMC_EBADATTRNAM	0x30025	attribute name is not defined	• array index of invalid attr • attribute name
RMC_EBADATTRTYPE	0x3002e	specified attribute type does not match defined type for specified attribute	• array index of invalid attr • attribute name
RMC_EBADLOCATORVALUE	0x30041	the locator attribute is of array type and does not contain just one element	locator attribute name
RMC_EBADNAMECOUNT	0x30038	a non-zero name count was specified but the remaining command arguments require that it be zero	no arguments
RMC_EBADREGID	0x30027	specified registration ID does not match a registered event for the session	No arguments
RMC_EBADRSRCHANDLE	0x3002c	the specified resource handle is invalid	no arguments
RMC_EBADSDUSEARG	0x30037	an invalid sd_use argument was specified	no arguments
RMC_EBADVVUSEARG	0x3003f	an invalid vv_use argument was specified	no arguments
RMC_ECLASSNOTDEFINED	0x30002	The specified class name is not defined.	class name
RMC_ECMDNOTSUPPORTED	0x3001e	the command is not supported by the specified class	class name
RMC_EDUPATTRNAM	0x3001f	attribute name is a duplicate	• array index of duplicate attr • attribute name
RMC_EEXPRDIFFATTRS	0x30023	expression and rearm expression contain different attribute names	• attribute name • attribute name
RMC_EEXPRNOATTRNAME	0x3004f	expression does not contain an attribute name	expression
RMC_EEXPRNODATTRNAME	0x30020	expression does not contain a dynamic attribute name	expression
RMC_EEXPRNOTFORQUANTUM	0x30030	expression contains more than just a single attribute name token and the named attribute is a quantum variable type	expression
RMC_EEXPRPATTRNOSUP	0x30050	expression contains a persistent attribute not supported in expressions	expression
RMC_EEXPRPTYPENOSUP	0x30051	expression contains a persistent attribute of type resource handle	expression
RMC_EINVALACL	0x30058	the specified ACL is not valid	no arguments
RMC_EINVALACLTYPE	0x30055	the ACL type argument is not valid	ACL type argument
RMC_EINVALAPDNAMEARG	0x3004d	a command argument specified a peer domain name but the peer domain name is not an active peer domain name	invalid peer domain name
RMC_EINVALLISTUSAGE	0x30053	the list usage argument is not valid	list usage argument
RMC_EINVALLOCATORVALUE	0x30048	the locator attribute value is not a configured Node ID.	locator attribute name

Table 31. Command specification errors (listed by return value) (continued)

Return value	Error code	Description	The *mc_args* field of the mc_errnum structure will be a pointer to an array containing, in order, the following argument(s):
RMC_EINVALNODENAMEARG	0x30045	a command argument specified a node name but the node name is not a configured node name	invalid node name
RMC_EINVALPERM	0x30052	the permission argument contains no valid permission bits	permission argument
RMC_EINVALSCOPE	0x30044	Requested scope does not match current daemon cluster configuration	no arguments
RMC_EMAYNOTSETPATTRS	0x30035	persistent attributes of a MtypDivided class itself cannot be set	name of class
RMC_ENOACLALLOWED	0x30057	the ACL flag argument indicates to use the Resource Shared ACL, but an ACL was also specified	no arguments
RMC_ENOACTIONSDINPUT	0x3003d	the specified action does not have a defined SD input	• array index of invalid action • action name
RMC_ENOACTIONSDRESPONSE	0x3003e	the specified action does not have a defined SD response	• array index of invalid action • action name
RMC_ENOAPDATTR	0x3004a	the ActivePeerDomain attribute was not included in the define resource command for the specified class.	• APD attribute name • class name
RMC_ENOATTRIBUTES	0x30034	the command specified no attributes but this command requires that at least one attribute be specified.	name of associated class
RMC_ENOCDATTRSDEFINED	0x3002b	no class dynamic attributes are defined for the specified class.	class name
RMC_ENOCLUSTERNODES	0x30046	The command specified a resource class that is not supported by any of the currently configured cluster nodes or there are no configured cluster nodes.	no arguments
RMC_ENOCPATTRSDEFINED	0x3002a	no class persistent attributes are defined for the specified class.	class name
RMC_ENODATTRSDEFINED	0x30029	no dynamic attributes are defined for the specified class.	class name
RMC_ENOLOCATORATTR	0x3002d	the locator attribute was not included in the define resource command for the specified class.	• locator attribute name • class name
RMC_ENOPATTRSDEFINED	0x30028	no persistent attributes are defined for the specified class.	class name
RMC_ENOQUERYQUANTUM	0x30032	a query dynamic attribute command specified an attribute that is of quantum variable type. Quantum dynamic attributes cannot be queried	• array index of quantum attr • attribute name
RMC_ENOREGEVENT	0x30026	event not registered; validity check only requested	No arguments
RMC_ENORSRCFORCMD	0x3002f	the command specified one or more resources but the specified class does not support resources	class name

Table 31. Command specification errors (listed by return value) (continued)

Return value	Error code	Description	The *mc_args* field of the mc_errnum structure will be a pointer to an array containing, in order, the following argument(s):
RMC_ENOTDMSCOPE	0x3004e	The command specified a peer domain name list argument but the session is not in Distributed Management session scope. The peer domain name list argument may only be used when the session scope is Distributed Management.	no arguments
RMC_ENOTGLOBALIZED	0x3004c	The command specified a peer domain name list argument but the specified resource class is not of the Globalized Management style. The peer domain name list argument may only be used with resource classes that are managed using the Globalized Management style.	no arguments
RMC_ENOTSUBDIVIDED	0x30047	The command specified a node name list argument but the specified resource class is not of the Subdivided Management style. The node name list argument may only be used with resource classes that are managed using the Subdivided Management style.	no arguments
RMC_ENOVALOTHERERROR	0x30036	This command could not be validated since a previous command in the ordered command group was incorrectly specified	no arguments
RMC_EOKBUTOTHERERROR	0x30001	This command was correctly specified but another command in the ordered command group was not.	No arguments
RMC_ERAEXPRISNULL	0x30022	rearm expression is a NULL string	No arguments
RMC_ERAEXPRNOTALLOWED	0x30031	a re-arm expression was specified with a dynamic attribute that is a quantum variable type	expression
RMC_ESDNOTDEFINED	0x30039	No SDs for the use specified in the command are defined for this class	class name
RMC_ESELSTRBADNAMEXPR	0x30043	select string contains an improper expression involving a node name, i.e. the node name is not a string value	select string
RMC_ESELSTRBADNEXPR	0x30014	select string contains an improper expression involving a node number, i.e. the node number is not an integral value	select string
RMC_ESELSTRBADNODID	0x30012	select string does not contain any configured node IDs	select string
RMC_ESELSTRBADNODNUM	0x30011	select string does not contain any node numbers within the range 1-N, where N is the highest configured cluster node number	• highest configured node number • highest configured node number • select string
RMC_ESELSTRNOSELECT	0x30042	select string contains attributes that are not all defined for any variety of the resource; no resources can be selected	select string
RMC_GENERR_CMD_SPEC	0x30018	The class name has not been specified	No arguments
RMC_GENERR_CMD_SPEC	0x30021	expression is missing from command	No arguments

Resource access or usage errors:

The following two tables describe the resource access or usage errors that may be returned to the application in response structures.

This first table lists the resource access or usage errors by error code. These same errors are also listed alphabetically by return value in Table 33 on page 258.

Table 32. Resource access or usage errors (listed by error code)

Error code	Return value	Description	The *mc_args* field of the mc_errnum structure will be a pointer to an array containing, in order, the following argument(s):
0x40001	RMC_ERSRCNOTDEFINED	The resource indicated by the resource handle in the response is not defined	no arguments
0x40002	RMC_ERSRCUNDEFINED	The resource indicated by the resource handle in the response has been undefined	no arguments
0x40003	RMC_ESHAREDMEMORY	The shared memory used to monitor the resource specified by the resource handle in the response is no longer valid	no arguments
0x40004	RMC_ERSRCSTALE	The resource indicated by the resource handle in the response is stale; the resource manager supplying the resource has terminated. The remaining data in the response represents the last known values	no arguments
0x40005	RMC_EEVALUATIONERROR	The evaluation of the dynamic attribute the resource specified by the resource handle resulted in an error	• expr evaluation error number **1** divide by 0 **2** invalid SD array index **3** array index is out of bounds **4** SD element ID is greater than the number of elements present **5** operator used where the left operand is greater than the right **>100** unexpected evaluation error • attribute name
0x40006	RMC_EMISSINGPATTRVALUES	Not all of the requested persistent attribute values could be obtained from the resource specified by the resource handle or from the class specified by the class name. The count in the response specifies how many are returned	number of attribute values not returned
0x40007	RMC_ENODENOTAVAILABLE	The command for the resource or resource class specified in the response could not be executed on the node specified by the error argument; the node is not currently in the cluster.	name of unavailable node
0x40008	RMC_ENORSRCSFOUND	No resources could be found using the select string specified in the command	no arguments

Table 32. Resource access or usage errors (listed by error code) (continued)

Error code	Return value	Description	The *mc_args* field of the mc_errnum structure will be a pointer to an array containing, in order, the following argument(s):
0x40009	RMC_ENOSELECTDONE	Resource selection could not be performed using the select string specified in the command	no arguments
0x4000a	RMC_ERMNOTAVAILABLE	**In Cluster Mode:** The command for the resource or resource class specified in the response could not be executed on the node specified by the second error argument; the Resource Manager identified by the first error argument is not available.**In Stand-alone Mode:** The command for the resource or resource class specified in the response could not be executed; the Resource Manager named by the error argument is not available.	**In Cluster Mode:** • Resource Manager name • number or name of unavailable node **In Stand-alone Mode:** Resource Manager name
0x4000b	RMC_ERMTERMWITHRSP	**In Cluster Mode:** The command for the resource or resource class specified in the response could not be completed on the node specified by the second error argument; the Resource Manager identified by the first error argument has terminated. A partial response was previously returned.**In Stand-alone Mode:** The command for the resource or resource class specified in the response could not be completed; the Resource Manager named by the error argument has terminated. A partial response was previously returned.	**In Cluster Mode:** • Resource Manager name • number or name of unavailable node **In Stand-alone Mode:** Resource Manager name
0x4000c	RMC_ERMTERMWITHNORSP	**In Cluster Mode:** The command for the resource or resource class specified in the response could not be completed on the node specified by the second error argument; the Resource Manager identified by the first error argument has terminated. While no response was returned, the command may have executed prior to RM termination.**In Stand-alone Mode:** The command for the resource or resource class specified in the response could not be completed; the Resource Manager named by the error argument has terminated. While no response was returned, the command may have executed prior to RM termination.	**In Cluster Mode:** • Resource Manager name • number or name of unavailable node **In Stand-alone Mode:** Resource Manager name
0x4000d	RMC_ENOENUMRSP	A Resource Manager terminated while attempting to enumerate resources for this command. If an FFDC ID is present it can be used to obtain more information about the error. Additional responses may still be returned for this command.	no arguments

Table 32. Resource access or usage errors (listed by error code) (continued)

Error code	Return value	Description	The *mc_args* field of the mc_errnum structure will be a pointer to an array containing, in order, the following argument(s):
0x4000e	RMC_EENUMERROR	An error was detected when attempting to enumerate resources for this command. If an FFDC ID is present it can be used to obtain more information about the error. Additional responses may still be returned for this command.	no arguments
0x4000f	RMC_ENODESCRIPTIONS	Descriptions were requested by this command for the specified resource class but cannot be returned; descriptions are not currently available. All other requested information is in the response.	Resource Class name
0x40010	RMC_EACCESS	**If the response structure is for a command that operates on a one or more resources:** Permission denied to access a resource specified in this command.**If the response structure is for a command that operates on a resource class:** Permission denied to access the resource class specified in this command.	**If the response structure is for a command that operates on a one or more resources:** • Network Identity • permission character • Resource Class name • node name • Resource Handle **If the response structure is for a command that operates on a resource class:** • Network Identity • permission character • Resource Class name • node name
0x40011	RMC_ENODEOUT	The resource or resource class specified the response is no longer available on the node specified by the error argument; the node has left the cluster	name of node
0x40012	RMC_EDATTRNOTSUPPORTED	The dynamic attribute specified in the expression(s) supplied in the event registration command is not supported in the resource specified by the resource handle or in the class specified by the class name.	no arguments
0x40013	RMC_EMISSINGDATTRVALUES	Not all of the requested dynamic attribute values could be obtained from the resource specified by the resource handle or from the class specified by the class name. The count in the response specifies how many are returned.	number of attribute values not returned
0x40014	RMC_ERSRCNOTAVAILABLE	The resource or resource class specified in the response is not currently available on the node specified by the error argument.	name of node
0x40015	RMC_EMONITRINGSUSPENDED	Monitoring of the resource indicated by the resource handle in the response is temporarily suspended; the monitoring location is changing.	no arguments
0x40016	RMC_EMONITRINGNOTSTARTD	Monitoring of the resource indicated by the resource handle in the response has not yet started; a response is pending from the resource manager.	no arguments

Table 32. Resource access or usage errors (listed by error code) (continued)

Error code	Return value	Description	The *mc_args* field of the mc_errnum structure will be a pointer to an array containing, in order, the following argument(s):
0x40017	RMC_ECLASSNOTINSTALLED	The resource class specified, explicitly or implicitly, in the response is not installed on the node specified by the error argument	name of node
0x40018	RMC_ENOENUMNODE	Cannot enumerate resources for this command on the node specified by the error argument; the node is not currently in the cluster. Additional responses may still be returned for this command.	name of unavailable node
0x40019	RMC_EMONITRINGNODATA	Monitoring of the resource indicated by the resource handle in the response has started but data is not yet available from the resource manager.	no arguments
0x4001a	RMC_EATTRNOTSUPPORTED	The attribute specified in the expression(s) supplied in the event registration command is not supported in the resource specified by the resource handle or in the class specified by the class name.	no arguments
0x4001b	RMC_ENODEFUNCLEVEL	The resource or resource class specified in the response cannot be accessed on the node specified by the error argument; the node is not operating at the necessary functional level for this command.	name of node
0x4001c	RMC_EDOMAINFUNCLEVEL	The resource or resource class specified the response cannot be accessed; the domain is not operating at the necessary functional level for this command.	name of peer domain

The following table lists the resource access or usage errors by return value.

Table 33. Resource access or usage errors (listed by return value)

Error code	Return value	Description	The *mc_args* field of the mc_errnum structure will be a pointer to an array containing, in order, the following argument(s):
0x40010	RMC_EACCESS	**If the response structure is for a command that operates on a one or more resources:** Permission denied to access a resource specified in this command.**If the response structure is for a command that operates on a resource class:** Permission denied to access the resource class specified in this command.	**If the response structure is for a command that operates on a one or more resources:** • Network Identity • permission character • Resource Class name • node name • Resource Handle **If the response structure is for a command that operates on a resource class:** • Network Identity • permission character • Resource Class name • node name

Table 33. Resource access or usage errors (listed by return value) (continued)

Error code	Return value	Description	The *mc_args* field of the mc_errnum structure will be a pointer to an array containing, in order, the following argument(s):
0x4001a	RMC_EATTRNOTSUPPORTED	The attribute specified in the expression(s) supplied in the event registration command is not supported in the resource specified by the resource handle or in the class specified by the class name.	no arguments
0x40017	RMC_ECLASSNOTINSTALLED	The resource class specified, explicitly or implicitly, in the response is not installed on the node specified by the error argument	name of node
0x40012	RMC_EDATTRNOTSUPPORTED	The dynamic attribute specified in the expression(s) supplied in the event registration command is not supported in the resource specified by the resource handle or in the class specified by the class name.	no arguments
0x4001c	RMC_EDOMAINFUNCLEVEL	The resource or resource class specified the response cannot be accessed; the domain is not operating at the necessary functional level for this command.	name of peer domain
0x4000e	RMC_EENUMERROR	An error was detected when attempting to enumerate resources for this command. If an FFDC ID is present it can be used to obtain more information about the error. Additional responses may still be returned for this command.	no arguments
0x40005	RMC_EEVALUATIONERROR	The evaluation of the dynamic attribute the resource specified by the resource handle resulted in an error	• expr evaluation error number **1** divide by 0 **2** invalid SD array index **3** array index is out of bounds **4** SD element ID is greater than the number of elements present **5** operator used where the left operand is greater than the right **>100** unexpected evaluation error • attribute name
0x40013	RMC_EMISSINGDATTRVALUES	Not all of the requested dynamic attribute values could be obtained from the resource specified by the resource handle or from the class specified by the class name. The count in the response specifies how many are returned.	number of attribute values not returned
0x40006	RMC_EMISSINGPATTRVALUES	Not all of the requested persistent attribute values could be obtained from the resource specified by the resource handle or from the class specified by the class name. The count in the response specifies how many are returned	number of attribute values not returned

Table 33. Resource access or usage errors (listed by return value) (continued)

Error code	Return value	Description	The *mc_args* field of the mc_errnum structure will be a pointer to an array containing, in order, the following argument(s):
0x40019	RMC_EMONITRINGNODATA	Monitoring of the resource indicated by the resource handle in the response has started but data is not yet available from the resource manager.	no arguments
0x40016	RMC_EMONITRINGNOTSTARTD	Monitoring of the resource indicated by the resource handle in the response has not yet started; a response is pending from the resource manager.	no arguments
0x40015	RMC_EMONITRINGSUSPENDED	Monitoring of the resource indicated by the resource handle in the response is temporarily suspended; the monitoring location is changing.	no arguments
0x4001b	RMC_ENODEFUNCLEVEL	The resource or resource class specified in the response cannot be accessed on the node specified by the error argument; the node is not operating at the necessary functional level for this command.	name of node
0x40007	RMC_ENODENOTAVAILABLE	The command for the resource or resource class specified in the response could not be executed on the node specified by the error argument; the node is not currently in the cluster.	name of unavailable node
0x40011	RMC_ENODEOUT	The resource or resource class specified the response is no longer available on the node specified by the error argument; the node has left the cluster	name of node
0x4000f	RMC_ENODESCRIPTIONS	Descriptions were requested by this command for the specified resource class but cannot be returned; descriptions are not currently available. All other requested information is in the response.	Resource Class name
0x4000d	RMC_ENOENUMRSP	A Resource Manager terminated while attempting to enumerate resources for this command. If an FFDC ID is present it can be used to obtain more information about the error. Additional responses may still be returned for this command.	no arguments
0x40008	RMC_ENORSRCSFOUND	No resources could be found using the select string specified in the command	no arguments
0x40009	RMC_ENOSELECTDONE	Resource selection could not be performed using the select string specified in the command	no arguments
0x40018	RMC_ENOENUMNODE	Cannot enumerate resources for this command on the node specified by the error argument; the node is not currently in the cluster. Additional responses may still be returned for this command.	name of unavailable node

Table 33. Resource access or usage errors (listed by return value) (continued)

Error code	Return value	Description	**The *mc_args* field of the mc_errnum structure will be a pointer to an array containing, in order, the following argument(s):**
0x4000a	RMC_ERMNOTAVAILABLE	**In Cluster Mode:** The command for the resource or resource class specified in the response could not be executed on the node specified by the second error argument; the Resource Manager identified by the first error argument is not available.**In Stand-alone Mode:** The command for the resource or resource class specified in the response could not be executed; the Resource Manager named by the error argument is not available.	**In Cluster Mode:** • Resource Manager name • number or name of unavailable node **In Stand-alone Mode:** Resource Manager name
0x4000c	RMC_ERMTERMWITHNORSP	**In Cluster Mode:** The command for the resource or resource class specified in the response could not be completed on the node specified by the second error argument; the Resource Manager identified by the first error argument has terminated. While no response was returned, the command may have executed prior to RM termination.**In Stand-alone Mode:** The command for the resource or resource class specified in the response could not be completed; the Resource Manager named by the error argument has terminated. While no response was returned, the command may have executed prior to RM termination.	**In Cluster Mode:** • Resource Manager name • number or name of unavailable node **In Stand-alone Mode:** Resource Manager name
0x4000b	RMC_ERMTERMWITHRSP	**In Cluster Mode:** The command for the resource or resource class specified in the response could not be completed on the node specified by the second error argument; the Resource Manager identified by the first error argument has terminated. A partial response was previously returned.**In Stand-alone Mode:** The command for the resource or resource class specified in the response could not be completed; the Resource Manager named by the error argument has terminated. A partial response was previously returned.	**In Cluster Mode:** • Resource Manager name • number or name of unavailable node **In Stand-alone Mode:** Resource Manager name
0x40014	RMC_ERSRCNOTAVAILABLE	The resource or resource class specified in the response is not currently available on the node specified by the error argument.	name of node
0x40001	RMC_ERSRCNOTDEFINED	The resource indicated by the resource handle in the response is not defined	no arguments
0x40004	RMC_ERSRCSTALE	The resource indicated by the resource handle in the response is stale; the resource manager supplying the resource has terminated. The remaining data in the response represents the last known values	no arguments
0x40002	RMC_ERSRCUNDEFINED	The resource indicated by the resource handle in the response has been undefined	no arguments

Table 33. Resource access or usage errors (listed by return value) (continued)

Error code	Return value	Description	The *mc_args* field of the mc_errnum structure will be a pointer to an array containing, in order, the following argument(s):
0x40003	RMC_ESHAREDMEMORY	The shared memory used to monitor the resource specified by the resource handle in the response is no longer valid	no arguments

Expression specification errors:

The following two tables describe the expression specification errors that may be returned to the application in response structures.

This first table lists the expression specification errors by error code. These same errors are also listed alphabetically by return value in Table 35.

Table 34. Expression specification errors (listed by error code)

Error code	Return value	Description
0x50002	CU_EINVALIDNAME	Invalid attribute name in expression
0x50003	CU_ENOMEM	Cannot allocate memory
0x50004	CU_EINVAL	Invalid constant value
0x50005	CU_EDIVIDEZERO	Divide by zero
0x50006	CU_EINVALIDSDINDEX	Invalid Structure Data index value
0x50007	CU_EINVALIDINDEX	Invalid array index
0x50008	CU_EINVALIDELEMENTID	Invalid SD element ID
0x50009	CU_EINVALIDOPTION	Invalid option to expression engine
0x5000a	CU_EMISSINGFUNC	Function not specified to expression engine
0x5000b	CU_ENOEXPR	Missing expression
0x5000c	CU_ETOOFEWVALUES	Too few values passed to expression engine
0x5000d	CU_EINVALIDRANGE	Invalid range
0x5000e	CU_EINVALIDFORMAT	Invalid data format given to expression engine
0x5000f	CU_EVARCONFLICT	Error in expression expansion

The following table lists the expression specification errors by return value.

Table 35. Expression specification errors (listed by return value)

Return value	Error code	Description
CU_EDIVIDEZERO	0x50005	Divide by zero
CU_EINVAL	0x50004	Invalid constant value
CU_EINVALIDELEMENTID	0x50008	Invalid SD element ID
CU_EINVALIDFORMAT	0x5000e	Invalid data format given to expression engine
CU_EINVALIDINDEX	0x50007	Invalid array index
CU_EINVALIDNAME	0x50002	Invalid attribute name in expression
CU_EINVALIDOPTION	0x50009	Invalid option to expression engine
CU_EINVALIDRANGE	0x5000d	Invalid range
CU_EINVALIDSDINDEX	0x50006	Invalid Structure Data index value
CU_EMISSINGFUNC	0x5000a	Function not specified to expression engine
CU_ENOEXPR	0x5000b	Missing expression

Table 35. Expression specification errors (listed by return value) (continued)

Return value	Error code	Description
CU_ENOMEM	0x50003	Cannot allocate memory
CU_ETOOFEWVALUES	0x5000c	Too few values passed to expression engine
CU_EVARCONFLICT	0x5000f	Error in expression expansion

Select string specification errors:

The following two tables describe the expression specification errors that may be returned to the application in response structures.

This first table lists the select string specification errors by error code. These same errors are also listed alphabetically by return value in Table 37.

Table 36. Select string specification errors (listed by error code)

Error code	Return value	Description
0x60002	CU_EINVALIDNAME	Invalid attribute name in expression
0x60003	CU_ENOMEM	Cannot allocate memory
0x60004	CU_EINVAL	Invalid constant value
0x60005	CU_EDIVIDEZERO	Divide by zero
0x60006	CU_EINVALIDSDINDEX	Invalid Structure Data index value
0x60007	CU_EINVALIDINDEX	Invalid array index
0x60008	CU_EINVALIDELEMENTID	Invalid SD element ID
0x60009	CU_EINVALIDOPTION	Invalid option to expression engine
0x6000a	CU_EMISSINGFUNC	Function not specified to expression engine
0x6000b	CU_ENOEXPR	Missing expression
0x6000c	CU_ETOOFEWVALUES	Too few values passed to expression engine
0x6000d	CU_EINVALIDRANGE	Invalid range
0x6000e	CU_EINVALIDFORMAT	Invalid data format given to expression engine
0x6000f	CU_EVARCONFLICT	Error in expression expansion

The following table lists the select string specification errors by return value.

Table 37. Select string specification errors (listed by return value)

Return value	Error code	Description
CU_EDIVIDEZERO	0x60005	Divide by zero
CU_EINVAL	0x60004	Invalid constant value
CU_EINVALIDELEMENTID	0x60008	Invalid SD element ID
CU_EINVALIDFORMAT	0x6000e	Invalid data format given to expression engine
CU_EINVALIDINDEX	0x60007	Invalid array index
CU_EINVALIDNAME	0x60002	Invalid attribute name in expression
CU_EINVALIDOPTION	0x60009	Invalid option to expression engine
CU_EINVALIDRANGE	0x6000d	Invalid range
CU_EINVALIDSDINDEX	0x60006	Invalid Structure Data index value
CU_EMISSINGFUNC	0x6000a	Function not specified to expression engine
CU_ENOEXPR	0x6000b	Missing expression
CU_ENOMEM	0x60003	Cannot allocate memory
CU_ETOOFEWVALUES	0x6000c	Too few values passed to expression engine

Table 37. Select string specification errors (listed by return value) (continued)

Return value	Error code	Description
CU_EVARCONFLICT	0x6000f	Error in expression expansion

Cluster utilities: error-related subroutines

The cluster utilities component of RSCT includes several subroutines that an application can use to get additional information about errors that are returned by RMC API subroutines.

Related concepts:

"Notifying the application of errors" on page 29
Errors can be detected in various ways when using the RMC API.

"Obtaining error information returned by the RMC API subroutines" on page 29
All RMC API subroutines return a value of type **ct_int32_t**. A return value of 0 indicates that the subroutine completed successfully. Any non-zero value is an error value.

cu_get_errmsg

Gets an error message that corresponds to an error structure that is obtained by the **cu_get_error** subroutine.

Library

Cluster utilities library (**libct_cu.a**)

Syntax

```
#include <rsct/ct_cu.h>

extern void
cu_get_errmsg(
     cu_error_t       *err_p,
     ct_char_t       **msg_pp
);
```

Parameters

Input

err_p Pointer to an error structure obtained by the cluster common subroutine **cu_get_error**.

Output

msg_pp
Pointer to a location where the subroutine will place the error message.

Description

The **cu_get_errmsg** subroutine enables the application to get an error message, suitably translated, that corresponds to an error structure obtained by the cluster common subroutine **cu_get_error**. The application passes the **cu_get_errmsg** subroutine a pointer to the error structure and specifies a location for the message.

The memory returned by the **cu_get_errmsg** subroutine is not reused by the subroutine. The application can therefore hold the memory as long as necessary, independent of any other subroutine call. However, the memory returned by the subroutine must not be modified by the application. When it no longer needs the error message, the application can invoke the **cu_rel_errmsg** subroutine to free the memory used to store the message.

Location

/usr/lib/libct_cu.a

Related reference:

"cu_get_error"
Gets detailed error information about errors that are returned by RMC API subroutines.

"cu_pkg_error, cu_vpkg_error" on page 267
Package error information into a cluster error structure or create cluster error structures that are returned by microsensor API subroutines.

"cu_rel_errmsg" on page 269
Frees the memory associated with a message that is obtained by the **cu_get_errmsg** subroutine.

"cu_rel_error" on page 270
Frees the memory associated with an error structure that is obtained by the **cu_get_error** subroutine.

cu_get_error

Gets detailed error information about errors that are returned by RMC API subroutines.

Library

Cluster utilities library (**libct_cu.a**)

Syntax

```
#include <rsct/ct_cu.h>

extern void
cu_get_error(
    cu_error_t     **err_pp,
);
```

Parameters

Input

err_pp The address of a location in which the **cu_get_error** subroutine returns a pointer to an error structure.

Description

The **cu_get_error** subroutine enables an application to obtain detailed error information for errors that are returned by RMC API subroutines.

RMC API subroutines save this error information in a common, per-thread area. Therefore, to obtain additional error information, the application must call the **cu_get_error** subroutine using the same thread that invoked the subroutine that returned the error, before calling any other subroutine on that thread.

This subroutine has one argument, which is the address of a location in which the subroutine returns a pointer to the following error structure:

```
typedef struct cu_error   cu_error_t;
struct cu_error {
     ct_int32_t           cu_error_id;
     ct_char_t            *cu_ffdc_id;
     ct_char_t            *cu_msg_cat;
     ct_int32_t           cu_msg_set
     ct_int32_t           cu_msg_num;
```

```
    ct_char_t           *cu_msg_default;
    ct_uint32_t         cu_arg_cnt;
    cu_error_arg_t      *cu_args;
};
```

The fields of this structure contain the following:

cu_error_id
: The error value originally returned by the RMC API subroutine.

cu_ffdc_id
: A pointer to a string that is a failure identifier. This failure identifier specifies additional error information that may have been logged by the subroutine. If this field contains a Null pointer, then no additional error information has been logged. If the application is logging errors that is has detected, then this failure identifier should be included in the information being logged.

cu_msg_cat
: A pointer to the name of the message catalog that contains the error.

cu_msg_set
: The message set number.

cu_msg_num
: The message number.

cu_msg_default
: A pointer to the error message.

cu_arg_cnt
: The number of elements in the *cu_args* array.

cu_args
: A pointer to an array of *cu_arg_cnt* elements of type **cu_error_arg_t**. Each **cu_error_arg_t** element is a structure of that contains an argument value and a value type.

```
typedef struct cu_error_arg          cu_error_arg_t;
struct cu_error_arg {
    cu_error_arg_type_t              cu_arg_type;
    cu_error_arg_value_t             cu_arg_value;
};
```

The fields of this structure contain the following:

cu_arg_type
: The argument type are defined by the **cu_error_arg_type_t** enumeration.

```
typdef enum cu_error_arg_type cu_error_arg_type_t;
enum cu_error_arg_type {
    CU_ERROR_ARG_INT,
    CU_ERROR_ARG_LONG,
    CU_ERROR_ARG_LONG_LONG,
    CU_ERROR_ARG_DOUBLE,
    CU_ERROR_ARG_RESERVED,
    CU_ERROR_ARG_CHAR_STR,
    CU_ERROR_ARG_VOID_PTR
};
```

cu_arg_value
: The argument value as defined by the **cu_error_arg_value_t** union.

```
typedef union cu_error_arg_value cu_error_arg_value_t;
union cu_error_arg_value {
    int         cu_arg_int;
    long        cu_arg_long;
    long long   cu_arg_long_long;
```

```
    double       cu_arg_double;
    char         *cu_arg_char_str;
    void         *cu_arg_void_ptr;
};
```

The memory returned by the **cu_get_error** subroutine is not reused by the subroutine. The application can therefore hold the memory as long as necessary, independent of any other subroutine call. However, the memory returned by this subroutine must not be modified by the application. When the detailed error information is no longer needed, the application can free it by calling the cluster common subroutine **cu_rel_errmsg**.

The application can obtain a message corresponding to the error, suitably translated, by invoking the cluster common subroutine **cu_get_errmsg**, passing it a pointer to the **cu_error** structure. When the application no longer needs the error message, it can call the **cu_rel_errmsg** subroutine to free the memory used to store the message.

Location

/usr/lib/libct_cu.a

Related reference:

"cu_get_errmsg" on page 264
Gets an error message that corresponds to an error structure that is obtained by the **cu_get_error** subroutine.

"cu_pkg_error, cu_vpkg_error"
Package error information into a cluster error structure or create cluster error structures that are returned by microsensor API subroutines.

"cu_rel_errmsg" on page 269
Frees the memory associated with a message that is obtained by the **cu_get_errmsg** subroutine.

"cu_rel_error" on page 270
Frees the memory associated with an error structure that is obtained by the **cu_get_error** subroutine.

cu_pkg_error, cu_vpkg_error

Package error information into a cluster error structure or create cluster error structures that are returned by microsensor API subroutines.

Library

Cluster utilities library (**libct_cu.a**)

Syntax

```
#include <rsct/ct_cu.h>

extern ct_int32_t
cu_pkg_error(
    cu_error_t          **err_pp,
    ct_int32_t           error_id,
    const ct_char_t     *ffdc_id,
    const ct_char_t     *msg_cat,
    ct_int32_t           msg_set,
    ct_int32_t           msg_num,
    const ct_char_t     *msg_default,
    ...
);

extern ct_int32_t
cu_vpkg_error(
    cu_error_t          **err_pp,
    ct_int32_t           error_id,
```

```
    const ct_char_t     *ffdc_id,
    const ct_char_t     *msg_cat,
    ct_int32_t           msg_set,
    ct_int32_t           msg_num,
    const ct_char_t     *msg_default,
    va_list              val
);
```

Parameters

Input

error_id
: The cluster function error code.

ffdc_id The first failure data capture identifier.

msg_cat
: Name of message catalog from which an error message describing the error can be obtained.

msg_set
: Number of the set within the message catalog identified by the *msg_cat* argument from which an error message describing the error can be obtained.

msg_num
: Number of the message within the set identified by the *msg_set* argument and the message catalog identified by the *msg_cat* argument describing the error.

msg_default
: Default message describing the error.

... Error arguments.

va_list
: Variable argument list as defined by **stdarg.h**.

Output

err_pp Specifies a pointer to memory where a pointer to a **cu_error_t** structure is to be returned.

Description

You can call these subroutines to package error information into a cluster error structure or to create cluster error structures that are returned by microsensor API subroutines.

These subroutines interpret the default error message specified by the *msg_default* parameter as a **printf()** format string. These routines recognize **printf()** conversion specifications within the format string, and like printf(), use the conversion specifications to interpret the remaining arguments.

Most, but not all, valid **printf()** conversion specifications are recognized by these subroutines.

A **printf()** conversion specification starts with the **%** character or the **%n$** character sequence, followed, in order, by any of the following:

- Zero or more flags
- An optional minimum field width
- An optional precision
- An optional length modifier that specifies the size of an error argument
- A conversion specifier character that indicates the type of conversion to apply to an error argument

See the **printf()** man page for details.

These subroutines recognize the following combinations of length modifier and conversion specifier.

In order to support integer error arguments, these subroutines recognize the following conversion specifiers: **d**, **i**, **o**, **u**, **x**, and **X**

These subroutines also recognize the following optional length modifiers applied to them: **h**, **l**, and **ll**

In order to support floating point error arguments, these subroutines recognize the following conversion specifiers: **e**, **E**, **f**, **g**, and **G**, and the following optional length modifier applied to them: **l**

In order to support character, character string, and void pointer error arguments, respectively, these subroutines also recognize the following conversion specifiers: **c**, **p**, and **s**

These subroutines recognize no optional length modifiers for these conversion specifiers.

The message identified by the *msg_cat*, *msg_num*, and *msg_set* parameters is expected to contain the same conversion specifications as the default message.

The returned cluster error structure can be released by calling **cu_rel_error()**.

Return values

If successful, this subroutine returns the value of the *error_id* argument. If unsuccessful, this subroutine returns -1.

Location

/usr/lib/libct_cu.a

Related reference:

"cu_get_errmsg" on page 264
Gets an error message that corresponds to an error structure that is obtained by the **cu_get_error** subroutine.

"cu_get_error" on page 265
Gets detailed error information about errors that are returned by RMC API subroutines.

"cu_rel_errmsg"
Frees the memory associated with a message that is obtained by the **cu_get_errmsg** subroutine.

"cu_rel_error" on page 270
Frees the memory associated with an error structure that is obtained by the **cu_get_error** subroutine.

cu_rel_errmsg

Frees the memory associated with a message that is obtained by the **cu_get_errmsg** subroutine.

Library

Cluster utilities library (**libct_cu.a**)

Syntax

```
#include <rsct/ct_cu.h>

extern void
cu_rel_errmsg(
        ct_char_t     *msg_p
);
```

Parameters

Input

msg_p Pointer to the error message to be freed.

Description

The **cu_rel_errmsg** subroutine enables the application to free the memory associated with a message previously obtained by the **cu_get_errmsg** subroutine. The memory returned by the **cu_get_errmsg** subroutine is not reused by the subroutine. The **cu_rel_errmsg** subroutine enables the application to free the memory.

Location

/usr/lib/libct_cu.a

Related reference:

"cu_get_errmsg" on page 264
Gets an error message that corresponds to an error structure that is obtained by the **cu_get_error** subroutine.

"cu_get_error" on page 265
Gets detailed error information about errors that are returned by RMC API subroutines.

"cu_pkg_error, cu_vpkg_error" on page 267
Package error information into a cluster error structure or create cluster error structures that are returned by microsensor API subroutines.

"cu_rel_error"
Frees the memory associated with an error structure that is obtained by the **cu_get_error** subroutine.

cu_rel_error

Frees the memory associated with an error structure that is obtained by the **cu_get_error** subroutine.

Library

Cluster utilities library (**libct_cu.a**)

Syntax

```
#include <rsct/ct_cu.h>

extern void
cu_rel_errmsg(
          cu_error_t     *err_p,
);
```

Parameters

Input

err_p Pointer to an error structure to be freed.

Description

The **cu_rel_error** subroutine enables the application to free the memory associated with an error structure previously obtained by the **cu_get_error** subroutine. The memory returned by the **cu_get_error** subroutine is not reused by the subroutine. The **cu_rel_error** subroutine enables the application to free the memory.

Location

/usr/lib/libct_cu.a

Related reference:

"cu_get_errmsg" on page 264
Gets an error message that corresponds to an error structure that is obtained by the **cu_get_error** subroutine.

"cu_get_error" on page 265
Gets detailed error information about errors that are returned by RMC API subroutines.

"cu_pkg_error, cu_vpkg_error" on page 267
Package error information into a cluster error structure or create cluster error structures that are returned by microsensor API subroutines.

"cu_rel_errmsg" on page 269
Frees the memory associated with a message that is obtained by the **cu_get_errmsg** subroutine.

Microsensor API concepts

Before you use calls to the microsensor API in an application, you must understand basic concepts. The microsensor API includes subroutines, macros, datatypes, and other data definitions you can use to create microsensors.

RSCT provides a C header file called **ct_microsensor.h** for microsensor implementers to include. This header file defines various datatypes, enumeration values, numeric constants, subroutines, and other data definitions. In order to use the microsensor API, the header file must be included as follows:

```
#include <rsct/ct_microsensor.h>
```

Comparing sensors and microsensors

RSCT microsensors are extensions of RSCT sensors.

A sensor is a command that the RMC subsystem runs to retrieve one or more user-defined values. A microsensor is a dynamically-loaded shared library that provides C-based functions invoked by the RMC subsystem's **IBM.MicroSensorRM** daemon to obtain control information and values for monitored dynamic attributes of the **IBM.MicroSensor** resource class.

The **IBM.SensorRM** and the **IBM.MicrosensorRM** processes run as **root**, so sensors and microsensors execute with **root** privileges.

The main difference between a sensor and a microsensor is that the sensor code is executed in a process outside the **IBM.SensorRM** process, while the microsensor code is executed inside the **IBM.MicrosensorRM** process. This creates a distinctly different paradigm when writing microsensors.

Microsensors do not need to spawn a new process and parse the output, so they are more efficient than sensors. In addition, the microsensor implementer can provide a file descriptor that the microsensor resource manager can use in a **select()** call to eliminate polling.

Programming errors in sensor code only affect the execution of a specific sensor. Programming errors that occur in microsensor code affect the execution of all microsensors.

Specifying the attributes that microsensors support

The microsensor API includes a subroutine for specifying which attributes the microsensor resource manager supports.

Use the **usf_get_control_data()** subroutine to specify which attributes the microsensor resource manager supports.

Related reference:

"usf_get_control_data" on page 277
Specifies which attributes the microsensor resource manager supports.

Instructing microsensors to start or stop fetching dynamic attribute values

The microsensor API includes subroutines for instructing microsensors to start and stop fetching standard and custom dynamic attribute values.

Use the **usf_start_standard_dattr** subroutine to instruct a microsensor to start fetching standard dynamic attribute values from the microsensor instance.

Use the **usf_stop_standard_dattr** subroutine to instruct a microsensor to stop fetching standard dynamic attribute values from the microsensor instance.

Use the **usf_start_custom_dattr** subroutine to instruct a microsensor to start fetching custom dynamic attribute values from the microsensor instance.

Use the **usf_stop_custom_dattr** subroutine to instruct a microsensor to stop fetching custom dynamic attribute values from the microsensor instance.

Related reference:

"usf_start_standard_dattr" on page 283
Instructs the microsensor to start fetching dynamic attribute values from the microsensor instance.

"usf_stop_standard_dattr" on page 285
Instructs the microsensor to stop fetching standard dynamic attribute values from the microsensor instance.

"usf_start_custom_dattr" on page 282
Acquires a custom dynamic attribute value from the microsensor instance.

"usf_stop_custom_dattr" on page 284
Stops acquiring custom dynamic attribute values from the microsensor instance.

Passing dynamic attribute values and other information to the microsensor resource manager

This section specifies the subroutines used to pass standard and custom dynamic attribute values and information to the microsensor resource manager.

Use the **usf_get_standard_dattr_values** subroutine to pass standard dynamic attribute values to the microsensor resource manager.

Use the **usf_get_custom_dattr_values** subroutine to pass custom dynamic attribute values to the microsensor resource manager.

Use the **usf_get_custom_dattr_info** subroutine to pass custom dynamic attribute information to the microsensor resource manager.

Releasing a microsensor instance

The microsensor API includes a subroutine for releasing a microsensor instance.

Use the **usf_fini** subroutine to release a microsensor instance. For more information about this subroutine, see "usf_fini" on page 276.

Writing safe microsensors

When writing a microsensor, or when accepting a microsensor from third-party software vendors (other than IBM), make sure that the microsensor code does exactly what it intended to do (and nothing more), that the execution of the microsensor code does not impact any of the microsensor resource manager's functions, and that the microsensor code is free of security vulnerabilities.

To ensure microsensor safety, it is essential that you adhere to the following guidelines:

- Microsensors must not create threads, processes, or call functions such as **exec()** that replace the **IBM.MicrosensorRM** process and, in general, must be tolerant of a multi-threaded environment. Assume that a microsensor could terminate at any time, without invocation of its **usf_fini()** routine.
- Microsensors must not set or change the disposition of signal.
- Microsensors must not include such process-ending calls as **exit()**.
- Microsensors must not change the process user IDs.

Multi-threading tolerance

Microsensors must be sufficiently tolerant to run successfully in multi-threading environments that they do not control.

To assure multi-threading tolerance, you need to adhere to the following principles when creating microsensors:

- Microsensors must not create threads. The purpose of a microsensor is to increase performance. This requires that it be small, simple and efficient. The execution of microsensor code must be entirely under the control of the microsensor resource manager in such a way that it can unload the microsensor safely.
- Microsensors must not create child or other processes. The interface defined by RMC and exported by the microsensors does not provide the means to control such processes. Microsensors should not call functions such as **fork()** or **exec()**, which duplicate the **IBM.MicrosensorRM** process or create a new process.
- Microsensors must be expected to run in a thread that can be cancelled at any time. Microsensors must be able to handle thread-cancellation requests safely and in a timely manner. When writing microsensors, use **pthread_testcancel()**, **pthread_cleanup_push()**, and **pthread_cleanup_pop()** to make sure that memory or file descriptors are not leaked when the thread in which they run is being cancelled. Also, pay attention to system calls that are thread cancellation points and make sure system resources are released when the thread of execution is cancelled.

Well-behaved microsensors

Microsensors are loadable objects that the microsensor resource manager loads in its process space. Consequently, they affect the overall behavior of the microsensor resource manager.

To maximize control over their behavior, you need to adhere to the following principles when creating microsensors:

- Microsensors must not change the disposition of signals. Signals affect the behavior of the entire process, and shared libraries or other modules that are loaded and executed in another process space cannot make any assumption about how the program loading them handles signals.
- Microsensors must not change the identity of the process in which they run. Microsensors cannot change the user identity (real, effective, and saved) of the process or the thread they in which they run. Microsensors are expected to run with root privileges in the IBM.MicrosensorRM process.
- Microsensors must not call process exiting routines, such as **exit()** and **abort()**. Like any loadable object or shared library, microsensors must not terminate the process in which they run.
- Microsensors must return clear and unambiguous return codes. Each error condition encountered by the exported interface must be clearly identified by a distinct return code that is thoroughly described in the microsensor documentation.

Secure microsensors

Microsensors run with **root** privileges inside the **IBM.MicrosensorRM** process. Make sure that their code is secure and does not introduce security vulnerabilities.

To assure their security, adhere to the following principles when developing microsensors:

- Do not use environment variables for a microsensor's functionality. Microsensors are intended to be simple and efficient, running their code self-contained in a single thread. Because environment variables affect the entire process, their use should be limited only to testing. Remove them when you complete testing, and before releasing the microsensor into a production environment.
- Handle files and directories with care. Microsensor code runs with **root** privileges. As a result, any operation on files and directories, especially operations that create or write to files, must ensure that system files are not destroyed or overwritten. If a microsensor needs to create files using well-known names, make sure the file does not exist. If the file exists, make sure it is not unexpected type. Writing to the file should be done in a safe manner, at a minimum, a manner that would ensure a small amount of space remains on the device.
- Create temporary files with random file names. If a microsensor needs to create a temporary file for whatever reason, use the **mkstemp()** system call to get a file descriptor to a new file with a unique file name. Make sure temporary files are removed when the **usf_fini()** subroutine is called.
- Check external input, boundary checks, within the range values. If external input is read from a file, the microsensor code must check the validity of that input, whether it comes from a regular file that only **root** can write to, whether it conforms to the defined format, and whether all values are within established boundaries.
- Limit memory allocations to what appears to be acceptable memory sizes. Do not allocate memory indiscriminately, especially as a result of parsing external input. Verify that the size of memory to allocate is within specified limits that make sense for how the allocated memory is to be used. Pay attention to allocating zero bytes, as results are undefined.
- Use datatypes consistently. Pay attention to signed data versus unsigned data, especially when dealing with pointers. Do not use unsigned data in the place of signed data, or the reverse. Use well-sized datatypes (for example, use **short** instead of **int** for port numbers).

Microsensor certification process

The purpose of microsensor certification is to ensure that each microsensor complies with good, secure engineering practices and that it does not introduce security vulnerabilities. This certification is often, but not always, provided by some form of external review, education, or assessment.

For microsensors, this certification is based on either of the following:

- Trust, if the microsensor is provided by a trusted software vendor, such as IBM.
- A review of the microsensor's functionality and implementation, and an assessment of its security posture. It is recommended that you independently review and assess all microsensors. Microsensors that you supply are easily reviewed and assessed by system administrators, since they have access to the source code, as well as to the developers that designed and implemented the microsensor.

Reviewing and assessing a microsensor

The task of reviewing and assessing a microsensor is the most important part of the certification process.

Once you are comfortable that a microsensor is tolerant of multi-threading, well-behaved, safe, and secure, run a command that certifies that specific copy or version of the microsensor. Reviewers who have access to a microsensor's code must follow the guidelines listed in “Writing safe microsensors” on page 273 to determine its compliance with multi-threading tolerance, well-behavior, and security. This requires a thorough code review. Reviewers who do not have access to a microsensor's code must ask the microsensor's provider pertinent questions to determine its compliance with multi-threading tolerance, behavior, and security requirements.

Certifying a microsensor

There are two ways to certify a microsensor module.

The first occurs when a microsensor is created. The second occurs when there is a need to re-certify a microsensor module because it was marked as unusable, updated, or upgraded.

Creating a microsensor:

Use the **mksensor** command to create a microsensor. Each time a new microsensor is created, the microsensor resource manager certifies the microsensor module.

The administrator creates a microsensor by running the **mksensor** command and specifying the module name as a parameter to the command:

```
mksensor -m microsensor_name microsensor_module_path
```

where *microsensor_name* is the name of the microsensor and *microsensor_module_path* is the full path of the microsensor module. The microsensor resource manager performs the following tasks before allowing the creation of the new microsensor:

1. Checks that the module exists and is a regular file.
2. Verifies that the module is not writable by users other than the owner.
3. Calculates and stores the module digital signature.

If all of these tasks succeed, the module is considered certified.

When the microsensor resource manager loads the microsensor module, its signature is calculated again and compared with that of the microsensor module that was stored originally. If the calculated signature does not match the stored signature, the microsensor resource manager marks the module in a manner that signifies it as unusable by each of the microsensors that might refer to it.

If the microsensor resource manager finds a microsensor module with an unrecoverable problem, it marks the module as unusable. To make such a microsensor module usable again, you must re-certify it. If the unusable module is loaded, the **Reload** action of the **IBM.MicroSensor** resource class must also be called so that the existing module is unloaded, then the new module is loaded. The module status is changed accordingly after the module is reloaded.

The **Reload** resource class action reloads the passed module. If the module is currently loaded, it is unloaded first, and calls to **usf_fini()** are made as appropriate, then the module is reloaded into memory.

Re-certifying a microsensor module:

To re-certify a microsensor module, the administrator must call the **Certify** action of the **IBM.MicroSensor** resource class.

To re-certify a microsensor module, enter:

```
runact -c IBM.MicroSensor Certify ModuleName=module_name
```

In this example, *module_name* is the name of the microsensor module that needs to be re-certified.

The **Certify** resource class action updates the signature value of all **IBM.MicroSensor** resources that use the passed module.

Obtaining error information returned by the microsensor API subroutines

All of the microsensor API subroutines return a **cu_error_t** structure pointer.

The format of the **cu_error_t** structure pointer follows:

```
typedef struct {
    ct_int32_t   cu_error_id;  /* error value */
    ct_char_t  *cu_ffdc_id;  /* FFDC ID library logged entry */
    ct_char_t  *cu_msg_cat;  /* message catalog name */
    ct_int32_t   cu_msg_set;  /* message catalog set */
    ct_int32_t   cu_msg_num;  /* message number */
    ct_char_t  *cu_msg_default; /* default message */
    ct_uint32_t   cu_arg_cnt;  /* count of error arguments */
    cu_error_arg_t *cu_args;  /* array of error arguments */
} cu_error_t;
```

The microsensor API subroutines can fill this structure in using the **cu_pkg_error()** subroutine (described in "cu_pkg_error, cu_vpkg_error" on page 267). The microsensor resource manager releases these structures as needed using the **cu_rel_error()** subroutine (described in "cu_rel_error" on page 270). A Null **cu_error_t** pointer value indicates success, while a non-Null **cu_error_t** pointer value indicates that an error occurred. If an error occurs, most of the microsensor API subroutines will assume the module is unusable and will fence it.

Microsensor API reference

The microsensor API includes several subroutines and data types you can use to create microsensors.

Microsensor API subroutines

The microsensor API includes several subroutines for performing various tasks related to creating microsensors.

usf_fini

Releases the microsensor instance when the microsensor is no longer needed.

Syntax

```
#include <rsct/ct_microsensor.h>

cu_error_t*
usf_fini(
    void           *anchor_p,
    ct_uint_32_t    unload_flag
);
```

Parameters

Input

anchor_p
: is the anchor pointer for the microsensor instance.

unload_flag
: if non-zero, indicates that the resource manager will unload the microsensor upon return of this subroutine.

Description

Called by: the microsensor resource manager

The **usf_fini** subroutine releases the microsensor instance when the microsensor is no longer needed. The microsensor is expected to free the anchor object. If the value of *unload_flag* is not zero, the microsensor is expected to perform any necessary clean-up appropriate for the module to be unloaded. Any error that is returned by this subroutine is logged.

usf_get_control_data

Specifies which attributes the microsensor resource manager supports.

Syntax

```
#include <rsct/ct_microsensor.h>

cu_error_t*
usf_get_control_data(
    ct_int32_t   argc,
    char         **argv,
    usf_control_data *control_data_p
);
```

Parameters

Input

argc is a count of entries in the argument vector.

argv is a pointer to an array of pointers to strings.

Output

control_data_p
is a pointer to a structure of type **usf_control_data_t**.

Description

Called by: the microsensor resource manager

The **usf_get_control_data** subroutine is used to specify which attributes the microsensor resource manager supports. This is the first function call that the microsensor resource manager makes to the microsensor after it is loaded. If the information returned by the microsensor indicates that it is re-entrant, this call is made for each **IBM.MicroSensor** resource for which it is specified.

The *argv* parameter contains *argc* + 1 entries, where the last entry is a Null pointer. The first element in this array is the path name of the microsensor to load. In the microsensor Arguments string, each subsequent element is derived from the microsensor Arguments attribute. Each token, separated by white space, becomes a string in the argument vector in the same order. This array is read-only for the microsensor. It will be de-allocated by the microsensor resource manager after this call is made.

The format of the **usf_control_data** structure is:

```
typedef struct usf_control_data
{
    ct_uint32_t    usf_API_version;
    ct_uint32_t    usf_uSensor_version;
    ct_uint64_t    usf_standard_dattrs;
    void          *usf_anchor;
    ct_uint32_t    usf_num_custom_dattrs;
} usf_control_data_t;
```

The structure fields of the **usf_control_data_t** datatype are set as follows by the subroutine:

- The *usf_API_version* field is set to the value **USF_API_VERSION**. The microsensor resource manager uses this field to determine the version of the microsensor API that is implemented by the microsensor.
- The *usf_uSensor_version* field is set to a value that is indicative of the version of the microsensor. This value should change whenever the semantic of the microsensor changes, that is, when new custom dynamic attributes are added or there is a change in support of the standard dynamic resource attributes, for example. The binary format of the value is opaque to the resource manager, other than the value for each successive microsensor version, which must compare numerically higher than the value for the prior version.
- If the microsensor supports one or more of the standard dynamic resource attributes of the **IBM.MicroSensor** resource class, the appropriate bits are set in the *usf_standard_dattrs* field.
- If the microsensor is re-entrant, the *usf_anchor* field is set to a non-Null pointer; otherwise, it is set to Null. It is assumed that this pointer refers to a control block allocated by the microsensor. The pointer is saved with its resource and is passed on all subsequent API calls for the same microsensor instance.
- The *usf_num_custom_dattrs* field is set to the number of custom dynamic attributes that are supported by the microsensor.

If the microsensor is not re-entrant, an error will occur if more than one resource uses it.

If this subroutine returns an error, the microsensor will be rendered unusable.

Related concepts:

"Specifying the attributes that microsensors support" on page 271
The microsensor API includes a subroutine for specifying which attributes the microsensor resource manager supports.

usf_get_custom_dattr_info

Gets information about the custom dynamic attributes of a microsensor.

Syntax

```
#include <rsct/ct_microsensor.h>
cu_error_t *
usf_get_custom_dattr_info(
    void                    anchor_p,
    ct_uint32_t             num_values,
    usf_custom_dattr_info_t dattr_info[]
 );
```

Parameters

Input

anchor_p
: is the anchor pointer for the microsensor instance.

num_values
: is the number of custom dynamic attributes (*N*) for which the microsensor resource manager is acquiring information.

Output

dattr_info[]
: is a pointer to an array of structures of type **usf_custom_dattr_info_t**.

Description

Called by: the microsensor resource manager

The **usf_get_custom_dattr_info** subroutine is used to obtain information about the custom dynamic attributes of a microsensor. It is called only in cases where the microsensor specifies that it supports at least one custom dynamic attribute. Also, it is called for each microsensor instance if the microsensor is re-entrant.

If the value of the **usf_control_data_t** datatype's *usf_num_custom_dattrs* field is *N*, where $N > 0$, it is assumed that each custom dynamic attribute has a unique attribute ID (internal to the microsensor instance) that is taken from the set {1, *N*}. The internal attribute ID of the custom dynamic attributes is its index in the *dattr_info* array +1. This internal attribute ID is used by the microsensor resource manager to reference the various custom dynamic attributes in this and other microsensor API calls. For more information about **usf_control_data_t**, see "usf_get_control_data" on page 277.

The format of the **usf_custom_dattr_info** structure is:

```
typedef struct usf_custom_dattr_info
{
    char          *usf_name;
    ct_int32_t     usf_data_type;
    ct_int32_t     usf_variable_type;
    ct_int32_t     usf_reporting_interval;
    ct_uint32_t    usf_properties;
} usf_custom_dattr_info_t;
```

The subroutine sets the structure fields of the **usf_custom_dattr_info_t** datatype as follows:

- The *usf_name* field is set to a pointer to the name of the custom dynamic attribute. This is the programmatic name of the attribute, which must begin with an alphabetic character and must be taken from the portable character set as defined in *IEEE Std 1003.1, 2004 Edition*. It can only contain these characters: **0** to **9**, **A** to **Z**, **a** to **z**, and _ (underscore). The pointer is assumed to reference static data that is copied by the microsensor resource manager.
- The *usf_data_type* field is set to the datatype of the attribute. The value is taken from the **ct_data_type_t** enumeration. For more information about **ct_data_type_t**, see "RMC API data types for values, resources, and resource attributes" on page 238.
- The *usf_variable_type* field is set to the variable type of the attribute. The variable type indicates the semantics of the attribute: **Counter**, **Quantity**, **Quantum**, or **State**. The value is taken from the **rmc_variable_type_t** enumeration. For more information about **rmc_variable_type_t**, see "RMC API data types for values, resources, and resource attributes" on page 238.
- The *usf_reporting_interval* field is set to the default time interval, in seconds, during which the microsensor resource manager fetches the custom dynamic attribute value from the microsensor. If this value is **0**, the microsensor might be queried for a new value when a file descriptor, supplied by the microsensor, is ready to be read. If this value is not **0** and no file descriptor is supplied, it is assumed a new value is pushed into the microsensor resource manager through a call to **usf_get_custom_dattr_values**. If the file descriptor value is not **0** and a reporting interval other than **0** is specified, an error is logged and the module is made unusable. If the *usf_reporting_interval* value is **0** and no file descriptor is specified, the module will obtain its values when the **Refresh** resource action is called on this particular microsensor. A **Refresh** resource action updates the signature value of all **IBM.MicroSensor** resources that use the passed module.

  The reporting interval can be changed by the RMC API client application.
- The *usf_properties* field is a bitfield that specifies the properties of the custom dynamic attribute in addition to the public property. The only property that can be specified is **QryReqsMonitoring**. To specify this property, the **USF_RSRC_DATTR_QRY_REQS_MONITORING** bit must be set in this field. If the value is **0**, the only property for this attribute is public.

Related reference:

"usf_get_custom_dattr_values" on page 280
Passes custom dynamic attribute values to the microsensor resource manager.

usf_get_custom_dattr_values

Passes custom dynamic attribute values to the microsensor resource manager.

Syntax

```
#include <rsct/ct_microsensor.h>

cu_error_t
usf_get_custom_dattr_values(
    void               *anchor_p,
    ct_uint32_t         num_values,
    usf_attribute_t     values[]
);
```

Parameters

Input

anchor_p
: is the anchor pointer for the microsensor instance.

num_values
: is the number of custom dynamic attributes for which the microsensor resource manager is acquiring a value.

Output

values[]
: is a pointer to an array of type **usf_attribute_t**.

Description

Called by: the microsensor resource manager

The **usf_get_custom_dattr_values** subroutine is used to obtain values for each custom dynamic attribute from the microsensor instance. It is called only in the case where the microsensor specifies that it supports at least one custom dynamic attribute.

This subroutine is called under one of the following conditions:

- The reporting interval is not **0** and the interval time has elapsed.
- The reporting interval is **0**, a non-negative file descriptor has been supplied by this microsensor and the descriptor is ready for read.
- The reporting interval is **0**, no file descriptor is supplied, and the **Refresh** resource action is called on this microsensor resource.

  A **Refresh** resource action updates the signature value of all **IBM.MicroSensor** resources that use the passed module.

The format of the **usf_attribute** structure is:

```
typedef struct usf_attribute
{
    rmc_attribute_id_t  attribute_id;
    ct_data_type_t      data_type;
    ct_value_t          value;
} usf_attribute_t;
```

The structure fields of the **usf_attribute_t** datatype are set as follows:

- The microsensor resource manager sets the *attribute_id* field to the ID of the custom dynamic attribute value that is being requested.

- The microsensor resource manager sets the *data_type* field to the datatype of the custom dynamic attribute value that is being requested. The valid values are: **USF_DATTR_FLOAT32_ID**, **USF_DATTR_FLOAT32_ARRAY_ID**, **USF_DATTR_FLOAT64_ID**, **USF_DATTR_FLOAT64_ARRAY_ID**, **USF_DATTR_INT32_ID**, **USF_DATTR_INT32_ARRAY_ID**, **USF_DATTR_INT64_ID**, **USF_DATTR_INT64_ARRAY_ID**, **USF_DATTR_QUANTUM_ID**, **USF_DATTR_STRING_ID**, **USF_DATTR_STRING_ARRAY_ID**, **USF_DATTR_UINT32_ID**, **USF_DATTR_UINT32_ARRAY_ID**, **USF_DATTR_UINT64_ID**, and **USF_DATTR_UINT64_ARRAY_ID**.
- The microsensor sets the *value* field to the *ct_value* of the custom dynamic attribute value that is being requested.

If the datatype of the attribute is scalar, the value is placed directly in the memory addressed by the *value* field. Otherwise, the memory that contains the aggregate type (array, binary, string, or structured data) must be allocated and set by the subroutine and a pointer to the aggregate type placed in the memory addressed by the *value* field. The microsensor resource manager will eventually free such allocated memory.

If a specified dynamic attribute's datatype is **USF_DATTR_QUANTUM_ID**, the *val_int32* field of the *value* field is set to a non-zero value to assert the attribute's **Quantum** variable type. If the field is set to **0**, the attribute's **Quantum** variable type is not asserted, that is, no event is generated.

Related reference:

"usf_get_custom_dattr_info" on page 278
Gets information about the custom dynamic attributes of a microsensor.

"usf_start_custom_dattr" on page 282
Acquires a custom dynamic attribute value from the microsensor instance.

"usf_stop_custom_dattr" on page 284
Stops acquiring custom dynamic attribute values from the microsensor instance.

usf_get_standard_dattr_values

Passes standard dynamic attribute values to the microsensor resource manager.

Syntax

```
#include <rsct/ct_microsensor.h>

cu_error_t*
usf_get_standard_dattr_values(
    void                 *anchor_p,
    ct_uint32_t           num_values,
    usf_attribute_t       values[]
);
```

Parameters

Input

anchor_p
is the anchor pointer for the microsensor instance.

num_values
is the number of elements in the *values*[] array.

Input/Output

values[]
is an array of **usf_attribute** structures.

Description

Called by: the microsensor resource manager

The **usf_get_standard_dattr_values** subroutine is used to pass standard dynamic attribute values to the microsensor resource manager. The microsensor resource manager calls this subroutine to obtain standard dynamic attribute values from the microsensor instance.

This subroutine is called under one of the following conditions:

- The refresh interval is not **0** and the interval time has elapsed.
- The refresh interval is **0**, a non-negative file descriptor has been supplied by this microsensor and the descriptor is ready for read.
- The refresh interval is **0**, no file descriptor is supplied, and the **Refresh** resource action is called on this microsensor resource.

  A **Refresh** resource action updates the signature value of all **IBM.MicroSensor** resources that use the passed module.

The format of the **usf_attribute** structure is:

```
typedef struct usf_attribute
{
    rmc_attribute_id_t  attribute_id;
    ct_data_type_t      data_type;
    ct_value_t          value;
} usf_attribute_t;
```

The structure fields of the **usf_attribute_t** datatype are set as follows:

- The microsensor resource manager sets the *attribute_id* field to the ID of the standard dynamic attribute value that is being requested.
- The microsensor resource manager sets the *data_type* field to the datatype of the standard dynamic attribute value that is being requested. The valid values are: **USF_DATTR_FLOAT32_ID**, **USF_DATTR_FLOAT32_ARRAY_ID**, **USF_DATTR_FLOAT64_ID**, **USF_DATTR_FLOAT64_ARRAY_ID**, **USF_DATTR_INT32_ID**, **USF_DATTR_INT32_ARRAY_ID**, **USF_DATTR_INT64_ID**, **USF_DATTR_INT64_ARRAY_ID**, **USF_DATTR_QUANTUM_ID**, **USF_DATTR_STRING_ID**, **USF_DATTR_STRING_ARRAY_ID**, **USF_DATTR_UINT32_ID**, **USF_DATTR_UINT32_ARRAY_ID**, **USF_DATTR_UINT64_ID**, and **USF_DATTR_UINT64_ARRAY_ID**.
- The microsensor sets the *value* field to the *ct_value* of the standard dynamic attribute value that is being requested.

If the datatype of the attribute is scalar, the value is placed directly in the memory addressed by the *value* field. Otherwise, the memory that contains the aggregate type (a string), must be allocated and set by the microsensor and a pointer to the aggregate type placed in the memory addressed by the *value* field. The microsensor resource manager will eventually free such allocated memory.

If a specified dynamic attribute's datatype is **USF_DATTR_QUANTUM_ID**, the *val_int32* field of the *value* field is set to a non-zero value to assert the attribute's **Quantum** variable type. If the field is set to **0**, the attribute's **Quantum** variable type is not asserted, that is, no event is generated.

usf_start_custom_dattr

Acquires a custom dynamic attribute value from the microsensor instance.

Syntax

```
#include <rsct/ct_microsensor.h>
```

```
cu_error_t*
usf_start_custom_dattr(
    void           *anchor_p,
    ct_uint32_t     dattr_ID,
    int            *fd_p
);
```

Parameters

Input

anchor_p
is the anchor pointer for the microsensor instance.

dattr_ID
is the internal attribute ID of the custom dynamic attribute for which the microsensor resource manager will be acquiring values.

Output

fd_p is a pointer to a memory location in which the microsensor optionally returns a file descriptor.

Description

Called by: the microsensor resource manager

The **usf_start_custom_dattr** subroutine is called by the microsensor resource manager to indicate that it will acquire a custom dynamic attribute value from the microsensor instance. It is called only in the case where the microSensor specifies it supports at least one custom dynamic attribute.

Microsensors use this subroutine to allocate any resources necessary to supply the attribute value - for allocation of memory, opening a file, or creating a socket, for example.

Typically, the microsensor resource manager calls the microsensor at some regular interval to fetch the value of a monitored dynamic attribute. If this subroutine returns a non-negative value at the location specified by the *fd_p* parameter, the microsensor resource manager interprets this value as a descriptor to be used in a **poll()** or **select()** system call.

When the descriptor is ready for read, the microsensor resource manager calls the microsensor to fetch the value of the attribute specified by the *dattr_ID* parameter. It is assumed that the descriptor is suitable for use in **poll()** or **select()** system calls, that is, it represents a socket or special device file. The same descriptor can be returned in other calls to the **usf_start_custom_dattr** subroutine or the **usf_start_standard_dattr** subroutine. The microsensor resource manager keeps track of how many times the same descriptor is returned.

Related concepts:

"Instructing microsensors to start or stop fetching dynamic attribute values" on page 272
The microsensor API includes subroutines for instructing microsensors to start and stop fetching standard and custom dynamic attribute values.

Related reference:

"usf_get_custom_dattr_values" on page 280
Passes custom dynamic attribute values to the microsensor resource manager.

"usf_stop_custom_dattr" on page 284
Stops acquiring custom dynamic attribute values from the microsensor instance.

usf_start_standard_dattr

Instructs the microsensor to start fetching dynamic attribute values from the microsensor instance.

Syntax

```
#include <rsct/ct_microsensor.h>

cu_error_t*
usf_start_standard_dattr(
    void           *anchor_p,
    ct_int32_t      dattr_ID,
    int            *fd_p
);
```

Parameters

Input

anchor_p
is the anchor pointer for the microsensor instance.

dattr_ID
is a bitfield that contains the attribute ID of the standard dynamic attribute for which the resource manager will be acquiring values.

Output

fd_p is a pointer to a memory location in which the microsensor optionally returns a file descriptor.

Description

Called by: the microsensor resource manager

The **usf_start_standard_dattr** subroutine is used to notify the microsensor to start fetching dynamic attribute values from the microsensor instance. The microsensor uses this subroutine to allocate any resources necessary to supply the attribute values for allocation of memory, opening a file, create a socket, for example.

Typically, the microsensor resource manager calls the microsensor at some regular interval to fetch the value of a monitored dynamic attribute. If this subroutine returns a non-negative value at the location specified by the *fd_p* parameter, the microsensor resource manager interprets this value as a descriptor to be used in a **poll()** or **select()** system call.

When the descriptor is ready for read, the microsensor resource manager calls the microsensor to fetch the value of the attribute specified by the *dattr_ID* parameter. It is assumed that the descriptor is suitable for use in **poll()** or **select()** system calls, that is, it represents a socket or special device file. The same descriptor can be returned in other calls to the **usf_start_custom_dattr** subroutine or the **usf_start_standard_dattr** subroutine. The microsensor resource manager tracks how many times the same descriptor is returned.

If this function returns an error, the microsensor will be rendered unusable.

Related concepts:

"Instructing microsensors to start or stop fetching dynamic attribute values" on page 272
The microsensor API includes subroutines for instructing microsensors to start and stop fetching standard and custom dynamic attribute values.

usf_stop_custom_dattr

Stops acquiring custom dynamic attribute values from the microsensor instance.

Syntax

```
#include <rsct/ct_microsensor.h>

cu_error_t*
usf_stop_custom_dattr(
    void           *anchor_p,
    ct_int32_t      dattr_ID
);
```

Parameters

Input

anchor_p
is the anchor pointer for the microsensor instance.

dattr_ID
is the internal attribute ID of the custom dynamic attribute for which the microsensor resource manager will no longer be acquiring values.

Description

Called by: the microsensor resource manager

The **usf_stop_custom_dattr** subroutine is called by the microsensor resource manager to indicate that it will no longer be acquiring a custom dynamic attribute value from the microsensor instance. It is called only in the case where the microsensor specifies that it supports at least one custom dynamic attribute.

The microsensor uses this subroutine to free any resources obtained by the call to the **usf_start_custom_dattr** function.

Any descriptor returned for the specified attribute is no longer used in a **poll()** or **select()** system call by the microsensor resource manager, unless the descriptor is referenced by other dynamic attributes still being monitored.

Related concepts:

"Instructing microsensors to start or stop fetching dynamic attribute values" on page 272
The microsensor API includes subroutines for instructing microsensors to start and stop fetching standard and custom dynamic attribute values.

Related reference:

"usf_get_custom_dattr_values" on page 280
Passes custom dynamic attribute values to the microsensor resource manager.

"usf_start_custom_dattr" on page 282
Acquires a custom dynamic attribute value from the microsensor instance.

usf_stop_standard_dattr

Instructs the microsensor to stop fetching standard dynamic attribute values from the microsensor instance.

Syntax

```
#include <rsct/ct_microsensor.h>

cu_error_t*
usf_stop_standard_dattr(
    void           *anchor_p,
    ct_int32_t      dattr_ID,
);
```

Parameters

Input

anchor_p
is the anchor pointer for the microsensor instance.

dattr_ID
is a bitfield that contains the attribute ID of the standard dynamic attribute for which the resource manager will be acquiring values.

Description

Called by: the microsensor resource manager

The **usf_stop_standard_dattr** subroutine is used to indicate that the microsensor resource manager will no longer fetch standard dynamic attribute values from the microsensor instance.

The microsensor uses this subroutine to free any resources obtained by the call to the **usf_start_standard_dattr** subroutine.

Any descriptor returned for the specified attribute is no longer used in a **poll()** or **select()** system call by the microsensor resource manager, unless the descriptor is referenced by other dynamic attributes still being monitored.

If this subroutine returns an error, the microsensor will be rendered unusable.

Related concepts:

"Instructing microsensors to start or stop fetching dynamic attribute values" on page 272
The microsensor API includes subroutines for instructing microsensors to start and stop fetching standard and custom dynamic attribute values.

Microsensor API data types

The microsensor API uses a number of data types, which are summarized in the following table.

The following table lists the data types that are associated with the microsensor API.

Table 38. Microsensor API data types

Data type	Description	For more information, see:
ct_data_type_t	An enumeration used to identify the data types, or pointers to data types, of values that are used by the microsensor API.	"RMC API data types for values, resources, and resource attributes" on page 238
ct_int32_t	Specifies a scalar value of type **int32_t**.	"RMC API data types for values, resources, and resource attributes" on page 238
ct_uint32_t	Specifies a scalar value of type **uint32_t**.	"RMC API data types for values, resources, and resource attributes" on page 238
ct_uint64_t	Specifies a scalar value of type **uint64_t**.	"RMC API data types for values, resources, and resource attributes" on page 238
ct_value_t	A union used to return resource attribute values.	"RMC API data types for values, resources, and resource attributes" on page 238
cu_error_t	A structure pointer for subroutines that handle cluster software error information in a common manner.	"Obtaining error information returned by the microsensor API subroutines" on page 276

Table 38. Microsensor API data types (continued)

Data type	Description	For more information, see:
rmc_attribute_id_t	A data type for attribute IDs.	"RMC API data types for values, resources, and resource attributes" on page 238
usf_attribute_t	Specifies basic information about resource attributes.	"usf_get_custom_dattr_values" on page 280, "usf_get_standard_dattr_values" on page 281
usf_control_data_t	Specifies basic information about microsensors.	"usf_get_control_data" on page 277
usf_custom_dattr_info_t	Specifies basic information about custom dynamic attributes.	"usf_get_custom_dattr_info" on page 278

Notices

This information was developed for products and services offered in the U.S.A.

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

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

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

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

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

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

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

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

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

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this

one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Dept. LRAS/Bldg. 903
11501 Burnet Road
Austin, TX 78758-3400
U.S.A.

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

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

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

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

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Privacy policy considerations

IBM® Software products, including software as a service solutions, (“Software Offerings”) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as the customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM’s Privacy Policy at http://www.ibm.com/privacy and IBM’s Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled “Cookies, Web Beacons and Other Technologies” and the “IBM Software Products and Software-as-a-Service Privacy Statement” at http://www.ibm.com/software/info/product-privacy.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Windows is a trademark of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Index

P

R

S

T

V

Printed in USA