IBM MQSeries Workflow for OS/390

Programming Guide

Version 3 Release 1

SC33-7031-00

IBM MQSeries Workflow for OS/390

Programming Guide

Version 3 Release 1

SC33-7031-00

Note!
Before using this information and the product it supports, be sure to read the general information under "Notices" on page xii.

First Edition (March 1999)

This edition applies to Version 3, Release 1, Modification 0 of IBM MQSeries Workflow for OS/390 (product number 5565-A96) and to all subsequent releases and modifications until otherwise indicated in new editions.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

IBM welcomes your comments. A form for your comments appears at the back of this publication. If the form has been removed, address your comments to:

IBM Deutschland Entwicklung GmbH
Department 3248
Schoenaicher Strasse 220
D-71032 Boeblingen
Federal Republic of Germany

FAX (Germany): 07031+16-3456
FAX (Other Countries): (+49)+7031-16-3456

IBM Mail Exchange: DEIBMBM9 at IBMMAIL
Internet: s390id@de.ibm.com

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

About this book

This book describes how to use the IBM MQSeries Workflow for OS/390 Client Application Programming Interfaces, hereafter called the MQSeries Workflow APIs. The first part of the book describes the concepts underlying the APIs while the rest of the book provides for an API reference manual. The book also describes the MQSeries Workflow predefined data structures, the MQSeries Workflow audit trail function, and how to debug applications running under the control of MQSeries Workflow. The main change in this book in comparison with the last version is "Part 4. Program Execution Server's Program Mapping" on page 97. It describes how to use program mappings in order to bring Workflow API containers into a format acceptable by legacy applications. Additionally there are the PES exits (mapping and invocation) described.

Note: The licensed books that were declassified in OS/390 Version 2 Release 4 appear on the OS/390 Online Library Collection, SK2T-6700. The remaining licensed books for OS/390 Version 2 appear on the OS/390 Licensed Product library, LK2T-2499, in unencrypted form.

Who should read this book

This book is intended for programmers who design and implement programs using an MQSeries Workflow API and who may participate in designing an MQSeries Workflow workflow model. It assumes that readers are experienced OS/390 programmers, and that they understand the process modeling concepts.

How to get additional information

Visit the MQSeries Workflow home page at http://www.software.ibm.com/ts/mqseries/workflow

For a list of additional publications, refer to "MQSeries Workflow publications" on page 485.

How to send your comments

Your feedback is important in helping to provide the most accurate and high-quality information. If you have any comments about this book or any other MQSeries Workflow documentation, choose one of the following methods:

- Send your comments by e-mail to: s390id@de.ibm.com

  Be sure to include the name of the book, the part number of the book, the version of MQSeries Workflow, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).
- Fill out one of the forms at the back of this book and return it by mail, by fax, or by giving it to an IBM representative.

How this book is organized

"Notices" on page xii describes some notices and trademarks.

"Part 1. Programming Concepts" on page 1 provides an overview on how to design applications to work with the MQSeries Workflow workflow manager.

"Part 2. The MQSeries Workflow APIs" on page 5 describes the concepts underlying the MQSeries Workflow APIs.

"Part 3. OS/390 specific considerations" on page 79 describes some special OS/390 considerations for COBOL, CICS and IMS.

"Part 4. Program Execution Server's Program Mapping" on page 97 describes how to use program mappings in order to bring Workflow API containers into a format acceptable by legacy applications.

"Part 5. Program Execution Server Exits" on page 147 describes how to use mapping and invocation exits.

"Part 6. Using the MQSeries Workflow APIs" on page 169 provides for an overview on the functions/subprograms supported by the APIs.

"Part 7. Application programming interfaces" on page 213 describes the MQSeries Workflow APIs that enable applications to manipulate worklists and work items, to work with process instances and container data, and to log on to and log off from an MQSeries Workflow server.

"Part 8. Examples" on page 441 provides some examples that show how to use the APIs.

"Appendix. Audit Trail" on page 471 describes the MQSeries Workflow audit trail function.

The back of the book includes a glossary that defines terms as they are used in this book, a bibliography, and an index.

How to read the syntax diagrams

Throughout this book, syntax is described the following way; all spaces and other characters are significant:

- Read the syntax diagrams from left to right, from top to bottom, following the main path of the line.

  The ►►— symbol indicates the beginning of a statement.

  The —► symbol indicates that the statement syntax is continued on the next line.

  The ►— symbol indicates that a statement is continued from the previous line.

  The —►◄ symbol indicates the end of a statement.
- Diagrams can be broken into fragments. A fragment is indicated by vertical bars with the name of the fragment between the bars. The fragment itself follows the same syntactical rules as the main diagram.

```
►►─┤ a-fragment ├─────────────────────────────────────────────────►◄
```

- Required items appear on the horizontal line, the main path.

```
►►──required-item─────────────────────────────────────────────────►◄
```

- Optional items appear below (or above) the main path.

```
►►──required-item──┬───────────────┬──────────────────────────────►◄
                   └─optional-item─┘
```

- If you can choose from one or more items, they appear vertically, in a stack.

  If you must choose one of the items, one item of the stack appears on the main path.

```
►►──required-item──┬─required-choice1─┬───────────────────────────►◄
                   └─required-choice2─┘
```

  If choosing one of the items is optional, the entire stack appears below the main path.

```
►►──required-item──┬──────────────────┬───────────────────────────►◄
                   ├─optional-choice1─┤
                   └─optional-choice2─┘
```

- An arrow returning to the left, above the main path, indicates an item that can be repeated.

```
                   ┌──────────────────┐
                   ▼                  │
►►──required-item────repeatable-item──┴───────────────────────────►◄
```

  If the repeat arrow contains a comma, you must separate repeated items with a comma.

```
                   ┌─,────────────────┐
                   ▼                  │
►►──required-item────repeatable-item──┴───────────────────────────►◄
```

- Keywords appear in uppercase, for example, NAME. They must be spelled exactly as shown. Variables appear in lowercase italic letters, for example, *string*. They represent user-supplied values.

Notices

Trademarks

Part 1. Programming Concepts

This part provides you with a general introduction to the programming concepts of MQSeries Workflow.

Chapter 1. Understanding the programming concept

This chapter introduces the concept of workflow modeling as it relates to the design of application programs for use with the IBM MQSeries Workflow, hereafter referred to as MQSeries Workflow.

The IBM MQSeries Workflow Workflow manager provides a way to model a process and assign applications to activities in the resulting workflow model. This enables the workflow manager to automate the control of activities and the flow of data.

Work can be routed to the person who performs the activity instance. An application program required to perform an activity instance can be designed to start when a user starts an activity instance.

The role of the programmer in modeling a process

As workflow models are defined, the applications and data structures needed to support program activities are identified. Programmers can create new applications or reengineer existing applications to support these program activities.

To use existing applications with the workflow model, programmers must determine if the applications used by the enterprise can be functionally decomposed.

To integrate the applications with the workflow model, the control and flow logic are separated from the application, the start and exit conditions are moved into the workflow model, and the program is divided into modules to be invoked by the workflow manager at the appropriate points.

The resulting modules are applications that are assigned to perform the program activities defined in the workflow model.

Most applications include many diverse functions, and many can support several different activities in different stages of a process. Output produced by one function of a program can be used as input by another function of the same program. Therefore, the same application can be used to support many different program activities in a workflow model.

Your enterprise might also use vendor-written programs like word- processing or spreadsheet applications.

With IBM MQSeries Workflow for OS/390 you will be able to use mappings so you can support any legacy application with this tool. There may be old applications where you can’t change the interfaces because other applications or programs have been configured to work with these long time ago and if you change one configuration of an interface you will have to change them all. This mapper enables you to use all legacy applications with your Workflow applications via the mapping tool.

Return codes, provided by the assigned program, can then be used to evaluate exit and transition conditions.

Programming interfaces

The MQSeries Workflow workflow manager provides application program interface (API) support and a set of predefined data structure members to assist programmers who develop applications for use with workflow models. In addition, several programming samples are provided.

The MQSeries Workflow predefined data structure members provide information about the current process, activity, or block, and are associated with the operating characteristics of a process instance or activity instance.

The following MQSeries Workflow programming aids are described in this book:

- MQSeries Workflow C language API
- MQSeries Workflow Cobol language API

The MQSeries Workflow C language and Cobol language APIs provide functions/subprograms

- To execute process models, that is, to work with process instances and container data and to manipulate worklists and work items
- To monitor the progress of execution
- To issue process administrator functions
- To process container data associated with an activity implementation
- To receive information sent by an MQSeries Workflow server

Part 2. The MQSeries Workflow APIs

This part provides an overview on the concepts underlying the MQSeries Workflow C and Cobol APIs.

Chapter 2. Prerequisites for programming

MQSeries Workflow application development assumes that the appropriate environment is established. This means that:

- IBM MQSeries Workflow for OS/390 must be installed on the machine where you are developing your applications.
- A compiler of one of the supported languages is installed and configured.
- BuildTime must be installed on the machine where you are developing your applications.

Chapter 3. Building an MQSeries Workflow application

Overview

There are essentially two different tasks which you can address by using the MQSeries Workflow application programming interface (API):

- You can write your own client application instead of using the MQSeries Workflow provided GUIs (Graphical User Interfaces) or command line interfaces. For example, you may want to:
  - Control the MQSeries Workflow functionality provided to your user.
  - Present the MQSeries Workflow functionality in a way that your user is accustomed to.
  - Run selected MQSeries Workflow tasks in batch mode.
- You can write a program that implements an activity or support tool in your workflow process model.

These two kinds of programs usually contain specific parts which are discussed in chapters “Coding an MQSeries Workflow client application” on page 10 and “Coding an MQSeries Workflow activity implementation or support tool” on page 11.

The concepts underlying the MQSeries Workflow API are common to all programs using the MQSeries Workflow APIs. They are summarized here and discussed in more detail in the following chapters.

All persistent objects such as work items and process instances are accessed through transient objects which represent their state at the time when they were queried from a server. In the C and Cobol language API a so-called *handle* represents a pointer to such a transient object.

In order to request an action on an object, a session must have been established with an appropriate MQSeries Workflow server. The action itself can then be executed synchronously.

Only objects for which you are authorized are returned from the server to the client.

Separate functions in the C language API respectively subprograms in the Cobol language API, hereafter called functions/subprograms, are available for each action on an object or to access each property of an object. This approach allows function/subprogram parameters to be checked by the compiler and best represents the object-action paradigm supported by MQSeries Workflow.

Detailed error information is provided by a so-called *result object*. This object is available in addition to the return code set by action functions/subprograms. See chapter “Chapter 4. Handling errors” on page 15 for further information on the result object.

Objects are managed by the application programmer but object memory is owned by the MQSeries Workflow API. The application programmer determines the lifetime of transient objects by using allocate, or query, and deallocate mechanisms. The MQSeries Workflow API hides the internal structure of transient objects.

Coding an MQSeries Workflow client application

An MQSeries Workflow client application typically contains the following parts, not necessarily divided that clearly.

```
         ┌──      #include <MQ Workflow Api-prerequisites (c++)>
         │        #include <MQ Workflow API>
         │        int main()
         │        {
 Setup   │          Declare objects
         │                :
         │          Connect
         │          Allocate service object
         ┌──         Logon()
         ┌──
         │
         │
 Actions │          MQSeries Workflow API calls
         │
         │
         ┌──
         ┌──         Logoff()
         │          Deallocate service object
 Cleanup │          Disconnect
         │          return 0;
         ┌──      }
```

To set up your program, you typically declare the program variables or objects you are going to use and you include the MQSeries Workflow API header files or copy the copybooks you need for your actions.

You should then initialize the MQSeries Workflow API by calling the Connect() function/subprogram so that resources held by the API are allocated correctly. Connect() - and Disconnect() - are to be called at the begin respectively end of each thread.

You then need to allocate a service object which represents the server you are going to ask services from. Once the service object is allocated, you can log on. Logon establishes a session between the user logging on and the server represented by your service object. All subsequent calls requiring client/server communication run through this session.

After a successful logon, you can issue action or program execution management functions/subprograms in order to query or manage MQSeries Workflow objects you are authorized for.

At the end of your program, you log off in order to close the session to the server and you deallocate any resources held by your program, especially the service object.

As a last step, you disconnect from the MQSeries Workflow API so that resources held by the API are deallocated correctly.

Coding an MQSeries Workflow activity implementation or support tool

An MQSeries Workflow activity implementation or support tool implementation typically contains the following parts.

```
          ┌──     #include <MQ Workflow Api-prerequisites (C++)>
          │       #include <MQ Workflow API>
          │       int main()
          │       {
Setup     │         Declare objects
          │               :
          │         Connect
          │         InContainer()
          ┌──        OutContainer() if no support tool
          ┌──
          │
          │
          │
Actions   │         read values
          │         set values if no support tool
          │
          │
          ┌──
          ┌──        SetOutContainer() if no support tool
          │         Deallocate objects
Cleanup   │         Disconnect
          │         return rc;
          ┌──     }
```

To set up your program, you typically declare the program variables or objects you are going to use and you include the MQSeries Workflow API header files or copy the copybooks you need for your actions. Include the respective files before the MQSeries Workflow API headers.

You should then initialize the MQSeries Workflow API by calling the Connect() function/subprogram so that resources held by the API are allocated correctly. Connect() - and Disconnect() - are to be called at the begin respectively end of each thread.

An activity implementation can then retrieve the activity's input and output containers from the MQSeries Workflow program execution server that started this program. A support tool can retrieve the activity's input container only.

Having access to the containers, you can read and set values according to your programming logic.

At the end of your program, the activity implementation returns the final output container to the MQSeries Workflow program execution server. Any resources held by your program are deallocated. The return value of your program tells the program execution agent about the overall outcome of your program.

The output container as well as the return code of your program are passed back to the MQSeries Workflow server which requested the execution of the activity implementation. The return code (_RC) can be used in exit or transition conditions in order to guide MQSeries Workflow navigation. [1]

1. For compilers which do not support an exit code of an application, it is possible to set the _RC data member of the output container.

As a last step, you disconnect from the MQSeries Workflow API so that resources held by the API are deallocated correctly.

Your activity implementation or support tool can as well behave like a client application (see "Coding an MQSeries Workflow client application" on page 10) and request services from an MQSeries Workflow server, normally the server from where its execution had been triggered. The Passthrough() function/subprogram is then used instead of the Logon() function/subprogram in order to logon to the server which caused the program execution with the user identification and authority known to the server from the work item start request.

Compiling and linking

All programs developed for use with MQSeries Workflow must include header files provided by MQSeries Workflow and link with the corresponding library files.

Note that bool.h and vector.h are part of the Standard Template Library delivered with MQSeries Workflow and copyrighted by the Hewlett-Packard Company.

The MQSeries Workflow features you use determine which header files to include and the compilers you use which library files to link with. Depending on the feature used, the following header files must be included:

Feature	**C-API Header**
Runtime client	fmcjcrun.h
Runtime activity implementation:	
- container access only	fmcjccon.h
- container and server access	fmcjcrun.h
Runtime support tool	
- container access only	fmcjccon.h
- container and server access	fmcjcrun.h

For the corresponding Cobol-API copybooks refer to "Chapter 13. Cobol specific considerations" on page 81

The following JCLs are provided as samples for the development and execution of IBM MQSeries Workflow for OS/390 applications. They are located in the SFMCCNTL library delivered with MQSeries Workflow.

Table 1. Provided JCLs

Job	**Sample**
Native OS/390 C full API Compile Job	FMCHJ1CF
Native OS/390 C API Run Job	FMCHJ1CR
Native OS/390 Cobol full API Compile Job	FMCHJ1BF
Native OS/390 Cobol API Run Job	FMCHJ1BR
CICS C full API Compile Job	FMCHJ2CF
CICS C Container API Compile Job	FMCHJ2CC
CICS Cobol full API Compile Job	FMCHJ2BF
CICS Cobol Container API Compile Job	FMCHJ2BC
IMS C Container API Compile Job	FMCHJ3CC
IMS Cobol Container API Compile Job	FMCHJ3BC

For more information about CICS/IMS specifics like stubs or precompiler refer to the documentation of this components.

The compilers given as prerequisites or newer versions can be used to compile and link your applications accessing the MQSeries Workflow APIs. Your compile and link options must ensure that the MQSeries Workflow APIs are called with the calling convention that is defined in the FMC_APIENTRY macro (see file fmcjcglo.h). FMC_APIENTRY has been defined to the standard C calling convention and should automatically be applied when you use the header files provided by MQSeries Workflow.

Access can be gained to C language functions using calls from all languages that support C calls.

Chapter 4. Handling errors

All action, activity implementation, or program execution management functions/subprograms show whether or not the call has been successfully executed by returning a so-called *return code* as their return value. The return code is one of a set of predefined return codes (see "List of return codes" on page 17). The exact return codes for each of those functions/subprograms are listed with the description of each call. You should design your programs to handle all return codes that can arise.

Additional to the return code, a so-called *result object* can be accessed which describes the result of the call in more detail.

Basic and accessor functions/subprograms do not return any value or return the value queried as their return value. Since they are querying transient objects and are able to return default values, an error does normally not occur. It can, however, happen during application development that a wrong handle or a buffer too small to hold a character value is specified. To look for such erroneous situations, the *result object* can be queried (besides checking the trace).

The result object

In general, a result object states the result of the last MQSeries Workflow API request (in the considered program). It especially allows for analyzing an erroneous situation in more detail and contains the following information:

- The return code.
- The origin of the result, that is, the file that caused the result to be written, and the line and function where the error or the completion of the request occurred.
- Parameters (up to five) which describe the objects involved.

The result can be retrieved as a formatted message text with all parameters added to the text. The current locale is considered when building that message text so that the message is provided in your selected language.

All results of function/subprogram calls are written into the result object associated with the thread the request executes in. It is sufficient to access the result object just once per-thread using the FmcjResultObjectOfCurrentThread function. As threads are not supported in IBM MQSeries Workflow for OS/390 the ″OfCurrentThread″ is mentioned here for compatibility reasons with versions supporting threads. The result object is automatically updated with each request.

A result object is automatically allocated by MQSeries Workflow when the first MQSeries Workflow API call is issued in that program. It can be accessed at any time and as often as needed.

For example, in the C language, you can access and use a result object in the following way:

```
#include <stdio.h>
#include <fmcjcrun.h>
int main()
{
  FmcjResultHandle  result       = 0;
  FmcjStringVectorHandle  parms  = 0;
  char              buffer[2000] = "";

  result= FmcjResultObjectOfCurrentThread();
  printf( "Accessed result object of current thread\n" );

  printf( "Return code: %i\n", FmcjResultRc(result) );
  printf( "Text       : %s",   FmcjResultMessageText(result,buffer,2000) );
  printf( "Origin     : %s\n", FmcjResultOrigin(result,buffer,2000) );
  parms= FmcjResultParameters(result);
  while ( 0 != FmcjStringVectorNextResultParmElement( parms, buffer, 2000 ) )
    printf( "Parameter  : %s\n", buffer );

  return 0;
}
```

Note: The NextResultParmElement() function is used on the string vector so that the result object is not changed while reading the parameters.

For example, in the Cobol language, you can access and use a result object the following way:

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. "RESOBJ".

       DATA DIVISION.

         WORKING-STORAGE SECTION.

         COPY fmcvars.

         01 buffer PIC X(2000) VALUE SPACES.

       PROCEDURE DIVISION.

            PERFORM FmcjResultObjOfCurrentThread.
            DISPLAY "Accessed result object of current thread".

            SET hdlResult TO FmcjResultHandleReturnValue.
            PERFORM FmcjResultRc.
            DISPLAY "Return code: " intReturnValue.
            MOVE 2000 TO bufferlength.
            CALL "SETADDR" USING buffer messageBuffer.
            PERFORM FmcjResultMessageText.
            DISPLAY "Text       : " buffer.
            CALL "SETADDR" USING buffer originBuffer.
            PERFORM FmcjResultOrigin.
            DISPLAY "Origin     : " buffer.
            PERFORM FmcjResultParms.
            SET hdlVector TO FmcjStrVHandleReturnValue.

            CALL "SETADDR" USING buffer elementBuffer.
            PERFORM FmcjStrVNextResultParmElement.

            PERFORM UNTIL pointerReturnValue = NULL
               DISPLAY "Parameter  : " buffer
               PERFORM FmcjStrVNextResultParmElement
            END-PERFORM.

            STOP RUN.

            COPY fmcperf.
       IDENTIFICATION DIVISION.
       PROGRAM-ID. "RESOBJ".
```

```
DATA DIVISION.

  WORKING-STORAGE SECTION.

  COPY fmcvars.

  01 buffer PIC X(2000) VALUE SPACES.

PROCEDURE DIVISION.


     CALL "FmcjResultObjectOfCurrentThread"
        RETURNING FmcjResultHandleReturnValue.
     DISPLAY "Accessed result object of current thread".

     SET hdlResult TO FmcjResultHandleReturnValue.
     CALL "FmcjResultRc"
        USING BY VALUE hdlResult
        RETURNING intReturnValue.
     DISPLAY "Return code: " intReturnValue.
     MOVE 2000 TO bufferlength.
     CALL "SETADDR" USING buffer messageBuffer.
     CALL "FmcjResultMessageText"
        USING BY VALUE hdlResult
                       messageBuffer
                       bufferLength
        RETURNING pointerReturnValue.
     DISPLAY "Text       : " buffer.
     CALL "SETADDR" USING buffer originBuffer.
     CALL "FmcjResultOrigin"
         USING BY VALUE hdlResult
                        originBuffer
                        bufferLength
         RETURNING pointerReturnValue.
     DISPLAY "Origin     : " buffer.
     CALL "FmcjResultParameters"
         USING BY VALUE hdlResult
         RETURNING FmcjStrVHandleReturnValue.
     SET hdlVector TO FmcjStrVHandleReturnValue.

     CALL "SETADDR" USING buffer elementBuffer.
     CALL "FmcjStringVectorNextResultParmElement"
         USING BY VALUE hdlVector
                        elementBuffer
                        bufferLength
         RETURNING pointerReturnValue.

     PERFORM UNTIL pointerReturnValue = NULL
        DISPLAY "Parameter  : " buffer
        CALL "FmcjStringVectorNextResultParmElement"
            USING BY VALUE hdlVector
                           elementBuffer
                           bufferLength
            RETURNING pointerReturnValue
     END-PERFORM.

     STOP RUN.
```

List of return codes

The following list shows the numeric values of the return codes that are issued by the MQSeries Workflow APIs; it is strongly advised to use the symbolic names instead of the integer values. For Cobol the return codes have a maximum length of 30 characters. Additional words in the return codes are separated by underscores and not by hyphens (like it is common for C). In order to avoid misunderstandings the C version of the return codes is used in this book especially in the chapter about the APIs ("Part 7. Application programming interfaces" on page 213).

Table 2. List of return codes

Numeric value	Symbolic value (C)	Symbolic value (Cobol)
0	FMC_OK	FMC-OK
1	FMC_ERROR	FMC-ERROR
10	FMC_ERROR_USERID_UNKNOWN	FMC-ERROR-USERID-UNKNOWN
11	FMC_ERROR_ALREADY_LOGGED_ON	FMC-ERROR-ALR-LOGGED-ON
12	FMC_ERROR_PASSWORD	FMC-ERROR-PASSWORD
13	FMC_ERROR_COMMUNICATION	FMC-ERROR-COMMUNICATION
14	FMC_ERROR_TIMEOUT	FMC-ERROR-TIMEOUT
100	FMC_ERROR_INTERNAL	FMC-ERROR-INTERNAL
101	FMC_ERROR_SERVER	FMC-ERROR-SERVER
102	FMC_ERROR_UNKNOWN	FMC-ERROR-UNKNOWN
103	FMC_ERROR_MESSAGE_FORMAT	FMC-ERROR-MESSAGE-FORMAT
104	FMC_ERROR_MESSAGE_DATA	FMC-ERROR-MESSAGE-DATA
105	FMC_ERROR_RESOURCE	FMC-ERROR-RESOURCE
106	FMC_ERROR_NOT_LOGGED_ON	FMC-ERROR-NOT-LOGGED-ON
107	FMC_ERROR_NEW_OWNER_NOT_FOUND	FMC-ERROR-NEW-OWNER-NOT-FOUND
108	FMC_ERROR_NO_OLD_OWNER	FMC-ERROR-NO-OLD-OWNER
109	FMC_ERROR_OLD_OWNER_ABSENT	FMC-ERROR-OLD-OWNER-ABSENT
110	FMC_ERROR_NEW_OWNER_ABSENT	FMC-ERROR-NEW-OWNER-ABSENT
111	FMC_ERROR_ALREADY_STARTED	FMC-ERROR-ALR-STRTD
112	FMC_ERROR_MEMBER_NOT_FOUND	FMC-ERROR-MEMBER-NOT-FOUND
113	FMC_ERROR_MEMBER_NOT_SET	FMC-ERROR-MEMBER-NOT-SET
114	FMC_ERROR_WRONG_TYPE	FMC-ERROR-WRONG-TYPE
115	FMC_ERROR_MEMBER_CANNOT_BE_SET	FMC-ERROR-MEMBER-CANNOT-BE-SET
116	FMC_ERROR_MEMBER_INVALID	FMC-ERROR-MEMBER-INVAL
117	FMC_ERROR_FORMAT	FMC-ERROR-FORMAT
118	FMC_ERROR_DOES_NOT_EXIST	FMC-ERROR-DOES-NOT-EXIST
119	FMC_ERROR_NOT_AUTHORIZED	FMC-ERROR-NOT-AUTH
120	FMC_ERROR_WRONG_STATE	FMC-ERROR-WRONG-STATE
121	FMC_ERROR_NOT_UNIQUE	FMC-ERROR-NOT-UNIQUE
122	FMC_ERROR_EMPTY	FMC-ERROR-EMPTY
123	FMC_ERROR_NO_MANUAL_EXIT	FMC-ERROR-NO-MANUAL-EXIT
124	FMC_ERROR_PROFILE	FMC-ERROR-PROFILE
125	FMC_ERROR_INVALID_FILTER	FMC-ERROR-INVAL-FILTER
126	FMC_ERROR_PROGRAM_EXECUTION	FMC-ERROR-PROGRAM-EXECUTION
127	FMC_ERROR_PROTOCOL	FMC-ERROR-PROTOCOL
128	FMC_ERROR_TOOL_FUNCTION	FMC-ERROR-TOOL-FUNCTION
129	FMC_ERROR_INVALID_TOOL	FMC-ERROR-INVAL-TOOL
130	FMC_ERROR_INVALID_HANDLE	FMC-ERROR-INVAL-HANDLE
131	FMC_ERROR_NOT_EMPTY	FMC-ERROR-NOT-EMPTY
132	FMC_ERROR_INVALID_USER	FMC-ERROR-INVAL-USER
133	FMC_ERROR_OWNER_ALREADY_ASSIGNED	FMC-ERROR-OWNER-ALR-ASSIGNED
134	FMC_ERROR_INVALID_NAME	FMC-ERROR-INVAL-NAME
135	FMC_ERROR_INVALID_PROGRAMID	FMC-ERROR-INVAL-PROGRAMID
136	FMC_ERROR_SIZE_EXCEEDED	FMC-ERROR-SIZE-EXCEEDED
406	FMC_ERROR_WRONG_ACT_IMPL_KIND	FMC-ERROR-WRONG-ACT-IMPL-KIND
500	FMC_ERROR_NON_LOCAL_USER	FMC-ERROR-NON-LOCAL-USER
501	FMC_ERROR_WRONG_KIND	FMC-ERROR-WRONG-KIND
502	FMC_ERROR_INVALID_ACTIVITY	FMC-ERROR-INVAL-ACT
503	FMC_ERROR_CHECKOUT_NOT_POSSIBLE	FMC-ERROR-CHECKOUT-NOT-POSSIBLE

Table 2. List of return codes (continued)

Numeric value	Symbolic value (C)	Symbolic value (Cobol)
800	FMC_ERROR_BUFFER	FMC-ERROR-BUFFER
801	FMC_ERROR_INVALID_SESSION	FMC-ERROR-INVAL-SESSION
802	FMC_ERROR_INVALID_TIME	FMC-ERROR-INVAL-TIME
804	FMC_ERROR_NO_MORE_DATA	FMC-ERROR-NO-MORE-DATA
805	FMC_ERROR_INVALID_OID	FMC-ERROR-INVAL-OID
807	FMC_ERROR_INVALID_THRESHOLD	FMC-ERROR-INVAL-THRESHOLD
808	FMC_ERROR_INVALID_SORT	FMC-ERROR-INVAL-SORT
809	FMC_ERROR_OBJECT_IN_USE	FMC-ERROR-OBJ-IN-USE
810	FMC_ERROR_INVALID_DESCRIPTION	FMC-ERROR-INVAL-DESCRIPTION
811	FMC_ERROR_INVALID_INVOCATION_TYPE	FMC-ERROR-INVAL-INV-TYPE
812	FMC_ERROR_OWNER_NOT_FOUND	FMC-ERROR-OWNER-NOT-FOUND
813	FMC_ERROR_INVALID_LIST_TYPE	FMC-ERROR-INVAL-LIST-TYPE
814	FMC_ERROR_INVALID_RESULT_HANDLE	FMC-ERROR-INVAL-RESULT-HANDLE
815	FMC_ERROR_MESSAGE_CATALOG	FMC-ERROR-MESSAGE-CATALOG
816	FMC_ERROR_INVALID_SPECIFICATION	FMC-ERROR-INVAL-SPECIFICATION
817	FMC_ERROR_QRY_RESULT_TOO_LARGE	FMC-ERROR-QRY-RESULT-TOO-LARGE
818	FMC_ERROR_NO_VERSION_2_FILTER	FMC-ERROR-NO-VERSION-2-FILTER
900	FMC_ERROR_NO_SYS_ADMIN	FMC-ERROR-NO-SYS-ADMIN
901	FMC_ERROR_INVALID_SESSION_MODE	FMC-ERROR-INVAL-SESSION-MODE
902	FMC_ERROR_PROGRAM_UNDEFINED	FMC-ERROR-PROGRAM-UNDEFINED
903	FMC_ERROR_PEA_NOT_RUNNING	FMC-ERROR-PEA-NOT-RUNNING
904	FMC_ERROR_PEA_NOT_LOCAL	FMC-ERROR-PEA-NOT-LOCAL
905	FMC_ERROR_INVALID_ABSENCE_SPEC	FMC-ERROR-INVAL-ABSENCE-SPEC
1000	FMC_ERROR_NOT_SUPPORTED	FMC-ERROR-NOT-SUPPORTED
1012	FMC_ERROR_PROGRAM_NOT_DEFINED	FMC-ERROR-PROGRAM-NOT-DEFINED
1014	FMC_ERROR_PEA_NOT_REACHABLE	FMC-ERROR-PEA-NOT-REACHABLE
1015	FMC_ERROR_INVALID_PEA_FROM_CTNR	FMC-ERROR-INVALID-PEA-FRM-CTNR
1016	FMC_ERROR_INVALID_PEA_FROM_MODEL	FMC-ERROR-INVAL-PEA-FRM-MODEL
1017	FMC_ERROR_INVALID_SYSTEM_FROM_CTNR	FMC-ERROR-INVAL-SYSTEM-FRM-CTNR
1018	FMC_ERROR_INVALID_SYSTEM_FROM_MODEL	FMC-ERROR-INVAL-SYSTEM-FRM-MODEL
1019	FMC_ERROR_SUB_PROC_TERMINATED_BY_ERROR	FMC-ERROR-SUB-PROC-TERMINATED-BY-ERROR
1020	FMC_ERROR_NO_PEA_FOUND_FOR_AUTO_START	FMC-ERROR-NO-PEA-FND-FR-AUT-ST
1021	FMC_ERROR_NO_CTNR_ACCESS	FMC-ERROR-NO-CTNR-ACCESS

Chapter 5. Memory management

Workflow process models, their instances, and resulting work items are all objects persistently stored in an MQSeries Workflow database. This means that they exist independently from an application program.

When persistent objects are queried by an application program, they are represented by *transient objects* which carry the states of the persistent objects at the time of the query. When multiple queries are issued, there can be multiple transient objects representing the same persistent object, even representing different states of that object.

The lifetime of transient objects and their memory is *fully managed* by you, because you know best when those objects are no longer needed, that is, when objects are to be deallocated. Transient objects are, however, no longer available when your application program ends.

Some transient objects are *explicitly allocated* by you. These are supporting objects, which do not reflect persistent ones. Examples are the FmcjStringVector when you specify a set of persons to stand in for or the FmcjExecutionService object, which allows services to be requested from an execution server.

Transient objects, which do reflect persistent objects are *implicitly allocated* by you when you create or when you retrieve persistent objects, for example, by querying.

Although the lifetime of transient objects is fully managed by you, their actual internal object structure is encapsulated by the MQSeries Workflow API. The MQSeries Workflow API provides a handle to you so that you can issue requests against the object.

The MQSeries Workflow API follows the *programming by contract* concept. This means that any handle passed to it which is not 0 (NULL) is assumed to be a valid handle which can be used to access an object.

Chapter 6. Communication protocols and data access models

When you request actions from an MQSeries Workflow server or when you want to observe the result of actions, you can:

- Use a synchronous protocol to view changes of the object which you used to call the action.
- Use a synchronous protocol to pull for data created or changed.
- Receive unsolicited information on created or changed objects pushed by the server.

For example, when you ask a process instance object to be started[2]:

- As an immediate result, the state of the process instance is updated.
- You can query work items in order to view (pull for) new objects created.
- You can automatically receive new work items sent (pushed) to you.

Applying a synchronous protocol means that you issue a request to an MQSeries Workflow server and then wait until you receive a response. All action functions/subprograms operate this way; your application is blocked until the response arrives or until your timeout set on the execution service object exceeds.

Receiving unsolicited information pushed by an MQSeries Workflow server means that you set up communication in a way that you are automatically informed about new or changed objects.

The push data access model

In order to obtain information pushed by an MQSeries Workflow server:

1. The server must be asked for sending data. This means that:
   - The settings of the considered process instance must specify *REFRESH_POLICY PUSH*. This setting is inherited from the domain level, through the system group to the system and down to the process template. Each specification can be overwritten on a lower level.
   - The users must be logged on with a *Present* or *PresentHere* session mode, that is, they are enabled to receive information.
2. The application must use functions/subprograms in order to receive data pushed.

Provided that these prerequisites are fulfilled, the MQSeries Workflow execution server pushes changes on work items or notifications to the owner of the item:

1. On creation of the item.
2. On deletion of the item.
3. Whenever a primary property of the item changes.

The caller of the action will, however, not receive such information because, as a result of the action, the transient object has already been updated with relevant data.

2. C language: FmcjProcessInstanceStart(instance,...); Cobol language: PERFORM FmcjPIStart

Changes on disabled work items are not pushed. Only the deletion of such work items is pushed.

Examples:

When a process instance is suspended and when its refresh policy is push, the MQSeries Workflow execution server sends informations to all owners of non-disabled items which are currently logged on as present.

When the description of a process instance is changed and when the refresh policy is push, the MQSeries Workflow execution server sends informations to all owners of process instance notifications which are currently logged on as present.

When a work item is transferred to user N by the owner of the work item and when the refresh policy of the associated process instance is push, the MQSeries Workflow execution server sends an information to user N when he/she is currently logged on as present. The owner of the work item as the requester of the action does not get any additional information.

Note: Filtering and sorting is left to the application. No indication about affected worklists is pushed to the client.

Receiving information

The execution service object provides for a means to receive information (execution data) pushed by an MQSeries Workflow execution server at any time wanted. The Receive() call blocks the calling application until some information is received or until the specified timeout value has been reached. This means that an application typically starts a separate thread for receiving data in order to prevent that the whole application is blocked.

A timeout value of -1 specifies an indefinite wait time interval. Note that in this case you must ensure that you stop receiving data before your application ends. There is a TerminateReceive() function/subprogram which can be used to send a terminate indication to the receiving part of the application in order to inform that receiving data may end.

Notes:
1. A Receive() call survives a Logoff() call. The execution server stops, however, pushing information when logoff has been executed.
2. If information is not received and therefore stays in the client input queue, the MQSeries expiration mechanism applies in order to get rid of such "dead" messages. The expiration time of client messages can be configured at system setup.

When receiving data, a correlation identification can be specified to indicate which information is to be read. This ID **must** currently be set to FMCJ_NO_CORRELID for **each** Receive() call; note that it is changed as the result of a successful receive.

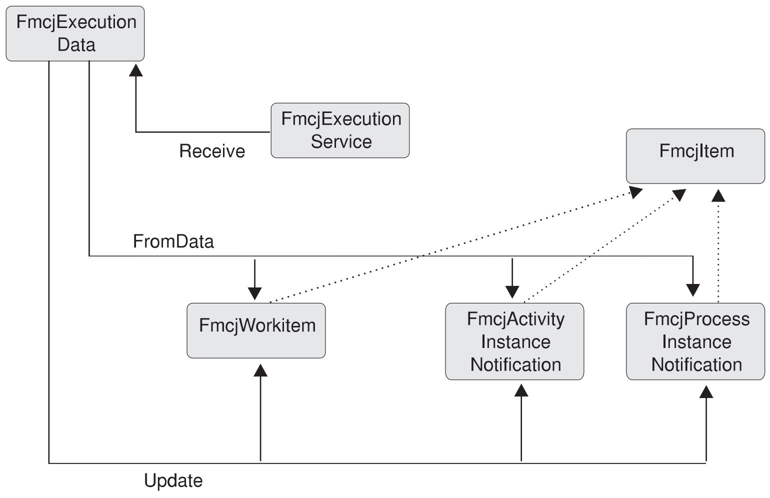


Figure 1. Handling data sent by an MQSeries Workflow server. Legend: --► Inheritance (C++); —► provides for access

Once execution data has been received, its type can be determined and the appropriate action can be called. For example, when a work item creation is indicated, a conversion from the execution data to a work item can be requested. When a work item change is indicated, the persistent object ID of the work item can be requested so that the appropriate work item can be updated.

Chapter 7. Establishing an MQSeries Workflow session

In order to communicate with an MQSeries Workflow server, a session must have been established between the user and that server. The server is either identified explicitly (system at system group) or taken from the user's profile. If the information is not found in the user's profile, the workstation profile is read.

The session is established by logging on. From then on services can be requested from the server; the service object which represents the session between the user logging on and the server, is set up accordingly.

Logon requires that the administration server is up and running on the selected system because the administration server manages sessions and checks the authentication of the user. It additionally cares for any severe errors to be written to the error log.

Any objects which are retrieved or created belong to the session where they have been queried or created. They carry the session identification so that further actions on those objects are executed in the same session with the authorization of the logged-on user.

A single application program or multiple application programs can allocate multiple service objects and log on with different users or the same user in parallel. Sessions are kept separate by the service objects. A single service object thus represents a single session. A second request to log on via a service object will be rejected if it comes from a different user. Otherwise, it is accepted but not repeated; the logon request has already been executed successfully.

A session can run in ″default″ mode or in ″present″ mode. When you are operating in a present session mode, activity instances which are started automatically can be scheduled on your behalf and you can receive information pushed by an MQSeries Workflow server. There can only be a single present session.

The service object provides for a timeout value to be set. This is the time the application waits for the answer from a server. The application is thus blocked during this time at a maximum. The timeout is specified in milliseconds. A value of -1 denotes an indefinite timeout value. The timeout value can be changed at any time.

Chapter 8. Querying data

There are essentially three means of querying data from an MQSeries Workflow server:

- A query via a service object, which returns all objects authorized for. The number of objects returned to the client can be restricted by a filter and a threshold.
- A query using a persistent list definition, which returns all objects qualifying through the list definition.
- A specific request, like the request for user settings or a refresh request for a specific object.

Persistent lists

A persistent list represents a set of objects of the same type. Moreover, all objects which are accessible through the list have the same characteristics. A list can be for public usage, that is, it is visible by all users, or for private usage, that is, it has an owner and is only visible by that owner.

The characteristics of the objects contained in the list are given by so-called *filter criteria*. The filter criteria specified and the authorization of the user issuing the query determine the contents of the list. This means that the contents itself is not stored persistently but determined when a query request is issued.

The number of objects transferred from the server to the client as the result of the query can be restricted by specifying a *threshold*. The threshold is used after *sort criteria* have been applied.

A list can be a process template list, a process instance list, or a worklist.

Using filters, sort criteria, and thresholds

A filter is a character string specifying criteria which must follow the rules stated by the filter syntax diagrams. Refer to the appropriate functions/subprograms for the exact syntax. Some sample criteria are shown here:

```
"NAME = 'MyProcessInstance'"
"NAME LIKE 'My*Ins?ance'"
"LAST_MODIFICATION_TIME > '1998-2-19 11:38:0'"
"STATE IN (READY,RUNNING)"
```

A sort criterion is a character string specifying criteria which must follow the rules stated by the sort criteria syntax diagrams. Refer to the appropriate functions/subprograms for the exact syntax. Some sample criteria are shown here: Note that objects are sorted on the server, that is, the code page of the server

```
"NAME ASC"
"NAME ASC, LAST_MODIFICATION_TIME DESC"
```

determines the sort sequence.

A threshold specifies the maximum number of objects to be returned to the client. That threshold is applied after the objects have been sorted.

Handling collections

The result of a query for a set of objects is a *vector* of objects. The vector is provided by the caller and filled by the MQSeries Workflow API. The ownership of the vector elements, the objects, stays with the vector. They are automatically deleted when the vector is deleted. When an element is read, it becomes an object on its own and thus has to be deleted when no longer used.

Any objects returned are appended to the supplied vector. If you want to read the current objects only, you have to clear the vector before you call the query method. This means that you should set the vector handle to 0 via the FmcjXxxVectorDeallocate function in “FmcjXxxVectorDeallocate” on page 31.

In the C- and Cobol language, the result of the query is the vector handle initialized to the set of objects. Special vector accessor functions are provided to access the objects (see below). When a vector element is read it becomes an object of its own and thus has to be deleted when no longer used. Any operations on that object refer to this object only and do not have any impacts on the vector element from which the object was copied. For example, a Refresh() changes the object only but not its original copy whithin the vector. This means that a further iteration through the vector finds any elements that are unchanged.

Vector accessor functions

Vector accessor functions are described below. This is because all these functions are similar looking and have similar requirements, even for different objects. They are all handled locally by the API, that is, they do not communicate with the server. Neither a connection to a server nor specific authorizations are required to execute.

Return codes

The C language functions or the result object can return the following codes, the number in parentheses shows their integer value:

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is expected, but 0 is passed.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_NO_MORE_DATA(804)
The vector contains no or no more element.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

Vector accessor functions allow for the operations listed below; ’Xxx’ denotes some scope, for example, FmcjXxxVectorFirstElement can stand for FmcjProcessInstanceVectorFirstElement.

FmcjXxxVectorDeallocate

Allows the application to deallocate the storage reserved for the specified transient vector object. All elements contained are also deallocated.

The handle is set to 0 so that it cannot be used any longer.

C language signature

```
APIRET FMC_APIENTRY FmcjXxxVectorDeallocate(
       FmcjXxxVectorHandle * hdlVector)
```

Cobol language signature

```
         FmcjXxxVectorDeallocate.

            CALL    "FmcjXxxVectorDeallocate"
                           USING
                           BY REFERENCE
                              hdlVector
                           RETURNING
                              intReturnValue.
```

Parameters

hdlVector
Input/Output. The address of the handle to the vector to be deallocated.

FmcjXxxVectorFirstElement

Returns the first element of the vector. That element becomes an object on its own and has to be deallocated if no longer used. The vector is positioned to the next element.

If the vector is empty or if an error occurred, 0 (zero) is returned.

C language signature

```
FmcjXxxHandle FMC_APIENTRY FmcjXxxVectorFirstElement(
              FmcjXxxVectorHandle  hdlVector )
```

Cobol language signature

```
       FmcjXxxVectorFirstElement.

          CALL    "FmcjXxxVectorFirstElement"
                         USING
                         BY VALUE
                            hdlVector
                         RETURNING
                            FmcjXxxHandleReturnValue.
```

Parameters

hdlVector
Input. The handle of the vector to be queried.

Return type

FmcjXxxHandle
The handle of the first element of the vector or 0.

FmcjXxxVectorNextElement

Returns the vector element at the current vector position; the initial vector position is the first element. That element becomes an object on its own and has to be deallocated if no longer used. The vector is positioned to the next element.

If the vector is empty, if there are no more elements in the vector, or if an error occurred, 0 (zero) is returned.

C language signature

```
FmcjXxxHandle FMC_APIENTRY FmcjXxxVectorNextElement(
              FmcjXxxVectorHandle  hdlVector )
```

Cobol language signature

```
       FmcjXxxVectorNextElement.

          CALL    "FmcjXxxVectorNextElement"
                         USING
                         BY VALUE
                            hdlVector
                         RETURNING
                            FmcjXxxHandleReturnValue.
```

Parameters

hdlVector
Input. The handle of the vector to be queried.

Return type

FmcjXxxHandle
The handle of the vector element at the current position or 0.

FmcjXxxVectorSize

Returns the number of elements in the vector.

C language signature

```
unsigned long FMC_APIENTRY FmcjXxxVectorSize(
         FmcjXxxVectorHandle       hdlVector )
```

Cobol language signature

```
         FmcjXxxVectorSize.

            CALL    "FmcjXxxVectorSize"
                           USING
                           BY VALUE
                              hdlVector
                           RETURNING
                              ulongReturnValue.
```

Parameters

hdlVector
Input. The handle of the vector to be queried.

Return type

unsigned long
The number of elements in the vector.

Examples

In the following, some C language examples on how to read a vector are shown; note that you can start with a first element call as well as with a next element call.

```
#include <stdio.h>
#include <fmcjrun.h>
int main()
{
  APIRET                          rc;
  FmcjExecutionServiceHandle      service     = 0;
  FmcjProcessInstanceVectorHandle hdlVector   = 0;
  FmcjProcessInstanceHandle       hdlInstance  = 0;
  unsigned long                   i           = 0;
  unsigned long                   numElements  = 0;
  char                            tInfo[FMC_PROCESS_INSTANCE_NAME_LENGTH]="";

  FmcjGlobalConnect();

  FmcjExecutionServiceAllocate(&service);
  rc = FmcjExecutionServiceLogon( service,
                                  "ADMIN", "PASSWORD",
                                  Fmc_SM_Default, Fmc_SA_Reset
                                );
  if ( rc != FMC_OK )
    return rc;
  printf("Logged on\n");

  rc= FmcjExecutionServiceQueryProcessInstances(
                              service,
                              FmcjNoFilter,
                              FmcjNoSortCriteria,
                              FmcjNoThreshold,
                              &hdlVector  );
  if ( rc != FMC_OK )
    return rc;
  printf("Queried process instances\n");

  hdlInstance= FmcjProcessInstanceVectorFirstElement(hdlVector);
  numElements= FmcjProcessInstanceVectorSize(hdlVector);

  printf("Instances in the vector:\n");
  for( i=0; i< numElements; i++ )
  {
    printf("- name: %s\n",
           FmcjProcessInstanceName(hdlInstance,tInfo,
                                   FMC_PROCESS_INSTANCE_NAME_LENGTH));
    FmcjProcessInstanceDeallocate(&hdlInstance);
    hdlInstance= FmcjProcessInstanceVectorNextElement(hdlVector) ;
  }

  FmcjProcessInstanceVectorDeallocate(&hdlVector);

  FmcjExecutionServiceLogoff(service);
  printf("Logged off\n");
  FmcjExecutionServiceDeallocate(&service);

  FmcjGlobalDisconnect();
  return FMC_OK;
}
```

Using NextElement() call only

```
#include <stdio.h>
#include <fmcjrun.h>
int main()
{
  APIRET                          rc;
  FmcjExecutionServiceHandle      service      = 0;
  FmcjProcessInstanceVectorHandle hdlVector    = 0;
  FmcjProcessInstanceHandle       hdlInstance  = 0;
  char                            tInfo[FMC_PROCESS_INSTANCE_NAME_LENGTH]="";

  FmcjGlobalConnect();

  FmcjExecutionServiceAllocate(&service);
  rc = FmcjExecutionServiceLogon( service,
                                  "ADMIN", "PASSWORD",
                                  Fmc_SM_Default, Fmc_SA_Reset
                                );
  if ( rc != FMC_OK )
    return rc;
  printf("Logged on\n");

  rc= FmcjExecutionServiceQueryProcessInstances(
                              service,
                              FmcjNoFilter,
                              FmcjNoSortCriteria,
                              FmcjNoThreshold,
                              &hdlVector  );
  if ( rc != FMC_OK )
    return rc;
  printf("Queried process instances\n");

  printf("Instances in the vector:\n");
  while (0 != (hdlInstance= FmcjProcessInstanceVectorFirstElement(hdlVector)))
  {
    printf("- name:%s \n",
           FmcjProcessInstanceName(hdlInstance,tInfo,
                                   FMC_PROCESS_INSTANCE_NAME_LENGTH));
    FmcjProcessInstanceDeallocate(&hdlInstance) );
  }
   FmcjProcessInstanceVectorDeallocate(&hdlVector) );

  FmcjExecutionServiceLogoff(service);
  printf("Logged off\n");
  FmcjExecutionServiceDeallocate(&service);

  FmcjGlobalDisconnect();
  return FMC_OK;
}
```

In the following, some Cobol examples on how to read a vector are shown; note that you can start with a first element call as well as with a next element call.

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. "VECTOR".

       DATA DIVISION.

         WORKING-STORAGE SECTION.

         COPY fmcvars.
         COPY fmcconst.
         COPY fmcrcs.

         01 localUserID   PIC X(30) VALUE z"ADMIN".
         01 localPassword PIC X(30) VALUE z"PASSWORD".
         01 numElements   PIC 9(9) BINARY.
         01 i             PIC 9(9) BINARY.
         01 buffer        PIC X(64) VALUE SPACES.

         LINKAGE SECTION.

         01 retCode            PIC S9(9) BINARY.

       PROCEDURE DIVISION USING retCode.

            PERFORM FmcjGlobalConnect.
            PERFORM FmcjESAllocate.

            CALL "SETADDR" USING localUserId userId.
            CALL "SETADDR" USING localPassword passwordValue.
            MOVE Fmc-SM-Default TO sessionMode.
            MOVE Fmc-SA-Reset TO absenceIndicator.
            PERFORM FmcjESLogon.

            MOVE intReturnValue TO retCode
            IF retCode NOT = FMC-OK THEN GOBACK.
            DISPLAY "Logged on".

            CALL "SETADDR" USING FmcjNoFilter filter.
            CALL "SETADDR" USING FmcjNoSortCriteria sortCriteria.
            MOVE FmcjNoThreshold TO threshold.
            PERFORM FmcjESQueryProcInsts.

            SET hdlVector TO instances.
            MOVE intReturnValue TO retCode
            IF retCode NOT = FMC-OK THEN GOBACK.
            DISPLAY "Queried Process Instances".

            PERFORM FmcjPIVFirstElement.
            SET hdlInstance TO FmcjPIHandleReturnValue.
            PERFORM FmcjPIVSize.
```

```
     MOVE ulongReturnValue TO numElements.

     DISPLAY "Instances in the vector:".
     MOVE FMC-PROC-INST-NAME-LENGTH TO bufferLength.
     CALL "SETADDR" USING buffer instanceNameBuffer.
     PERFORM VARYING i FROM 0 BY 1 UNTIL i >= numElements
        PERFORM FmcjPIName
        DISPLAY "- name: " buffer
        PERFORM FmcjPIDeallocate
        PERFORM FmcjPIVNextElement
        SET hdlInstance TO FmcjPIHandleReturnValue
     END-PERFORM

     PERFORM FmcjPIVDeallocate.
     PERFORM FmcjESLogoff.
     DISPLAY "Logged off".
     PERFORM FmcjESDeallocate.
     PERFORM FmcjGlobalDisconnect.
     MOVE FMC-OK TO retCode.
     GOBACK.

     COPY fmcperf.
IDENTIFICATION DIVISION.
PROGRAM-ID. "VECTOR".

DATA DIVISION.

  WORKING-STORAGE SECTION.

  COPY fmcvars.
  COPY fmcconst.
  COPY fmcrcs.

  01 localUserID   PIC X(30) VALUE z"ADMIN".
  01 localPassword PIC X(30) VALUE z"PASSWORD".
  01 buffer        PIC X(64) VALUE SPACES.

  LINKAGE SECTION.

  01 retCode            PIC S9(9) BINARY.

PROCEDURE DIVISION USING retCode.

     PERFORM FmcjGlobalConnect.
     PERFORM FmcjESAllocate.

     CALL "SETADDR" USING localUserId userId.
     CALL "SETADDR" USING localPassword passwordValue.
     MOVE Fmc-SM-Default TO sessionMode.
     MOVE Fmc-SA-Reset TO absenceIndicator.
     PERFORM FmcjESLogon.

     MOVE intReturnValue TO retCode
     IF retCode NOT = FMC-OK THEN GOBACK.
     DISPLAY "Logged on".

     CALL "SETADDR" USING FmcjNoFilter filter.
     CALL "SETADDR" USING FmcjNoSortCriteria sortCriteria.
     MOVE FmcjNoThreshold TO threshold.
     PERFORM FmcjESQueryProcInsts.

     SET hdlVector TO instances.
     MOVE intReturnValue TO retCode
     IF retCode NOT = FMC-OK THEN GOBACK.
     DISPLAY "Queried Process Instances".

     DISPLAY "Instances in the vector:".
```

```
MOVE FMC-PROC-INST-NAME-LENGTH TO bufferLength.
CALL "SETADDR" USING buffer instanceNameBuffer.

PERFORM FmcjPIVNextElement.

PERFORM UNTIL FmcjPIHandleReturnValue = NULL
   SET hdlInstance TO FmcjPIHandleReturnValue
   PERFORM FmcjPIName
   DISPLAY "- name: " buffer
   PERFORM FmcjPIDeallocate
   PERFORM FmcjPIVNextElement
END-PERFORM

PERFORM FmcjPIVDeallocate.
PERFORM FmcjESLogoff.
DISPLAY "Logged off".
PERFORM FmcjESDeallocate.
PERFORM FmcjGlobalDisconnect.
MOVE FMC-OK TO retCode.
GOBACK.

COPY fmcperf.
```

Chapter 9. Handling Containers

A container represents input or output data of a process template, process instance, work item, activity implementation, or support tool at *Runtime*. Each container is defined by a *data structure* which declares the container to be of the type of that data structure.

Data structure/container type

A data structure is uniquely identified by its name and contains an ordered list of *data members*.

The data structures and their usage as input containers or output containers are defined during modeling. A special data structure called DEFAULT_DATA_STRUCTURE is provided by MQSeries Workflow and contains no user-defined data members when installed. The DEFAULT_DATA_STRUCTURE cannot be deleted, however, it can be extended during modeling.

Data member/container element

A data member of a data structure has a name and a *data type*. Data types are either basic and then STRING, LONG, BINARY, or FLOAT, or another data structure. Using a data structure as the data type of a data member (nesting) allows for recursive definitions of data members.

A data member can represent a one-dimensional array. If a data member represents an array, the number of elements in that array is shown in parenthesis ().

A data structure can have up to 512 user-defined data members. A data member that represents an array of data members counts with as many data members as it has elements.

Data members are specified using their fully qualified name within the container. The fully qualified name of a data member is a name in dot notation where the hierarchy of nested data members is presented from left to right, and their names are separated by a dot.

If a data member actually specifies an array of data members, the index number of a specific data member is specified in brackets ([]).

When a data structure denotes the type of a container, then its data members (first level of any hierarchy) are also called *container elements*. They define the *structural members* of the container. When the data type of a container element (n-th level of any hierarchy) is a data structure (nesting), then that container element again has container elements or structural members.

Container elements of a basic data type are also called the *leaves* of the container. These are the members which can hold a value, that is, which can be asked for a value and which can be set to a new value.

For example, assume that the data structure PERSON describes an input container or output container and that PERSON has been defined as:
PERSON has two structural data members named Name and Addr. Name is of basic

```
Name        STRING
Addr        ADDRESS
  Street    STRING
  POBOX     LONG(2)
```

data type STRING and Addr is of data type ADDRESS. That is the data structure ADDRESS is nested within the data structure PERSON.

The input or output container described by PERSON then has two container elements or structural members named Name and Addr, where Addr defines a structure by itself. The container elements or structural members of the container element Addr are Street and POBOX.

The leaves of the container, that is, the container elements which can carry a value, and their fully qualified names within the container are:
Note that since the size of the POBOX array is 2, the valid index numbers are 0 and

```
Name
Addr.Street
Addr.POBOX[0]
Addr.POBOX[1]
```

1. This is because all array indexes start with 0 (zero).

Also note that the fully qualified names are not prefixed with the name of the data structure PERSON. That data structure denotes the type of the container. There is only one exception to the rule, when the container itself is specified to be an array, for example, an array of PERSONs. Then, to set the name of a specific person, the fully qualified name is specified as

```
PERSON[i]·Name
```

Predefined data members

All containers automatically specify data members predefined by MQSeries Workflow. They can hold values associated with the operational characteristics of an activity or process. Predefined data members are data members that need not be defined by the modeler but are automatically available. They can be accessed by the container API. Their names start with the reserved character "_".

Predefined data member values can be:
- Used to evaluate activity exit criteria.
- Accessed by activity implementations or support tools.
- Dynamically set to change the operational characteristics of subsequent activities.

Predefined data members provide for the flexibility of modelers. The decision on operational characteristics of a process or activity is taken at Runtime. They also provide activity implementations and support tools as a means to access the operational characteristics through the use of API functions/subprograms.

There are the following sets of predefined data members:
- Fixed data members
- Process information data members
- Activity information data members

Fixed data members provide information about the current activity instance. They *cannot or should not be set* using an API function/subprogram.

Process information and activity information data members are associated with the operational characteristics of a process or activity. They operate the same way as any user-defined data members. This means that the values for specific operational characteristics of a process instance or activity instance can be accessed or changed just like the values for any other user-defined data member.

The following provides the fully qualified name and a brief description of each of the predefined data members.

There are no arrays of any predefined data member.

Fixed data members

Fixed data members _ACTIVITY, _PROCESS, and _PROCESS_MODEL *cannot be set* using API functions/subprograms. Their values *can be read* using API container functions/subprograms. Fixed data member _RC *cannot be read* and should only be set when your compiler does not support a program exit code.

`_ACTIVITY`
: This data member contains the name of the considered activity instance. The value of this data member is automatically set when the activity instance respectively an associated work item is started.

  Data type: STRING

`_PROCESS`
: This data member contains the name of the associated process instance. The value of this data member is automatically set when the activity instance respectively an associated work item is started.

  Data type: STRING

`_PROCESS_MODEL`
: This data member contains the name of the associated process model. The value of this data member is automatically set when the activity instance respectively an associated work item is started.

  Data type: STRING

`_RC`
: This data member contains the return code of the activity implementation. Typically it is used to evaluate exit and transition conditions. It cannot be read and is set automatically to the exit code of the activity implementation when that program ends.

  In cases where your compiler does not support an exit code, you can use the Container API to set its value.

  Data type: LONG

Process information data members

Process information data members serve to dynamically specify properties of a process instance. In general, the process modeler can choose where values for process instance properties are to be obtained from.

- Values can be inherited from a top-level process instance.

- Values can be obtained from the process information data members in the input container. They are then either set as default values or provided in the input container when the process instance is started.

If specified via the *DATA_FROM_INPUT_CONTAINER* indicator, the values of the process information data members are read by MQSeries Workflow when the process instance is started. If a value for a process information data member is not set, then a default value is used (see the detailed descriptions below).

`_PROCESS_INFO.Role`

A role that people assigned to an activity instance of the process instance must fulfill.

Any role set becomes an additional criterion to roles set for the activity instance. Only people who are members of all the specified roles are eligible.

If no role is set and no roles are specified for the activity instance, then no role criteria are applied.

Data type: STRING

`_PROCESS_INFO.Organization`

The organization to which people must belong to receive work items of the process instance. This setting is only regarded if no organization is specified for the activity instance.

If no organization is set and no organization is specified for the activity instance, the default is the organization of the person who starts the process instance.

Data type: STRING

`_PROCESS_INFO.ProcessAdministrator`

The user ID of the person notified if:
- The process instance is expired.
- No person meets the criteria to perform an activity instance.
- No valid person has been specified for notification.
- The person notified that an activity instance is overdue has exceeded the time allowed for an action, that is, the second notification is sent.

If not set, the default process administrator is the person who starts the process instance.

Data type: STRING

`_PROCESS_INFO.Duration`

Specifies how long the process instance is allowed to take. The value is expressed in seconds.

If not set, the default is "Endless".

Data type: LONG

Activity information data members

Activity information data members serve to dynamically specify properties of an activity instance. In general, the process modeler can choose where values for activity instance properties are to be obtained from.

- Values can be obtained from the activity information data members in the input container. They are then either set as default values or provided in the input container when an activity instance or associated work item is started.

If specified, the values of the activity information data members are read by MQSeries Workflow when the activity instance is scheduled. If a value is not set, then a default value is used (see the detailed descriptions below).

Following indicators specify that activity information data members are to be read:
- DONE_BY STAFF DEFINED_IN INPUT_CONTAINER
- NOTIFICATION DEFINED_IN INPUT_CONTAINER
- PRIORITY DEFINED_IN INPUT_CONTAINER

`_ACTIVITY_INFO.Priority`

The numeric value assigned as the priority of an activity instance. MQSeries Workflow does not deduce any meaning from this value; it is just used for client purposes. Any integer value between 0 and 9 can be specified. If the value specified is invalid or the data member is not set, a default of 0 (zero) is used.

Data type: LONG

`_ACTIVITY_INFO.MembersOfRoles`

The role or roles a person must fulfill to receive a work item for the activity instance. Multiple roles may be specified and are then to be separated by a semicolon (;).

Any role or roles set for this data member become an additional criterion to the role set for the process instance. Only people who are members of all the specified roles are eligible.

If not set, the role specified for the process instance is used. If no role is set for the process instance and no roles are specified for the activity instance, then no role criteria are applied.

Note: This specification is ignored if any specific people are set using the `_ACTIVITY_INFO.People` data member.

Data type: STRING

`_ACTIVITY_INFO.CoordinatorOfRole`

The role or roles a person must coordinate to receive a work item for the activity instance. Multiple roles to coordinate may be specified and are then to be separated by a semicolon (;).

To receive a work item, the eligible person must be assigned as coordinator of all the specified roles in addition to being a member of all roles specified for the process instance and for the activity instance.

If not set, the roles specified by the process instance and the activity instance are solely used. If no roles to be member of nor roles to coordinate have been specified, no role criteria are applied.

Note: This criterion is ignored if any specific people are set using the `_ACTIVITY_INFO.People` data member.

Data type: STRING

`_ACTIVITY_INFO.Organization`

The organization to which people must belong to receive work items of the activity instance.

If an organization is set using this data member, any organization set for the process instance is ignored.

If not set, the organization specified by the process instance is used. If no organization is set and no organization is specified for the process instance properties, the default is the organization of the person who starts the process instance.

Note: This criterion is ignored if any specific people are set using the `_ACTIVITY_INFO.People` data member.

Data type: STRING

`_ACTIVITY_INFO.OrganizationType`

This data member is used to indicate if a work item for the activity instance should be assigned to persons in a child organization.

To make all persons in the specified organization and all of its child organizations eligible, set the value of this data member to 0.

To limit the persons who are eligible to the members of the specified organization and the managers of the first level of child organizations, set this data member to any nonzero value.

If not set, the default is 0. If no organization is set for the `_ACTIVITY_INFO.Organization` data member, any value set here is ignored.

Note: This criterion is ignored if any specific people are set using the `_ACTIVITY_INFO.People` data member.

Data type: long

`_ACTIVITY_INFO.LowerLevel`

The level persons must at least have to receive work items of the activity instance. A value between 0 and 9 can be set. The default value is 0 (zero).

If the level specified here is greater than the value specified for the upper level, or if the level is not set, the default value of 0 (zero) is used.

Note: This criterion is ignored if any specific people are set using the `_ACTIVITY_INFO.People` data member.

Data type: LONG

`_ACTIVITY_INFO.UpperLevel`

The level persons should not exceed to receive work items of the activity instance. A value between 0 and 9 can be set. The default value is 9.

If the level specified here is less than the value specified for the lower level, or the level is not set, the default value of 9 is used.

Note: This criterion is ignored if any specific people are set using the `_ACTIVITY_INFO.People` data member.

Data type: LONG

`_ACTIVITY_INFO.People`

This data member is used to specifically identify the people who should receive a work item of the activity instance. Multiple entries are possible and are then to be separated by a semicolon (;).

If any people are identified using this data member, any values set for data members `_ACTIVITY_INFO.MembersOfRoles`, `_ACTIVITY_INFO.CoordinatorOfRole`, `_ACTIVITY_INFO.Organization`, `_ACTIVITY_INFO.OrganizationType`, `_ACTIVITY_INFO.LowerLevel`, and `_ACTIVITY_INFO.UpperLevel` are ignored.

If no value is set, any values set for the above data members are used. If no values have been set for those, the values set for staff definition for the process instance are used.

If no values have been set for the process instance, the people in the organization and all child organizations of the process starter receive a work item for the activity instance.

Data type: STRING

`_ACTIVITY_INFO.PersonToNotify`

Used to identify the person to notify if the specified duration to complete the activity instance expires before the activity instance is complete.

If the user ID specified by the data member is invalid or the data member is not set, no person is notified.

Data type: LONG

`_ACTIVITY_INFO.Duration`

Used to specify the maximum number of seconds allowed to complete the activity.

If the activity is not completed before the specified duration, the defined person is notified.

If the value specified by the data member is invalid or the data member is not set, no notification occurs.

Data type: LONG

`_ACTIVITY_INFO.Duration2`

Used to specify the maximum number of hours allowed to act on an activity instance notification.

If the notification is not acted on before the specified number of hours expires, the process administrator is notified.

If the value specified by the data member is invalid or the data member is not set, no notification occurs.

Data type: LONG

Chapter 10. Monitoring a process instance

MQSeries Workflow allows for obtaining a monitor for a specified process instance. A process instance monitor typically allows for:

- Observing the progress of a process instance execution.
- Determining the state of execution, that is, to determine which activity instance is currently in progress, is waiting to be executed by whom, is InError and waiting for some action. It allows to determine whether notifications occurred because the maximum work time was exceeded.
- Viewing the history of execution, that is, what path has been taken through the process instance and why. It allows to determine where the bottlenecks of execution are or where the most time-consuming parts are.

Obtaining a process instance monitor

Once a process instance[3] has been accessed, a **process instance monitor** can be obtained. The transient process instance monitor object then represents all information about activity instances directly contained in the described process instance as well as all information on control connector instances connecting those activity instances.

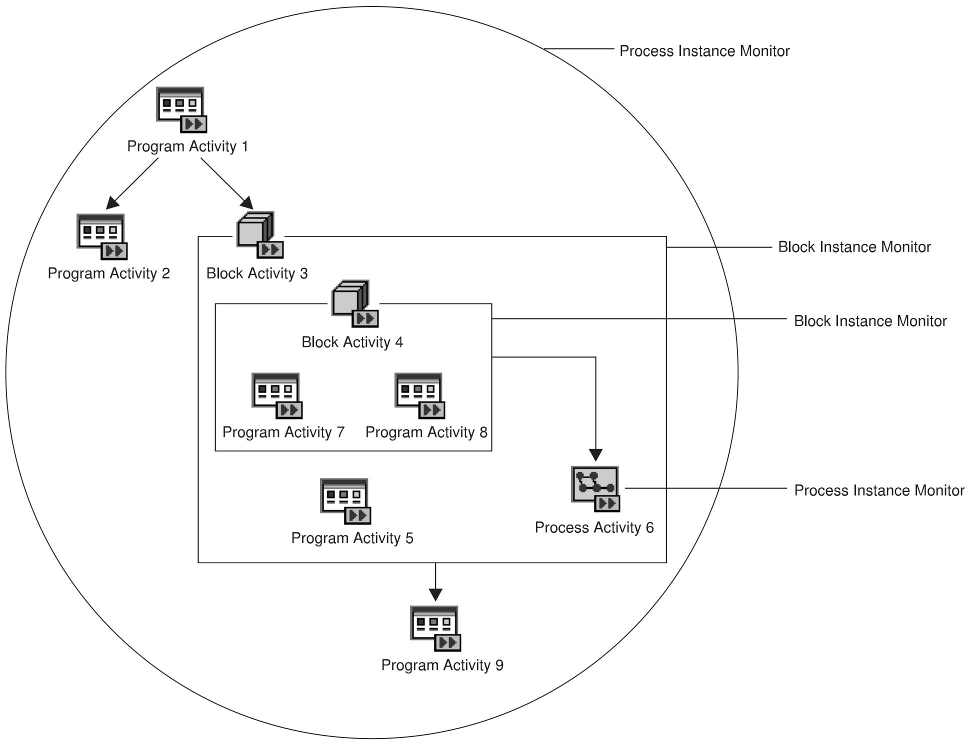


Figure 2. Process instance monitors and block instance monitors

3. or activity instance or a (work) item

For example, the illustrated process instance monitor describes two program activities, *Program Activity 1* and *Program Activity 2*, and an activity of type Block, *Block Activity 3*. There are two control connectors between these activities.

The process instance monitor can then be asked for the activity instances and the control connector instances described and their properties can be determined, for example, the state of the activity and its graphical layout, or the result of control connector instance evaluation and activities to connect or bend points to be drawn.

When an activity of type *Block* is encountered, it is possible to obtain its **block instance monitor**. Similar to a process instance monitor, a block instance monitor object represents all information about activity instances directly contained in the described block activity instance as well as all information on control connector instances connecting those activity instances. For example, the block instance monitor of *Block Activity 3* describes *Block Activity 4*, *Program Activity 5*, and *Process Activity 6*. There is a control connector between *Block Activity 4* and *Process Activity 6*.

When an activity of type *Process* is encountered, it is again possible to obtain its process instance monitor, either via the embracing monitor object or by retrieving the implementing (sub)process instance of the activity and then obtaining the associated process instance monitor. The process instance monitor obtained is a monitor which is completely separate from any other process instance monitor.

When obtaining a process instance monitor, it is possible to use the *deep option* in order to specify that *all* monitors for activities of kind Block are to be returned from the MQSeries Workflow execution server in the same step. The block instance monitors then all show the state of the process instance at this retrieval time. This means, when a block instance monitor is obtained via an API call, the API finds this monitor in its cache and provides it to the caller. When the deep option is not used, it can happen that a block instance monitor is not available. The API then automatically fetches the requested monitor from the execution server; it then represents a newer state than the ones previously retrieved.

Note: The deep option is currently ignored.

Ownership of monitors

As any other transient object, a process instance monitor is owned by the caller of the API. When a process instance monitor is no longer needed, you should delete/deallocate the object.

A block instance monitor, however, is considered to be part of a process instance monitor. It is cached by the API as part of the process instance monitor. It cannot be deallocated in the C language. Deletion in the C++ language only deletes the C++ representation but not the block instance monitor itself in the API cache. Block instance monitors are **automatically deleted** when the owning process instance monitor is deleted/deallocated. This means that block instance monitor objects or handles can only be used as long as the containing process instance monitor exists. When the process instance monitor does no longer exist, then using a block instance monitor object or handle will return unexpected results; your program can even trap since the usage of a nonexisting object or handle violates the MQSeries Workflow *programming by contract* concept.

Chapter 11. Authorization considerations

In general, authorization is granted to persons, either explicitly or implicitly. Implicitly means that the authority has been given as the result of performing some MQSeries Workflow action; performing that action can itself request some specific authority (See the *Administration* handbook for more detailed information).

Special authority is granted to a person playing the role of a *system administrator*. The system administrator has all privileges except on (work) items. Only the owner of a (work) item can issue any actions; the system administrator can, however, transfer the (work) item to himself. The system administrator role must be assigned to a single person at any time.

When a process instance is started, its *process administrator* is determined. The person determined to be the process administrator receives process administration rights for that process instance.

The person who is to become the process administrator of a process instance is specified when the process model is defined. Identification of the process administrator can be done in the following ways:

- Specification of a user identification for the PROCESS_ADMINISTRATOR keyword. In this case, the process administrator is already known when the process model is defined.
- Specification of a member in the process input container via the PROCESS_ADMINISTRATOR TAKEN_FROM specification.
- Specification of DATA FROM INPUT_CONTAINER. The process administrator is then taken from the process information member _PROCESS_INFO.ProcessAdministrator field in the input container (see "Process information data members" on page 41 for details).

The following table shows the authorizations and the MQSeries Workflow functions which can be called when that authority has been granted. The E/I (Explicit/Implicit) column indicates how the authorization is granted to persons.

Note: Once a user has authenticated himself to MQSeries Workflow (logged on), he can retrieve all objects he is authorized to see without any further special authorization. These are all objects he has created and all objects which are not specially secured or which are for public usage.

Table 3. Authorization for persons

Name	E/I	Authorized Functions
Authorization definition authorization	E	Create, update, and delete authorization information. Retrieve and update passwords. The appropriate FDL authorization keyword is AUTHORIZATION.
Operation administration authorization	E	Can perform all operation administration functions. The appropriate FDL authorization keyword is OPERATION.

Table 3. Authorization for persons (continued)

Name	E/I	Authorized Functions
Process modeling authorization	E	Create, retrieve, update, and delete process models and process templates. The appropriate FDL authorization keyword is PROCESS_MODELING.
Staff definition authorization	E	Create, retrieve, update, and delete staff information. As such, it includes authorization definition authorization. Create, retrieve, update, and delete public and private process instance lists, process template lists, and worklists. The appropriate FDL authorization keyword is STAFF.
Topology definition authorization	E	Create, retrieve, update, and delete topology information. The appropriate FDL authorization keyword is TOPOLOGY.
Process authorization	E	Can perform the following process instance functions for all process instances (global process authorization) or for process instances in categories authorized for (selected process authorization): • Create • Start • Create and start • Set process instance name • Query • Refresh Can perform the following process template functions for all process templates (global process authorization) or for process templates in categories authorized for (selected process authorization): • Query • Refresh The appropriate FDL authorization keyword is PROCESS_CATEGORY.
Process administration authorization	E	Has process authorization and can perform the following additional process instance functions for all process instances (global process administration authorization) or for process instances in categories authorized for (selected process administration authorization): • Delete • Suspend • Resume • Terminate Can perform the following work item functions for all process instances (global process administration authorization) or for process instances in categories authorized for (selected process administration authorization): • Force-finish • Force-restart The appropriate FDL authorization keyword is PROCESS_CATEGORY AS ADMINISTRATOR.

Table 3. Authorization for persons (continued)

Name	**E/I**	**Authorized Functions**
Process administrator	I	Has process administration authority for the appropriate process instance.
Process creator	I	Can perform the following process instance functions: • Set process instance name • Delete, if not yet started • Query • Refresh • Start
Work item authority	E	Can perform the following functions on (work) items for all (global work item authority) or for selected persons (selected work item authority): • Query • Refresh • Transfer The appropriate FDL authorization keyword is WORKITEMS_OF.
Workitem owner	I	Can perform all functions on the assigned (work) item except: • Force Finish • Force Restart

Chapter 12. Function/subprogram types

MQSeries Workflow functions/subprograms can be divided into several categories which characterize the kind and behavior of the request to be executed.

basic	to manage transient objects
accessor	to read properties of transient objects
action	to read or manipulate persistent objects
activity implementation	to deal with containers from within an activity implementation or support tool

Basic and accessor functions/subprograms are described in more detail but generally below. This is because all these functions/subprograms are similar looking and have similar requirements, even for different objects. They are all handled locally by the API, that is, they do not communicate with the server. The functions/subprograms of the other categories are described separately in "Part 7. Application programming interfaces" on page 213. Those are the functions/subprograms which require client/server communication or communication with the program execution server.

Basic functions/subprograms

Basic functions/subprograms are essentially provided so that transient objects can be allocated or constructed and deallocated or destructed. They allow for the construction of supporting objects like service objects. They allow for the destruction of such objects as well as for the destruction of transient representations of persistent objects allocated implicitly by the MQSeries Workflow API. Refer also to "Chapter 5. Memory management" on page 21.

Because of the nature of transient objects, neither a connection to a server nor some specific authorization is required to execute.

Return Codes

The C and Cobol language functions and the MQSeries Workflow result object can return the following codes, the number in parentheses shows their integer value:

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is expected, but 0 is passed.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

Basic functions/subprograms allow for the basic operations listed below; ’Xxx’ denotes some class or scope, for example, FmcjXxxEqual can stand for FmcjProcessInstanceEqual.

FmcjXxxEqual

Allows an application to compare two transient objects in order to determine whether they represent the same persistent or API object.

Comparison is done on the basis of the object identifiers. True is returned if both transient objects represent the same persistent object. The contents of the transient objects to be compared are not further checked, that is, it is not checked whether both transient objects carry the same states of the persistent object.

Exceptions:
- Service objects are equal when they represent the same session.
- Error objects are equal when they report the same error, that is, when they contain the same return code and the same parameters.
- Program data objects are equal when they belong to the same work item.
- Control connector instance objects are equal when they have the same source and target activity instances.
- Point and symbol layout objects are equal when their properties are equal.

In the C and Cobol language, the return code of the result object is set to ″invalid handle″, if one of the handles passed is invalid. True is returned, if both are invalid, else false.

C language signature

```
bool FMC_APIENTRY FmcjXxxEqual( FmcjXxxHandle handle1,
                                FmcjXxxHandle handle2 )
```

Cobol-language signature

```
FmcjXxxEqual.

   CALL    "FmcjXxxEqual"
                  USING
                  BY VALUE
                     handle1
                     handle2
                  RETURNING
                     boolReturnValue.
```

Parameters

handle1
Input. The first object to be compared.

handle2
Input. The other object to be compared.

FmcjXxxAllocate()

Allows the application to set up the respective object. This is needed for supporting objects like string vectors. Transient objects representing persistent objects are allocated implicitly by the MQSeries Workflow API when persistent objects are created or queried from an MQSeries Workflow server.

All constructed objects are transient.

C language signature

```
APIRET FMC_APIENTRY FmcjXxxAllocate( FmcjXxxHandle * handle )
```

Cobol language signature

```
       FmcjXxxAllocate.

          CALL    "FmcjXxxAllocate"
                         USING
                         BY REFERENCE
                            handle
                         RETURNING
                            intReturnValue.
```

Parameters

handle
: Input/Output. The address of the handle to the object to be set when the object has been constructed. Care that the handle passed is not pointing to a still valid object since that object is not automatically deallocated before the new object’s handle is set.

A constructor or FmcjXxxAllocate function/subprogram can require additional input parameters. See the header files or copybooks for the exact declarations.

FmcjXxxCopy()

Allows the application to make a copy of a particular transient object. That copy becomes a separate object and thus carries its own state.

An exception is the execution service where a copy points to the same session established by the original object. This especially means, when you request to log off on either object, then the (common) session is closed.

C language signature

```
APIRET FMC_APIENTRY FmcjXxxCopy( FmcjXxxHandle   handle,
                                 FmcjXxxHandle * newHandle )
```

Cobol language signature

```
        FmcjXxxCopy.

           CALL    "FmcjXxxCopy"
                          USING
                          BY VALUE
                             handle
                          BY REFERENCE
                             newHandle
                          RETURNING
                             intReturnValue.
```

Parameters

handle
: Input. The handle of the object to be copied.

newHandle
: Input/Output. The address of a handle to be set when the object has been constructed. Care that the handle passed is not pointing to a still valid object since that object is not automatically deallocated before the new object's handle is set.

FmcjXxxDeallocate

Allows the application to delete the specified transient object. Deletion of a transient object has no impact on the represented persistent object, if any.

The C language handle is set to 0 so that it can no longer be used.

C language signature

```
APIRET FMC_APIENTRY FmcjXxxDeallocate( FmcjXxxHandle * handle )
```

Cobol language signature

```
        FmcjXxxDeallocate.

           CALL    "FmcjXxxVectorDeallocate"
                          USING
                          BY REFERENCE
                             handle
                          RETURNING
                             intReturnValue.
```

Parameters

handle
: Input/Output. The address of the handle to the object to be deallocated.

FmcjXxxIsComplete()

Returns true when the object has been completely read from an MQSeries Workflow server, that is, both primary and secondary properties are available (see also “Accessor function/subprograms” on page 62).

C language signature

```
bool FMC_APIENTRY FmcjXxxIsComplete( FmcjXxxHandle handle )
```

Cobol language signature

```
        FmcjXxxIsComplete.

           CALL    "FmcjXxxIsComplete"
                          USING
                          BY VALUE
                             handle
                          RETURNING
                             boolReturnValue.
```

Parameters

handle
Input. The handle of the object to be queried.

Return type

bool True if the object has been completely read from the server, otherwise false.

C language Example: using basic functions

```
#include <stdio.h>
#include <fmcjcrun.h>
int main()
{
  APIRET                       rc;
  FmcjExecutionServiceHandle  service   = 0;
  FmcjWorkitemVectorHandle    wList     = 0;
  FmcjWorkitemHandle          workitem1 = 0;
  FmcjWorkitemHandle          workitem2 = 0;
  FmcjWorkitemHandle          workitem3 = 0;

  FmcjGlobalConnect();

  /* logon */
  FmcjExecutionServiceAllocate(&service);
  rc = FmcjExecutionServiceLogon( service,
                                  "USERID", "password",
                                  Fmc_SM_Default, Fmc_SA_Reset
                                );

  /*  Query Workitems */
  rc= FmcjExecutionServiceQueryWorkitems( service,
                                          FmcjNoFilter,
                                          FmcjNoSortCriteria,
                                          FmcjNoThreshold,
                                          &wList );
  printf( "\nQuery workitems returns rc : %u\n", rc );
  fflush(stdout);
  if ( rc == FMC_OK && FmcjWorkitemVectorSize(wList) >= 2 )
  {                                    /* access first element         */
    workitem1= FmcjWorkitemVectorFirstElement(wList);
    if ( FmcjWorkitemIsComplete(workitem1) )
      printf( "Surprise - more than primary data available\n" );
    else
      printf( "Primary data of first workitem available\n" );
    fflush(stdout);
                                       /* access next element          */
    workitem2= FmcjWorkitemVectorNextElement(wList) ;
    if ( FmcjWorkitemEqual(workitem1,workitem2) )
      printf( "Surprise - workitems are equal\n" );
    else
      printf( "Workitems represent different objects\n" );
    fflush(stdout);
                                       /* copy workitem                */
    FmcjWorkitemCopy(workitem1,&workitem3);
    if ( ! FmcjWorkitemEqual(workitem1,workitem3) )
      printf( "Surprise - workitems are not equal\n" );
    else
      printf( "Workitems represent same persistent object\n" );
    fflush(stdout);
                                       /* cleanup                      */
    FmcjWorkitemDeallocate(&workitem1);
    FmcjWorkitemDeallocate(&workitem2);
    FmcjWorkitemDeallocate(&workitem3);
  }

  FmcjWorkitemVectorDeallocate( &wList );

  /* logoff */
  FmcjExecutionServiceLogoff(service);
  FmcjExecutionServiceDeallocate(&service);

  FmcjGlobalDisconnect();
  return FMC_OK;
}
```

Cobol Example: using basic functions

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. "BASIC".

       DATA DIVISION.

         WORKING-STORAGE SECTION.

         COPY fmcvars.
         COPY fmcconst.
         COPY fmcrcs.

         01 localUserID   PIC X(30) VALUE z"USERID".
         01 localPassword PIC X(30) VALUE z"PASSWORD".
         01 workitem1     USAGE IS POINTER VALUE NULL.
         01 workitem2     USAGE IS POINTER VALUE NULL.
         01 workitem3     USAGE IS POINTER VALUE NULL.

         LINKAGE SECTION.

         01 retCode            PIC S9(9) BINARY.

       PROCEDURE DIVISION USING retCode.

            PERFORM FmcjGlobalConnect.
      * logon
            PERFORM FmcjESAllocate.
```

```
           CALL "SETADDR" USING localUserId userId.
           CALL "SETADDR" USING localPassword passwordValue.
           MOVE Fmc-SM-Default TO sessionMode.
           MOVE Fmc-SA-Reset TO absenceIndicator.
           PERFORM FmcjESLogon.
      * Query Workitems
           CALL "SETADDR" USING FmcjNoFilter filter.
           CALL "SETADDR" USING FmcjNoSortCriteria sortCriteria.
           MOVE FmcjNoThreshold TO threshold.
           PERFORM FmcjESQueryWorkitems.
           SET hdlVector TO workitems.
           MOVE intReturnValue TO retCode.
           DISPLAY "Query Workitems returns rc : " retCode.

           IF retCode = FMC-OK
              PERFORM FmcjWIVSize
              IF ulongReturnValue >= 2

      * access first element
                 PERFORM FmcjWIVFirstElement
                 SET workitem1 TO FmcjWIHandleReturnValue
                 SET hdlItem TO workitem1
                 PERFORM FmcjWIIsComplete
                 IF boolReturnValue = 1
                    DISPLAY "Surprise - more than primary data"
                    DISPLAY "available"
                 ELSE
                    DISPLAY "Primary data of first workitem"
                    DISPLAY "available"
                 END-IF

      * access next element
                 PERFORM FmcjWIVNextElement
                 SET workitem2 TO FmcjWIHandleReturnValue
                 SET hdlItem2 TO workitem2
                 PERFORM FmcjWIEqual
                 IF boolReturnValue = 1
                    DISPLAY "Surprise - workitems are equal"
                 ELSE
                    DISPLAY "Workitems represent different objects"
                 END-IF

      * copy workitem
                 SET hdlWorkitem TO workitem1
                 PERFORM FmcjWICopy
                 SET workitem3 TO newWorkItem
                 SET hdlItem2 TO workitem3
                 PERFORM FmcjWIEqual
                 IF boolReturnValue = 0
                    DISPLAY "Surprise - workitems are not equal"
                 ELSE
                    DISPLAY "Workitems represent same persistent"
                    DISPLAY "objects"
                 END-IF

      * cleanup
                 SET hdlWorkitem TO workitem1
                 PERFORM FmcjWIDeallocate
                 SET hdlWorkitem TO workitem2
                 PERFORM FmcjWIDeallocate
                 SET hdlWorkitem TO workitem3
                 PERFORM FmcjWIDeallocate
              END-IF
           END-IF

           PERFORM FmcjWIVDeallocate.
```

```
* logoff
      PERFORM FmcjESLogoff.
      DISPLAY "Logged off".
      PERFORM FmcjESDeallocate.
      PERFORM FmcjGlobalDisconnect.
      MOVE FMC-OK TO retCode.
      GOBACK.

      COPY fmcperf.
 IDENTIFICATION DIVISION.
 PROGRAM-ID. "BASIC".

 DATA DIVISION.

   WORKING-STORAGE SECTION.

   COPY fmcvars.
   COPY fmcconst.
   COPY fmcrcs.

   01 localUserID   PIC X(30) VALUE z"USERID".
   01 localPassword PIC X(30) VALUE z"PASSWORD".
   01 workitem1     USAGE IS POINTER VALUE NULL.
   01 workitem2     USAGE IS POINTER VALUE NULL.
   01 workitem3     USAGE IS POINTER VALUE NULL.

   LINKAGE SECTION.

   01 retCode            PIC S9(9) BINARY.

 PROCEDURE DIVISION USING retCode.

      CALL "FmcjGlobalConnect".
      CALL "FmcjExecutionServiceAllocate"
         USING BY REFERENCE serviceValue
         RETURNING intReturnValue.
* logon
      CALL "SETADDR" USING localUserId userId.
      CALL "SETADDR" USING localPassword passwordValue.
      CALL "FmcjExecutionServiceLogon"
         USING BY VALUE serviceValue
                        userID
                        passwordValue
                        Fmc-SM-Default
                        Fmc-SA-Reset
         RETURNING intReturnValue.
* Query Workitems
      CALL "SETADDR" USING FmcjNoFilter filter.
      CALL "SETADDR" USING FmcjNoSortCriteria sortCriteria.
      CALL "FmcjExecutionServiceQueryWorkitems"
         USING BY VALUE     serviceValue
                            filter
                            sortCriteria
                            FmcjNoThreshold
               BY REFERENCE workitems
         RETURNING intReturnValue.
      MOVE intReturnValue TO retCode.
      DISPLAY "Query Workitems returns rc : " retCode.

      IF retCode = FMC-OK
         CALL "FmcjWorkitemVectorSize"
            USING BY VALUE workitems
            RETURNING ulongReturnValue

         IF ulongReturnValue >= 2

* access first element
```

```
            CALL "FmcjWorkitemVectorFirstElement"
               USING BY VALUE workitems
               RETURNING FmcjWIHandleReturnValue
            SET workitem1 TO FmcjWIHandleReturnValue
            CALL "FmcjItemIsComplete"
               USING BY VALUE workitem1
               RETURNING boolReturnValue
            IF boolReturnValue = 1
               DISPLAY "Surprise - more than primary data"
               DISPLAY "available"
            ELSE
               DISPLAY "Primary data of first workitem"
               DISPLAY "available"
            END-IF

* access next element
            CALL "FmcjWorkitemVectorNextElement"
               USING BY VALUE workitems
               RETURNING FmcjWIHandleReturnValue
            SET workitem2 TO FmcjWIHandleReturnValue
            CALL "FmcjItemEqual"
               USING BY VALUE workitem1
                              workitem2
               RETURNING boolReturnValue
            IF boolReturnValue = 1
               DISPLAY "Surprise - workitems are equal"
            ELSE
               DISPLAY "Workitems represent different objects"
            END-IF

* copy workitem
            CALL "FmcjWorkitemCopy"
               USING BY VALUE workitem1
                     BY REFERENCE workitem3
               RETURNING intReturnValue
            CALL "FmcjItemEqual"
               USING BY VALUE workitem1
                              workitem3
               RETURNING boolReturnValue
             IF boolReturnValue = 0
               DISPLAY "Surprise - workitems are not equal"
            ELSE
               DISPLAY "Workitems represent same persistent"
               DISPLAY "objects"
            END-IF

* cleanup
            CALL "FmcjWorkitemDeallocate"
               USING BY REFERENCE workitem1
               RETURNING intReturnValue
            CALL "FmcjWorkitemDeallocate"
               USING BY REFERENCE workitem2
               RETURNING intReturnValue
            CALL "FmcjWorkitemDeallocate"
               USING BY REFERENCE workitem2
               RETURNING intReturnValue
         END-IF
      END-IF
      CALL "FmcjWorkitemVectorDeallocate"
         USING BY REFERENCE workitems
         RETURNING intReturnValue.

* logoff
      CALL "FmcjExecutionServiceLogoff"
         USING BY VALUE serviceValue
         RETURNING intReturnValue.
      DISPLAY "Logged off".
```

```
CALL "FmcjExecutionServiceDeallocate"
   USING BY REFERENCE serviceValue
   RETURNING intReturnValue.
CALL "FmcjGlobalDisconnect".
MOVE FMC-OK TO retCode.
GOBACK.
```

Accessor function/subprograms

Accessor functions/subprograms are provided so that properties of transient objects can be read or changed. If the transient object represents a persistent one, then the values that are returned reflect the state of the persistent object when it was retrieved and used to set the transient object or when it was created or updated. Retrieval has been done from an MQSeries Workflow server using the appropriate create, query, or refresh functions/subprograms. Creation or update can be done on the client when the MQSeries Workflow server sends new information (pushes information).

Default values are provided to you as long as the transient object is *empty* or *not complete*, or when the accessed property is *optional* and not set.

Default values are: an empty string or buffer for character-valued properties, 0 (zero) for integer-valued properties, false for boolean-valued properties, a timestamp with all members set to 0 (zero) for time-valued properties, "NotSet" for enumeration-valued properties, and an empty vector for multi-valued properties.

By default, the MQSeries Workflow API provides for two views on persistent objects. They divide the persistent object into so-called *primary* properties and so-called *secondary* properties. Primary properties are considered “more important” from an access point of view. They are immediately returned when objects are queried. Secondary properties, and a refresh of the primary properties, are only returned on an explicit Refresh() request; on a per-object basis. You can use the *IsComplete()* function/subprogram to determine whether both primary and secondary object values have been read from the server.

Besides being primary or secondary, properties of a persistent object can be optional. This means that they can carry a value or not. When a default value is returned to you, you can use the *IsNull()* function/subprogram to determine whether that value is a value explicitly set or whether that value actually denotes that no value has been set. For example, when Threshold() returns 0 (zero), the threshold can have been set to zero, that is, no object is returned to you, or the threshold cannot have been set to a value, that is, all qualifying objects are returned to you.

Note that being Null is a concept orthogonal to being completely read. As long as the object is not complete, IsNull() will return true for a secondary, optional property because nothing is known yet about the actual value and whether it has been set or not. For example, the documentation is a secondary and optional property of an object. When the object has been queried, then only the primary properties have been retrieved from the server. The Documentation() function/subprogram returns an empty string or buffer. To determine whether a documentation has been set at all, you can use the DocumentationIsNull() function/subprogram. The result will be “true” independent from the actual documentation setting as long as IsComplete() returns false. The documentation is assumed to be not set as long as the secondary data has not been retrieved.

Data values are accessible as long as the transient objects exist, regardless of the state of the persistent objects or of the current logon or logoff state. In general, you decide about the lifetime of your transient objects.

Because of the nature of transient objects, neither a connection to a server nor some specific authorization is required to access object properties. The operations listed below are headlined according to the C type. For more detailed information on the corresponding Cobol types see Table 5 on page 82.

Return codes

Accessor functions/subprograms provide the value asked for as their return value. Default values are returned when an error occurred during the execution of the accessor function/subprogram. You can query the MQSeries Workflow result object for any errors encountered. It can contain the following codes, the number in parentheses shows their integer value:

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is expected, but 0 is passed.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_EMPTY(122)
The object has not yet been read from the database, that means default values are returned.

FMC_ERROR_BUFFER(800)
The buffer provided is too small to hold the largest possible value. See file fmcmxcon.h for required lengths.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

Accessor functions/subprograms allow for the operations listed below; ’Xxx’ denotes some class or scope and ″Property″ denotes the property queried. For example, FmcjXxxProperty() can stand for FmcjItemDescription().

Accessing a value of type bool

Returns the value of a property of type *bool*. A default of *false* is returned if no information is available.

C language signature

```
bool FMC_APIENTRY FmcjXxxProperty( FmcjXxxHandle handle )
```

Cobol language signature

```
        FmcjXxxProperty.

           CALL    "FmcjXxxProperty"
                          USING
                          BY VALUE
                             handle
                          RETURNING
                             boolReturnValue.
```

Parameters

handle
Input. The handle of the object to be queried.

Return type

bool The property value.

Accessing a value of type char

Returns the value of a property of type *char*. An empty string or buffer is returned if no information is available. For an explanation of Cobol String handling in IBM MQSeries Workflow for OS/390 see "Chapter 13. Cobol specific considerations" on page 81.

C language signature

```
char * FMC_APIENTRY FmcjXxxProperty( FmcjXxxHandle handle,
                                     char *        buffer,
                                     unsigned long bufferLength )
```

Cobol language signature

```
        FmcjXxxProperty.

           CALL    "FmcjXxxProperty"
                          USING
                          BY VALUE
                             handle
                             buffer
                             bufferlength
                          RETURNING
                             pointerReturnValue.
```

Parameters

handle
Input. The handle of the object to be queried.

buffer Input/Output. A pointer to a buffer to contain the property value.

bufferLength
Input. The length of the buffer; must be big enough to hold the largest

possible value (see file fmcmxcon.h for the minimum required lengths). You can use a single buffer for retrieving all your character values.

Return type

char*/string
: The property value.

Accessing a value of type date/time

Returns the value of a property of type *FmcjCDateTime*. A zero timestamp is returned if no information is available.

C language signature

```
FmcjCDateTime FMC_APIENTRY FmcjXxxProperty( FmcjXxxHandle handle )
```

Cobol language signature

```
         FmcjXxxProperty.

            CALL    "FmcjXxxProperty"
                          USING
                          BY VALUE
                             handle
                          RETURNING
                             dateTimeReturnValue.
```

Parameters

handle
: Input. The handle of the object to be queried.

Return type

FmcjCDateTime
: The property value.

Accessing an enumerated value

Returns an enumerating value of a property. It is strongly advised to use the symbolic names in order to determine the actual value instead of the corresponding integer values. It is not guaranteed that integer values always stay the same.

"NotSet" or a similar indicator is returned if no information is available.

C language signature

```
enum FmcjXxxEnum FMC_APIENTRY FmcjXxxProperty( FmcjXxxHandle handle )
```

Cobol language signature

```
        FmcjXxxProperty.

           CALL    "FmcjXxxProperty"
                          USING
                          BY VALUE
                             handle
                          RETURNING
                             intReturnValue.
```

Parameters

handle
Input. The handle of the object to be queried.

Return type

FmcjXxxEnum
The property value, some element of an enumeration.

Accessing a value of type (unsigned) long

Returns the value of a property of type *long* or *unsigned long*. Zero (0) is returned if no information is available.

C language signature

```
long FMC_APIENTRY FmcjXxxProperty( FmcjXxxHandle handle )
unsigned long FMC_APIENTRY FmcjXxxProperty( FmcjXxxHandle handle )
```

Cobol language signature, type unsigned long

```
        FmcjXxxProperty.

           CALL    "FmcjXxxProperty"
                          USING
                          BY VALUE
                             handle
                          RETURNING
                             ulongReturnValue.
```

Cobol language signature, type long

```
        FmcjXxxProperty.

           CALL    "FmcjXxxProperty"
                          USING
                          BY VALUE
                             handle
                          RETURNING
                             longReturnValue.
```

Parameters

handle
: Input. The handle of the object to be queried.

Return type

long/unsigned long
: The property value.

Accessing a multi-valued property

Returns the value of a multi-valued property by providing a vector of values. The vector itself has to be provided by the caller. Use the appropriate vector accessor functions/subprograms to read a single value (refer to “Handling collections” on page 30).

An unchanged vector is returned if no information is available.

All values are appended to the supplied vector. If you want to read the actual values only, you have to clear the vector before you call the accessor function/subprogram. This means that you should set the vector handle to 0 via the FmcjXxxVectorDeallocate function.

C language signature

```
FmcjYyyVectorHandle FMC_APIENTRY FmcjXxxProperty( FmcjXxxHandle handle )
```

Cobol language signature

```
      FmcjXxxProperty.

         CALL    "FmcjXxxProperty"
                        USING
                        BY REFERENCE
                           handle
                        RETURNING
                           intReturnValue.
```

Parameters

handle
: Input. The handle of the object to be queried.

Return type

FmcjYyyVectorHandle
: The vector of values of the property.

FmcjXxxIsNull()

This function/subprogram states whether an optional property is set.

When the property is a secondary property and the object queried is not yet completely read, it is unknown whether the property is set or not so that a default value of true is returned.

C language signature

```
bool FMC_APIENTRY FmcjXxxPropertyIsNull( FmcjXxxHandle handle )
```

Cobol language signature

```
        FmcjXxxPropertyIsNull.

         CALL    "FmcjXxxPropertyIsNull"
                 USING
                 BY VALUE
                    handle
                 RETURNING
                    boolReturnValue.
```

Parameters

handle
Input. The handle of the object to be queried.

Return type

bool True if the property is not set, otherwise false.

Setting a value of type long

This function/subprogram sets the specified property to the specified value.

C language signature

```
void FMC_APIENTRY FmcjXxxSetProperty( FmcjXxxHandle handle,
                                      long          newValue )
```

Cobol language signature

```
        FmcjXxxSetProperty.

           CALL    "FmcjXxxSetProperty"
                          USING
                          BY VALUE
                             handle
                             newValue
```

Parameters

handle
Input. The handle of the object to be queried.

newValue
Input. The new value of the property.

Return type

bool True if the property is not set, otherwise false.

An example is the FmcjServiceSetTimeout function/subprogram which sets the timeout value for requests issued by the client to an MQSeries Workflow server via this FmcjService object. In other words, it sets the time the client is willing to wait for an answer.

When set, the new timeout value is used for all functions/subprograms requiring communication between the client and the server. It can be set (changed) as often as wanted. It is to be provided as microseconds. A negative value is interpreted as -1, that is, an indefinite timeout.

The default timeout value is taken from the user's profile; if not found, from the workstation profile. If it is also not found there, the default is 180000 ms.

Note: It is possible that, even though FMC_ERROR_TIMEOUT is returned when you issue a client-server call, the MQSeries Workflow server has successfully processed the request. However, the server could not send back FMC_OK because communication reported a timeout in the meantime. If the request has not been processed, increase the value set for the timeout and retry the call.

Setting a value of type unsigned short

This function/subprogram sets the specified property to the specified value.

C language signature

```
void FMC_APIENTRY FmcjXxxSetProperty( FmcjXxxHandle  handle,
                                      unsigned short newValue)
```

Cobol language signature

```
         FmcjXxxSetProperty.

            CALL    "FmcjXxxSetProperty"
                           USING
                           BY VALUE
                              handle
                              newValue.
```

Parameters

handle
Input. The handle of the object to be queried.

newValue
Input. The new value of the property.

Setting a value of type FmcjBinary *

This function/subprogram sets the specified property to the specified value.

C language signature

```
void FMC_APIENTRY FmcjXxxSetProperty( FmcjXxxHandle      handle,
                                      FmcjBinary const * newValue,
                                      unsigned long      dataLength)
```

Cobol language signature

```
        FmcjXxxSetProperty.

           CALL    "FmcjXxxSetProperty"
                          USING
                          BY VALUE
                             handle
                             newValue
                             dataLength.
```

Parameters

handle
Input. The handle of the object to be queried.

newValue
Input. The new value of the property.

dataLength
Input. The length of the new value.

Updating an object

This function/subprogram updates the specified object with information sent from an MQSeries Workflow server. The update information must have been provided for the specified object.

The server pushes update information for work items—as long as they are not disabled, activity instance notifications, and process instance notifications. The process setting of the associated process instance must specify REFRESH_POLICY PUSH for that process instance itself or as a process default. Logon must have been performed with a present session mode.

C language signature

```
APIRET FMC_APIENTRY FmcjXxxUpdate( FmcjXxxHandle          handle,
                                   FmcjExecutionDataHandle data );
```

Cobol language signature

```
FmcjXxxUpdate.

     CALL    "FmcjXxxUpdate"
               USING
               BY REFERENCE
                  handle
                  data
               RETURNING
                  intReturnValue.
```

Parameters

handle
: Input. The handle of the object to be updated.

data
: Input. The data which is to be used for the update.

Return codes

The C language functions and the MQSeries Workflow result object can return the following codes, the number in parentheses shows their integer value:

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is expected, but 0 is passed.

FMC_ERROR_EMPTY(122)
: The object has not yet been read from the database, that is, it does not yet represent a persistent one.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_INVALID_OID(805)
: The execution data is no data to update the specified object; it does not belong to the specified object.

FMC_ERROR_WRONG_KIND(501)
: The execution data is no data to update the specified object; it is no update data.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

C language Example: accessing values

```
#include <stdio.h>
#include <fmcjcrun.h>
int main()
{
  APIRET                      rc;
  FmcjExecutionServiceHandle  service   = 0;
  FmcjWorkitemHandle          workitem  = 0;
  FmcjStringVectorHandle      sList     = 0;
  char                        category[FMC_CATEGORY_NAME_LENGTH+1]
```

```
    char                        generalBuffer[200]
    unsigned long               priority  = 0;
    int                         enumValue = 0;
    FmcjCDateTime               startTime;
    unsigned long               i         = 0;

    FmcjGlobalConnect();

    /* logon */
    FmcjExecutionServiceAllocate(&service);
    rc = FmcjExecutionServiceLogon( service,
                                    "USERID", "password",
                                    Fmc_SM_Default, Fmc_SA_Reset
                                  );
                                         /* set the timeout for requests */
    FmcjExecutionServiceSetTimeout( service, 60000 );

    /* assumption: workitem has been queried from the server            */

                                         /* access a value of type bool  */
    if ( FmcjWorkitemCategoryIsNull( workitem ) )
      printf( "Category is not set\n" );
    else                                 /* access a value of type char  */
    {                                    /* use a buffer which fits      */
      FmcjWorkitemCategory( workitem, category, FMC_CATEGORY_NAME_LENGTH+1 );
      printf( "Category   : %s\n", category );
    }
                                         /* access a date/time value     */
    startTime= FmcjWorkitemCategoryStartTime( workitem );
    printf( "Start time : %s\n",
      FmcjDateTimeAsString("startTime, generalBuffer, 200) );
                                         /* access a value of type long  */
    priority = FmcjWorkitemPriority( workitem );
    printf( "Priority   : %u\n", priority );
                                         /* access an enumerated value   */
    enumValue= FmcjWorkitemCategoryReceivedAs( workitem );
    if ( enumValue == Fmc_IR_Normal )
      printf( "Received as: %s\n","qualified user" );
    ...
                                         /* access a multi-valued field  */
     sList= FmcjWorkitemSupportTools( workitem );
     printf( "Support tools: " );
     for( i=0; i< FmcjStringVectorSize(sList); i++ )
     {                                   /* use a large buffer           */
       printf("%s ", FmcjStringVectorNextElement(sList, generalBuffer, 200) );
     }

    /* logoff */
    FmcjExecutionServiceLogoff(service);
    FmcjExecutionServiceDeallocate(&service);

    FmcjGlobalDisconnect();
    return FMC_OK;
  }
```

Cobol Example: accessing values

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. "VALUES".

       DATA DIVISION.

         WORKING-STORAGE SECTION.

         COPY fmcvars.
         COPY fmcconst.
         COPY fmcrcs.
```

```
   01 localUserID   PIC X(30) VALUE z"USERID".
   01 localPassword PIC X(30) VALUE z"PASSWORD".
   01 categBuffer   PIC X(34).
   01 generalBuffer PIC X(200).
   01 i             PIC 9(9) BINARY VALUE 0.

   LINKAGE SECTION.

   01 retCode            PIC S9(9) BINARY.

 PROCEDURE DIVISION USING retCode.

      PERFORM FmcjGlobalConnect.
* logon
      PERFORM FmcjESAllocate.

      CALL "SETADDR" USING localUserId userId.
      CALL "SETADDR" USING localPassword passwordValue.
      MOVE Fmc-SM-Default TO sessionMode.
      MOVE Fmc-SA-Reset TO absenceIndicator.
      PERFORM FmcjESLogon.
* set the timeout for requests
      MOVE 60000 TO newTimeOutValue.
      PERFORM FmcjESSetTimeout.

* assumption: workitem has been queried from the server
* and hdlItem points to this workitem

* access a value of type bool (PIC 9 BINARY)
      PERFORM FmcjWICategIsNull.
      IF boolReturnValue = 1
         DISPLAY "Category is not set"
      ELSE
* access a value of type char (POINTER to a PIC X(n))
*        use a buffer which fits
         CALL "SETADDR" USING categBuffer categoryNameBuffer
         MOVE FMC-CATEG-NAME-LENGTH TO bufferLength
         PERFORM FmcjWICateg
         DISPLAY "Category   : " categBuffer
      END-IF
* access a date/time value
      PERFORM FmcjWIStartTime.
      MOVE dateTimeReturnValue TO timeValue.
      CALL "SETADDR" USING generalBuffer dateTimeBuffer.
      MOVE 200 TO bufferLength.
      PERFORM FmcjDateTimeAsString.
      DISPLAY "Start time : " generalBuffer.
* access a value of type unsigned long (PIC 9(9) BINARY)
      SET hdlWorkitem TO hdlItem.
      PERFORM FmcjWIPriority.
      DISPLAY "Priority   : " ulongReturnValue.
* access an enumerated value (PIC S9(9) BINARY)
      PERFORM FmcjWIReceivedAs.
      IF intReturnValue = Fmc-IR-Normal
         DISPLAY "Received as: qualified user"
      END-IF
* access a multi-valued field
      PERFORM FmcjWISupportTools.
      SET hdlVector TO FmcjStrVHandleReturnValue.
      PERFORM FmcjStrVSize.
      DISPLAY "Support tools: ".
*     use a large buffer
      CALL "SETADDR" USING generalBuffer elementBuffer
      PERFORM VARYING i FROM 0 BY 1 UNTIL i >= ulongReturnValue
         PERFORM FmcjStrVNextElement
         DISPLAY generalBuffer
      END-PERFORM
* logoff
      PERFORM FmcjESLogoff.
      DISPLAY "Logged off".
      PERFORM FmcjESDeallocate.
      PERFORM FmcjGlobalDisconnect.
      MOVE FMC-OK TO retCode.
      GOBACK.

      COPY fmcperf.
 IDENTIFICATION DIVISION.
 PROGRAM-ID. "VALUES".

 DATA DIVISION.

   WORKING-STORAGE SECTION.

   COPY fmcvars.
   COPY fmcconst.
   COPY fmcrcs.

   01 localUserID   PIC X(30) VALUE z"USERID".
   01 localPassword PIC X(30) VALUE z"PASSWORD".
   01 categBuffer   PIC X(34).
   01 generalBuffer PIC X(200).
   01 i             PIC 9(9) BINARY VALUE 0.

   LINKAGE SECTION.

   01 retCode            PIC S9(9) BINARY.

 PROCEDURE DIVISION USING retCode.

      CALL "FmcjGlobalConnect".
* logon
      CALL "FmcjExecutionServiceAllocate"
         USING BY REFERENCE serviceValue
         RETURNING intReturnValue.

      CALL "SETADDR" USING localUserId userId.
      CALL "SETADDR" USING localPassword passwordValue.
      CALL "FmcjExecutionServiceLogon"
         USING BY VALUE serviceValue
                        userID
                        passwordValue
                        Fmc-SM-Default
                        Fmc-SA-Reset
         RETURNING intReturnValue.

* set the timeout for requests
      MOVE 60000 TO newTimeOutValue.
      CALL "FmcjServiceSetTimeout"
         USING BY VALUE serviceValue
                        newTimeoutValue.

* assumption: workitem has been queried from the server
* and hdlItem points to this workitem

* access a value of type bool (PIC 9 BINARY)
      CALL "FmcjItemCategoryIsNull"
         USING BY VALUE hdlItem
         RETURNING boolReturnValue.
      IF boolReturnValue = 1
         DISPLAY "Category is not set"
      ELSE
* access a value of type char (POINTER to a PIC X(n))
*        use a buffer which fits
```

```
            CALL "SETADDR" USING categBuffer categoryNameBuffer
            MOVE FMC-CATEG-NAME-LENGTH TO bufferLength
            CALL "FmcjItemCategory"
               USING BY VALUE hdlItem
                              categoryNameBuffer
                              bufferLength
               RETURNING pointerReturnValue
            DISPLAY "Category   : " categBuffer
         END-IF
      * access a date/time value
         CALL "FmcjItemStartTime"
            USING BY VALUE hdlItem
            RETURNING dateTimeReturnValue.
         CALL "SETADDR" USING generalBuffer dateTimeBuffer.
         MOVE 200 TO bufferLength.
         CALL "FmcjDateTimeAsString"
            USING BY REFERENCE dateTimeReturnValue
                  BY VALUE     dateTimeBuffer
                               bufferLength
            RETURNING pointerReturnValue.
         DISPLAY "Start time : " generalBuffer.
      * access a value of type unsigned long (PIC 9(9) BINARY)
         SET hdlWorkitem TO hdlItem.
         CALL "FmcjWorkitemPriority"
            USING BY VALUE hdlItem
            RETURNING ulongReturnValue.
         DISPLAY "Priority   : " ulongReturnValue.
      * access an enumerated value (PIC S9(9) BINARY)
         CALL "FmcjItemReceivedAs"
            USING BY VALUE hdlItem
            RETURNING intReturnValue.
          IF intReturnValue = Fmc-IR-Normal
            DISPLAY "Received as: qualified user"
         END-IF
      * access a multi-valued field
         CALL "FmcjWorkitemSupportTools"
            USING BY VALUE hdlItem
            RETURNING FmcjStrVHandleReturnValue.
         SET hdlVector TO FmcjStrVHandleReturnValue.
         CALL "FmcjStringVectorSize"
            USING BY VALUE hdlVector
            RETURNING ulongReturnValue.
         DISPLAY "Support tools: ".
      *  use a large buffer
         CALL "SETADDR" USING generalBuffer elementBuffer
         PERFORM VARYING i FROM 0 BY 1 UNTIL i >= ulongReturnValue
         CALL "FmcjStringVectorNextElement"
            USING BY VALUE hdlVector
                           elementBuffer
                           bufferLength
            RETURNING pointerReturnValue
            DISPLAY generalBuffer
         END-PERFORM
      * logoff
         CALL "FmcjExecutionServiceLogoff"
            USING BY VALUE serviceValue
            RETURNING intReturnValue.
         DISPLAY "Logged off".
         CALL "FmcjExecutionServiceDeallocate"
            USING BY REFERENCE serviceValue
            RETURNING intReturnValue.
         CALL "FmcjGlobalDisconnect".
         MOVE FMC-OK TO retCode.
         GOBACK.
```

Action functions/subprograms

Action functions/subprograms are client-server calls, involving communication with an MQSeries Workflow server. As such, they require to be logged on.

Action functions/subprograms can be issued on service objects and on transient objects representing persistent ones. These objects remember the context of a user session so that a communication path to an MQSeries Workflow server can be established. As a consequence, empty objects cannot be used in order to issue action calls.

Action functions/subprograms are either synchronous requests waiting for the server's reply or functions/subprograms receiving information from an MQSeries Workflow server.

All action function/subprograms are described separately in "Part 7. Application programming interfaces" on page 213. You can find examples in "Part 8. Examples" on page 441.

Activity implementation functions/subprograms

An activity or support tool can be implemented by a program which uses the MQSeries Workflow API. In this case, the activity implementation functions/subprograms provide access to the input and output containers of the activity instance respectively work item or of the input container of the support tool. They also allow the program implementing an activity to return the updated output container to MQSeries Workflow so that navigation can continue on the basis of those values.

A program implementing an activity or support tool is usually executed under the control of an MQSeries Workflow program execution server on request from some MQSeries Workflow execution server. When an MQSeries Workflow execution server receives a request to start a work item or support tool, it determines the implementing program to be started and sends an appropriate request together with the input and output containers, if needed, to the logged-on user's MQSeries Workflow program execution server. Since containers are sent to the program execution agent, input and output containers are requested from and returned to an MQSeries Workflow program execution agent by the implementing program. You do *not* have to create a service object and log on to an MQSeries Workflow execution server to handle containers from within an activity implementation.

However, if you want to access not only containers, for example, if you want to query information about the process instance the work item is a part of, you have to log on to the MQSeries Workflow execution server that requested to start your program. You can use the FmcjExecutionService::Passthrough() function/subprogram to begin a session with the execution server from within the activity implementation or support tool program. This way, you can use the environment of the work item or support tool, that is, you do not need any other user ID, password, system group, or system information.

An MQSeries Workflow program execution agent can run more than one program at a time. When a container is requested, it determines the calling program and provides the container sent by the server for this program's usage.

If the activity implementation does not handle all work by itself but distributes work by starting subprograms that run as separate operating system processes, and when those subprograms request containers, then the CICS COMMAREA/IMS I/O Area must be supplied to those subprograms.

See “Chapter 51. An activity implementation” on page 465 for activity implementation examples.

Part 3. OS/390 specific considerations

Chapter 13. Cobol specific considerations

There are two ways to use the Cobol API:

1. directly through the LE ″CALL″ mechanism.
2. by using the FMCPERF copybook and the COBOL ″PERFORM″ mechanism.

The Cobol language signatures and examples given in this document are providing the information to use the API in either way.

API Call

The Cobol API uses LE ILCs to call the C API

Therefore the compiler option PGMNAME(LM) has to be used to allow

a) calls to functions with names no longer than 30 characters

b) case sensitivity in function names

String handling

Since C strings (char *) are null-terminated strings the Cobol programmer must provide string parameters null-terminated. String output parameters have to be checked for the first occurrence of X′00′ to get the correct value. A function cannot be called with String/PIC X(n) constants but must use a pointer referencing such a PIC X(n) which must be null-terminated.

Function Calls

Cobol Strings (PIC X(n)) cannot be used directly as parameters for IBM MQSeries Workflow for OS/390 function calls. A POINTER to a PIC X(n) has to be provided instead.

Provided Copybooks

Table 4. Provided Copybooks

Copybook	Contents
FMCCONST	Constants
FMCRCS	Return Codes
FMCPERF	Subprograms of full API
FMCPERFL	Subprograms of lightweight container API
FMCVARS	Variables

Mapping C to Cobol data types

Table 5 on page 82 shows how to map C to Cobol data types:

Table 5. Mapping C to Cobol data types

type in C	type in Cobol	BY VALUE / BY REFERENCE
XxxHandle	01 ptr USAGE IS POINTER. (pointing to a C++ object)	BY VALUE
XxxHandle *	01 ptr USAGE IS POINTER. (pointing to a pointer to a C++ object)	BY REFERENCE
char *, char const *	01 ptr USAGE IS POINTER. (pointing to a PIC X(n))	BY VALUE
FmcjCorrelID *	01 ptr USAGE IS POINTER. (pointing to a PIC X(24))	BY VALUE
FmcjBinary *	01 ptr USAGE IS POINTER.	BY VALUE
FmcjCDateTime const *	01 FmcjCDateTime. 05 the-year PIC 9(4) BINARY. 05 the-month PIC 9(4) BINARY. 05 the-day PIC 9(4) BINARY. 05 the-hour PIC 9(4) BINARY. 05 the-minute PIC 9(4) BINARY. 05 the-second PIC 9(4) BINARY.	BY REFERENCE
int, long, signed long, enum Xxx	01 int PIC S9(9) BINARY.	BY VALUE
APIRET	01 int PIC S9(9) BINARY.	n/a (not used as parameters)
unsigned long	01 ulong PIC 9(9) BINARY.	BY VALUE
unsigned short	01 ushort PIC 9(4) BINARY.	BY VALUE
double	01 double COMP-2.	BY VALUE
bool (1=true 0=false)	01 bool PIC 9 BINARY.	BY VALUE
long *	01 int PIC S9(9) BINARY.	BY REFERENCE
double *	01 double COMP-2.	BY REFERENCE
unsigned long const *	01 ulong PIC 9(9) BINARY.	BY VALUE

Name changes between Cobol and C

Some of the functions are declared as

```
#define functionA functionB
```

In these cases, if you want to call functionA directly via ″CALL″ in Cobol, you have to use functionB instead. All functions belonging to this category are listed in the table below.

Table 6. Function Name Mapping

FunctionA	FunctionB
FmcjActivityInstanceNotificationCategory	FmcjItemCategory

Table 6. Function Name Mapping (continued)

FunctionA	FunctionB
FmcjActivityInstanceNotificationCategoryIsNull	FmcjItemCategoryIsNull
FmcjActivityInstanceNotificationCreationTime	FmcjItemCreationTime
FmcjActivityInstanceNotificationDelete	FmcjItemDelete
FmcjActivityInstanceNotificationDescription	FmcjItemDescription
FmcjActivityInstanceNotificationDescriptionIsNull	FmcjItemDescriptionIsNull
FmcjActivityInstanceNotificationDocumentation	FmcjItemDocumentation
FmcjActivityInstanceNotificationDocumentationIsNull	FmcjItemDocumentationIsNull
FmcjActivityInstanceNotificationEndTime	FmcjItemEndTime
FmcjActivityInstanceNotificationEndTimeIsNull	FmcjItemEndTimeIsNull
FmcjActivityInstanceNotificationEqual	FmcjItemEqual
FmcjActivityInstanceNotificationIcon	FmcjItemIcon
FmcjActivityInstanceNotificationInContainerName	FmcjItemInContainerName
FmcjActivityInstanceNotificationIsComplete	FmcjItemIsComplete
FmcjActivityInstanceNotificationKind	FmcjItemKind
FmcjActivityInstanceNotificationLastModificationTime	FmcjItemLastModificationTime
FmcjActivityInstanceNotificationName	FmcjItemName
FmcjActivityInstanceNotificationObtainProcessInstanceMonitor	FmcjItemObtainProcessInstanceMonitor
FmcjActivityInstanceNotificationOutContainerName	FmcjItemOutContainerName
FmcjActivityInstanceNotificationOwner	FmcjItemOwner
FmcjActivityInstanceNotificationPersistentOid	FmcjItemPersistentOid
FmcjActivityInstanceNotificationProcessAdmin	FmcjItemProcessAdmin
FmcjActivityInstanceNotificationProcessInstance	FmcjItemProcessInstance
FmcjActivityInstanceNotificationProcessInstanceName	FmcjItemProcessInstanceName
FmcjActivityInstanceNotificationProcessInstanceState	FmcjItemProcessInstanceState
FmcjActivityInstanceNotificationProcessInstanceSystemGroupName	FmcjItemProcessInstanceSystemGroupName
FmcjActivityInstanceNotificationProcessInstanceSystemName	FmcjItemProcessInstanceSystemName
FmcjActivityInstanceNotificationReceivedAs	FmcjItemReceivedAs
FmcjActivityInstanceNotificationReceivedTime	FmcjItemReceivedTime
FmcjActivityInstanceNotificationRefresh	FmcjItemRefresh
FmcjActivityInstanceNotificationSetDescription	FmcjItemSetDescription
FmcjActivityInstanceNotificationSetName	FmcjItemSetName
FmcjActivityInstanceNotificationStartTime	FmcjItemStartTime
FmcjActivityInstanceNotificationStartTimeIsNull	FmcjItemStartTimeIsNull
FmcjActivityInstanceNotificationTransfer	FmcjItemTransfer
FmcjActivityInstanceNotificationUpdate	FmcjItemUpdate
FmcjExecutionServiceIsLoggedOn	FmcjServiceIsLoggedOn
FmcjExecutionServiceSetPassword	FmcjServiceSetPassword
FmcjExecutionServiceSetTimeout	FmcjServiceSetTimeout
FmcjExecutionServiceSystemGroupName	FmcjServiceSystemGroupName
FmcjExecutionServiceSystemName	FmcjServiceSystemName

Table 6. Function Name Mapping (continued)

FunctionA	FunctionB
FmcjExecutionServiceTimeout	FmcjServiceTimeout
FmcjExecutionServiceUserID	FmcjServiceUserID
FmcjExecutionServiceUserSettings	FmcjServiceUserSettings
FmcjProcessInstanceListDelete	FmcjPersistentListDelete
FmcjProcessInstanceListDescription	FmcjPersistentListDescription
FmcjProcessInstanceListDescriptionIsNull	FmcjPersistentListDescriptionIsNull
FmcjProcessInstanceListFilter	FmcjPersistentListFilter
FmcjProcessInstanceListFilterIsNull	FmcjPersistentListFilterIsNull
FmcjProcessInstanceListName	FmcjPersistentListName
FmcjProcessInstanceListOwnerOfList	FmcjPersistentListOwnerOfList
FmcjProcessInstanceListOwnerOfListIsNull	FmcjPersistentListOwnerOfListIsNull
FmcjProcessInstanceListRefresh	FmcjPersistentListRefresh
FmcjProcessInstanceListSetDescription	FmcjPersistentListSetDescription
FmcjProcessInstanceListSetFilter	FmcjPersistentListSetFilter
FmcjProcessInstanceListSetSortCriteria	FmcjPersistentListSetSortCriteria
FmcjProcessInstanceListSetThreshold	FmcjPersistentListSetThreshold
FmcjProcessInstanceListSortCriteria	FmcjPersistentListSortCriteria
FmcjProcessInstanceListSortCriteriaIsNull	FmcjPersistentListSortCriteriaIsNull
FmcjProcessInstanceListThreshold	FmcjPersistentListThreshold
FmcjProcessInstanceListThresholdIsNull	FmcjPersistentListThresholdIsNull
FmcjProcessInstanceListType	FmcjPersistentListType
FmcjProcessInstanceNotificationCategory	FmcjItemCategory
FmcjProcessInstanceNotificationCategoryIsNull	FmcjItemCategoryIsNull
FmcjProcessInstanceNotificationCreationTime	FmcjItemCreationTime
FmcjProcessInstanceNotificationDelete	FmcjItemDelete
FmcjProcessInstanceNotificationDescription	FmcjItemDescription
FmcjProcessInstanceNotificationDescriptionIsNull	FmcjItemDescriptionIsNull
FmcjProcessInstanceNotificationDocumentation	FmcjItemDocumentation
FmcjProcessInstanceNotificationDocumentationIsNull	FmcjItemDocumentationIsNull
FmcjProcessInstanceNotificationEndTime	FmcjItemEndTime
FmcjProcessInstanceNotificationEndTimeIsNull	FmcjItemEndTimeIsNull
FmcjProcessInstanceNotificationEqual	FmcjItemEqual
FmcjProcessInstanceNotificationIcon	FmcjItemIcon
FmcjProcessInstanceNotificationInContainerName	FmcjItemInContainerName
FmcjProcessInstanceNotificationIsComplete	FmcjItemIsComplete
FmcjProcessInstanceNotificationIsManagedByRemoteSystem	FmcjItemIsManagedByRemoteSystem
FmcjProcessInstanceNotificationKind	FmcjItemKind
FmcjProcessInstanceNotificationLastModificationTime	FmcjItemLastModificationTime
FmcjProcessInstanceNotificationName	FmcjItemName
FmcjProcessInstanceNotificationOutContainerName	FmcjItemOutContainerName

Table 6. Function Name Mapping (continued)

FunctionA	FunctionB
FmcjProcessInstanceNotificationObtainProcessInstanceMonitor	FmcjItemObtainProcessInstanceMonitor
FmcjProcessInstanceNotificationOwner	FmcjItemOwner
FmcjProcessInstanceNotificationPersistentOid	FmcjItemPersistentOid
FmcjProcessInstanceNotificationProcessAdmin	FmcjItemProcessAdmin
FmcjProcessInstanceNotificationProcessInstance	FmcjItemProcessInstance
FmcjProcessInstanceNotificationProcessInstanceName	FmcjItemProcessInstanceName
FmcjProcessInstanceNotificationProcessInstanceState	FmcjItemProcessInstanceState
FmcjProcessInstanceNotificationProcessInstanceSystemGroupName	FmcjItemProcessInstanceSystemGroupName
FmcjProcessInstanceNotificationProcessInstanceSystemName	FmcjItemProcessInstanceSystemName
FmcjProcessInstanceNotificationReceivedAs	FmcjItemReceivedAs
FmcjProcessInstanceNotificationReceivedTime	FmcjItemReceivedTime
FmcjProcessInstanceNotificationRefresh	FmcjItemRefresh
FmcjProcessInstanceNotificationSetDescription	FmcjItemSetDescription
FmcjProcessInstanceNotificationSetName	FmcjItemSetName
FmcjProcessInstanceNotificationSuspensionTime	FmcjItemSuspensionTime
FmcjProcessInstanceNotificationSuspensionTimeIsNull	FmcjItemSuspensionTimeIsNull
FmcjProcessInstanceNotificationStartTime	FmcjItemStartTime
FmcjProcessInstanceNotificationStartTimeIsNull	FmcjItemStartTimeIsNull
FmcjProcessInstanceNotificationTransfer	FmcjItemTransfer
FmcjProcessInstanceNotificationUpdate	FmcjItemUpdate
FmcjProcessTemplateListDelete	FmcjPersistentListDelete
FmcjProcessTemplateListDescription	FmcjPersistentListDescription
FmcjProcessTemplateListDescriptionIsNull	FmcjPersistentListDescriptionIsNull
FmcjProcessTemplateListFilter	FmcjPersistentListFilter
FmcjProcessTemplateListFilterIsNull	FmcjPersistentListFilterIsNull
FmcjProcessTemplateListName	FmcjPersistentListName
FmcjProcessTemplateListOwnerOfList	FmcjPersistentListOwnerOfList
FmcjProcessTemplateListOwnerOfListIsNull	FmcjPersistentListOwnerOfListIsNull
FmcjProcessTemplateListRefresh	FmcjPersistentListRefresh
FmcjProcessTemplateListSetDescription	FmcjPersistentListSetDescription
FmcjProcessTemplateListSetFilter	FmcjPersistentListSetFilter
FmcjProcessTemplateListSetSortCriteria	FmcjPersistentListSetSortCriteria
FmcjProcessTemplateListSetThreshold	FmcjPersistentListSetThreshold
FmcjProcessTemplateListSortCriteria	FmcjPersistentListSortCriteria
FmcjProcessTemplateListSortCriteriaIsNull	FmcjPersistentListSortCriteriaIsNull
FmcjProcessTemplateListThreshold	FmcjPersistentListThreshold
FmcjProcessTemplateListThresholdIsNull	FmcjPersistentListThresholdIsNull
FmcjProcessTemplateListType	FmcjPersistentListType
FmcjReadOnlyContainerAllLeafCount	FmcjContainerAllLeafCount
FmcjReadOnlyContainerAllLeaves	FmcjContainerAllLeaves

Table 6. Function Name Mapping (continued)

FunctionA	FunctionB
FmcjReadOnlyContainerArrayBinaryLength	FmcjContainerArrayBinaryLength
FmcjReadOnlyContainerArrayBinaryValue	FmcjContainerArrayBinaryValue
FmcjReadOnlyContainerArrayFloatValue	FmcjContainerArrayFloatValue
FmcjReadOnlyContainerArrayLongValue	FmcjContainerArrayLongValue
FmcjReadOnlyContainerArrayStringLength	FmcjContainerArrayStringLength
FmcjReadOnlyContainerArrayStringValue	FmcjContainerArrayStringValue
FmcjReadOnlyContainerBinaryLength	FmcjContainerBinaryLength
FmcjReadOnlyContainerBinaryValue	FmcjContainerBinaryValue
FmcjReadOnlyContainerFloatValue	FmcjContainerFloatValue
FmcjReadOnlyContainerGetElement	FmcjContainerGetElement
FmcjReadOnlyContainerLeafCount	FmcjContainerLeafCount
FmcjReadOnlyContainerLeaves	FmcjContainerLeaves
FmcjReadOnlyContainerLongValue	FmcjContainerLongValue
FmcjReadOnlyContainerMemberCount	FmcjContainerMemberCount
FmcjReadOnlyContainerStringLength	FmcjContainerStringLength
FmcjReadOnlyContainerStringValue	FmcjContainerStringValue
FmcjReadOnlyContainerStructMembers	FmcjContainerStructMembers
FmcjReadOnlyContainerType	FmcjContainerType
FmcjReadWriteContainerAllLeafCount	FmcjContainerAllLeafCount
FmcjReadWriteContainerAllLeaves	FmcjContainerAllLeaves
FmcjReadWriteContainerArrayBinaryLength	FmcjContainerArrayBinaryLength
FmcjReadWriteContainerArrayBinaryValue	FmcjContainerArrayBinaryValue
FmcjReadWriteContainerArrayFloatValue	FmcjContainerArrayFloatValue
FmcjReadWriteContainerArrayLongValue	FmcjContainerArrayLongValue
FmcjReadWriteContainerArrayStringLength	FmcjContainerArrayStringLength
FmcjReadWriteContainerArrayStringValue	FmcjContainerArrayStringValue
FmcjReadWriteContainerBinaryLength	FmcjContainerBinaryLength
FmcjReadWriteContainerBinaryValue	FmcjContainerBinaryValue
FmcjReadWriteContainerFloatValue	FmcjContainerFloatValue
FmcjReadWriteContainerGetElement	FmcjContainerGetElement
FmcjReadWriteContainerLeafCount	FmcjContainerLeafCount
FmcjReadWriteContainerLeaves	FmcjContainerLeaves
FmcjReadWriteContainerLongValue	FmcjContainerLongValue
FmcjReadWriteContainerMemberCount	FmcjContainerMemberCount
FmcjReadWriteContainerStringLength	FmcjContainerStringLength
FmcjReadWriteContainerStringValue	FmcjContainerStringValue
FmcjReadWriteContainerStructMembers	FmcjContainerStructMembers
FmcjReadWriteContainerType	FmcjContainerType
FmcjWorkitemCategory	FmcjItemCategory
FmcjWorkitemCategoryIsNull	FmcjItemCategoryIsNull

Table 6. Function Name Mapping (continued)

FunctionA	FunctionB
FmcjWorkitemCreationTime	FmcjItemCreationTime
FmcjWorkitemDelete	FmcjItemDelete
FmcjWorkitemDescription	FmcjItemDescription
FmcjWorkitemDescriptionIsNull	FmcjItemDescriptionIsNull
FmcjWorkitemDocumentation	FmcjItemDocumentation
FmcjWorkitemDocumentationIsNull	FmcjItemDocumentationIsNull
FmcjWorkitemEndTime	FmcjItemEndTime
FmcjWorkitemEndTimeIsNull	FmcjItemEndTimeIsNull
FmcjWorkitemEqual	FmcjItemEqual
FmcjWorkitemIcon	FmcjItemIcon
FmcjWorkitemInContainerName	FmcjItemInContainerName
FmcjWorkitemIsComplete	FmcjItemIsComplete
FmcjWorkitemKind	FmcjItemKind
FmcjWorkitemLastModificationTime	FmcjItemLastModificationTime
FmcjWorkitemName	FmcjItemName
FmcjWorkitemOutContainerName	FmcjItemOutContainerName
FmcjWorkitemObtainProcessInstanceMonitor	FmcjItemObtainProcessInstanceMonitor
FmcjWorkitemOwner	FmcjItemOwner
FmcjWorkitemPersistentOid	FmcjItemPersistentOid
FmcjWorkitemProcessAdmin	FmcjItemProcessAdmin
FmcjWorkitemProcessInstance	FmcjItemProcessInstance
FmcjWorkitemProcessInstanceName	FmcjItemProcessInstanceName
FmcjWorkitemProcessInstanceState	FmcjItemProcessInstanceState
FmcjWorkitemProcessInstanceSystemGroupName	FmcjItemProcessInstanceSystemGroupName
FmcjWorkitemProcessInstanceSystemName	FmcjItemProcessInstanceSystemName
FmcjWorkitemReceivedAs	FmcjItemReceivedAs
FmcjWorkitemReceivedTime	FmcjItemReceivedTime
FmcjWorkitemRefresh	FmcjItemRefresh
FmcjWorkitemSetDescription	FmcjItemSetDescription
FmcjWorkitemSetName	FmcjItemSetName
FmcjWorkitemStartTime	FmcjItemStartTime
FmcjWorkitemStartTimeIsNull	FmcjItemStartTimeIsNull
FmcjWorkitemTransfer	FmcjItemTransfer
FmcjWorkitemUpdate	FmcjItemUpdate
FmcjWorklistDelete	FmcjPersistentListDelete
FmcjWorklistDescription	FmcjPersistentListDescription
FmcjWorklistDescriptionIsNull	FmcjPersistentListDescriptionIsNull
FmcjWorklistFilter	FmcjPersistentListFilter
FmcjWorklistFilterIsNull	FmcjPersistentListFilterIsNull
FmcjWorklistName	FmcjPersistentListName

Table 6. Function Name Mapping (continued)

FunctionA	FunctionB
FmcjWorklistOwnerOfList	FmcjPersistentListOwnerOfList
FmcjWorklistOwnerOfListIsNull	FmcjPersistentListOwnerOfListIsNull
FmcjWorklistRefresh	FmcjPersistentListRefresh
FmcjWorklistSetDescription	FmcjPersistentListSetDescription
FmcjWorklistSetFilter	FmcjPersistentListSetFilter
FmcjWorklistSetSortCriteria	FmcjPersistentListSetSortCriteria
FmcjWorklistSetThreshold	FmcjPersistentListSetThreshold
FmcjWorklistSortCriteria	FmcjPersistentListSortCriteria
FmcjWorklistSortCriteriaIsNull	FmcjPersistentListSortCriteriaIsNull
FmcjWorklistThreshold	FmcjPersistentListThreshold
FmcjWorklistThresholdIsNull	FmcjPersistentListThresholdIsNull
FmcjWorklistType	FmcjPersistentListType

To cope with the Cobol restriction of 30 characters per word some class name prefixes as well as function and constant names had to be abbreviated. The abbreviations for the class name prefixes are listed in the table below.

Table 7. Abbreviation Class Prefixes

Class Name	Abbrevation
FmcjActivityInstanceList	FmcjAIL
FmcjActivityInstance	FmcjAI
FmcjActivityInstanceNotification	FmcjAIN
FmcjActivityInstanceNotificationVector	FmcjAINV
FmcjActivityInstanceVector	FmcjAIV
FmcjBlockInstanceMonitor	FmcjBIM
FmcjContainerElement	FmcjCE
FmcjContainerElementVector	FmcjCEV
FmcjContainer	FmcjC
FmcjControlConnectorInstance	FmcjCCI
FmcjControlConnectorInstanceVector	FmcjCCIV
FmcjDllOptions	FmcjDO
FmcjExeOptions	FmcjExeO
FmcjExecutionData	FmcjED
FmcjExecutionService	FmcjES
FmcjExternalOptions	FmcjExtO
FmcjImplementationData	FmcjID
FmcjImplementationDataVector	FmcjIDV
FmcjPerson	FmcjP
FmcjPoint	FmcjPnt
FmcjPointVector	FmcjPntV
FmcjProgramData	FmcjPD
FmcjProcessInstance	FmcjPI

Table 7. Abbreviation Class Prefixes (continued)

Class Name	Abbrevation
FmcjProcessInstanceList	FmcjPIL
FmcjProcessInstanceListVector	FmcjPILV
FmcjProcessInstanceMonitor	FmcjPIM
FmcjProcessInstanceNotification	FmcjPIN
FmcjProcessInstanceNotificationVector	FmcjPINV
FmcjProcessInstanceVector	FmcjPIV
FmcjPersistentList	FmcjPL
FmcjProcessTemplateList	FmcjPTL
FmcjProcessTemplateListVector	FmcjPTLV
FmcjProcessTemplate	FmcjPT
FmcjProcessTemplateVector	FmcjPTV
FmcjReadOnlyContainer	FmcjROC
FmcjReadWriteContainer	FmcjRWC
FmcjStringVector	FmcjStrV
FmcjSymbolLayout	FmcjSL
FmcjWorkitem	FmcjWI
FmcjWorkitemVector	FmcjWIV
FmcjWorklist	FmcjWL

The abbreviations for function and constant names are listed in the table below (Note that these abbreviations take place after the class name prefix abbreviations). Instead of constructing the names with the help of this table you can also search the FMCPERF copybook for the C function name to get the corresponding Cobol function and variable names.

Table 8. Abbreviations

Original String	Abbrevation
Activity	Act
ACTIVITY	ACT
Administration	Admin
Administration	Admin
ADMINSTRATION	ADMIN
Already	Alr
ALREADY	ALR
Authorization	Auth
AUTHORIZATION	AUTH
Authorized	Auth
AUTHORIZED	AUTH
Backward	Backw
BACKWARD	BACKW
Categories	Categs
CATEGORIES	CATEGS

Table 8. Abbreviations (continued)

Original String	Abbrevation
Category	Categ
CATEGORY	CATEG
CHECKOUT	CHKOUT
Container	Ctnr
CONTAINER	CTNR
ControlConnector	ContrConn
CONTROLCONNECTOR	CONTRCONN
Definition	Def
DEFINITION	DEF
Directory	Dir
DIRECTORY	DIR
Executable	Exec
EXECUTABLE	EXEC
EXTERNAL	EXT
ForeGround	ForeGr
FOREGROUND	FOREGR
Forward	Forw
FORWARD	FORW
FOUND-FOR-AUTO-START	FND-FR-AUT-ST
FROM	FRM
IMPLEMENTATION	IMP
Instance	Inst
INSTANCE	INST
INVALID	INVAL
Invocation	Invoc
INVOCATION	INVOC
Location	Loc
LOCATION	LOC
Mapping	Map
MAPPING	MAP
Monitor	Mon
MONITOR	MON
Notification	Notif
NOTIFICATION	NOTIF
Notified	Notif
NOTIFIED	NOTIF
Object	Obj
OBJECT	OBJ
Organization	Org
ORGANIZATION	ORG

Table 8. Abbreviations (continued)

Original String	Abbrevation
Parameter	Parm
PARAMETER	PARM
PersistentOidOf	PersOidOf
PERSISTENTOIDOF	PERSOIDOF
Persons	Pers
PERSON	PERS
Process	Proc
PROCESS	PROC
Second	Sec
SECOND	SEC
Service	Serv
SERVICE	SERV
Started	Strtd
STARTED	STRTD
SUB-PROC	SB-PRC
Suspension	Susp
SUSPENSION	SUSP
System	Syst
SYSTEM	SYST
Template	Templ
TEMPLATE	TEMPL
Terminated	Term
TERMINATED	TERM
Transition	Trans
TRANSITION	TRANS

The C API uses some variable names that are reserved words in Cobol. These variable names were changed in the Cobol API by adding ″Value″ to the variable name. All variables belonging to this category are listed below.

```
year   month    day     hour    minute  second   data
file   function index   input   line    output
owner  password service time    type    timeout
```

Finally, the variable name ″value″ is used in the C API with different types. In Cobol this variable is renamed according to its type:

intValue, doubleValue, pointerValue

Example how to use Strings

To call a C function with the signature char* cfunc(char* x) the code looks like the following:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. "STRTEST".
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 PTR1 USAGE IS POINTER VALUE NULL.
  01 PTR2 USAGE IS POINTER VALUE NULL.
  01 STRING-STRUCT1.
     05 X PIC X(20).
  01 STRING-STRUCT2.
     05 Y PIC X(20).
  01 STRLEN PIC 99 VALUE 0.
PROCEDURE DIVISION.
    MOVE z"Initial String" TO X.
    CALL "SETADDR" USING STRING-STRUCT1 PTR1.
    CALL "cfunc" USING BY VALUE PTR1
                 RETURNING PTR2.
    CALL "GETADDR" USING STRING-STRUCT2 PTR2.
    INSPECT Y TALLYING STRLEN FOR CHARACTERS
        BEFORE INITIAL X"00".
    DISPLAY "Y is " Y(1:STRLEN).
    STOP RUN.
END PROGRAM "STRTEST".

IDENTIFICATION DIVISION.
PROGRAM-ID. "SETADDR".
DATA DIVISION.
  LINKAGE SECTION.
  01 PTR3 USAGE IS POINTER.
  01 STRING-STRUCT3.
     05 Z PIC X(20).
PROCEDURE DIVISION USING BY REFERENCE STRING-STRUCT3 PTR3.
    SET PTR3 TO ADDRESS OF Z.
    GOBACK.
END PROGRAM "SETADDR".

IDENTIFICATION DIVISION.
PROGRAM-ID. "GETADDR".
DATA DIVISION.
  LINKAGE SECTION.
  01 PTR4 USAGE IS POINTER.
  01 STRING-STRUCT4.
     05 Z PIC X(20).
  01 DUMMY-STRUCT.
     05 W PIC X(20).
PROCEDURE DIVISION USING BY REFERENCE STRING-STRUCT4 PTR4.
    SET ADDRESS OF DUMMY-STRUCT TO PTR4.
    MOVE W TO Z.
    GOBACK.
END PROGRAM "GETADDR".
```

Chapter 14. CICS specific considerations

In CICS Activity implementations and Input-/OutputContainer data are not retrieved from the PEA as in the LAN version but are sent with the invocation and stored in the COMMAREA. If the user changes the COMMAREA without using the IBM MQSeries Workflow for OS/390 API or once the OutputContainer has been set this data cannot be retrieved anymore.

Information on how to enable CICS to work with IBM MQSeries Workflow for OS/390 see *Customization & Administration*.

Chapter 15. IMS specific considerations

In IMS Activity implementations, Input-/OutputContainer data and the Passthrough ticket are not retrieved from the PEA as in the LAN version but are sent with the invocation and stored in the I/O AREA. If the user changes the I/O AREA without using the IBM MQSeries Workflow for OS/390 API or once the OutputContainer has been set this data cannot be retrieved anymore.

IMS only supports the Container API.

Information on how to enable IMS to work with IBM MQSeries Workflow for OS/390 see *Customization & Administration*.

Part 4. Program Execution Server’s Program Mapping

Chapter 16. Introduction

Whenever legacy applications should be invoked by MQSeries Workflow a mapping of the MQSeries container data into a format acceptable by the legacy application is needed.

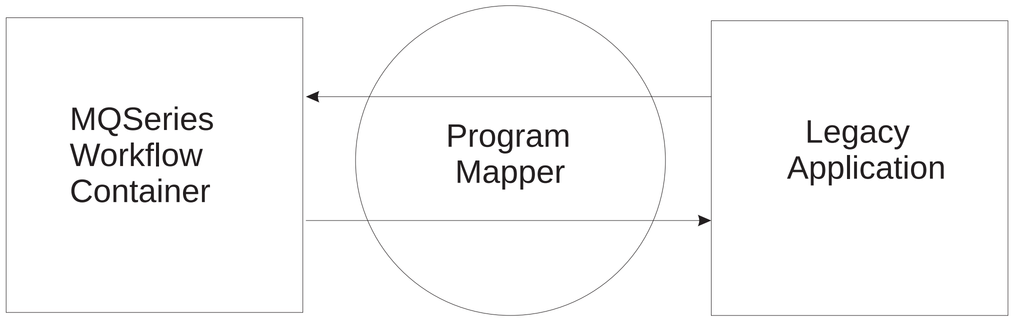


Figure 3. Program mapping illustration.

MQSeries Workflow offers a default program mapper with basic functionality. This chapter gives a quick overview about the program execution server's (PES) program mapping component which does the mapping of MQSeries Workflow containers into a format acceptable by legacy applications. This is a basic mapper so that legacy applications like IMS and CICS are supported. If there are more complicate mappings to be done other mapping tools can be used.

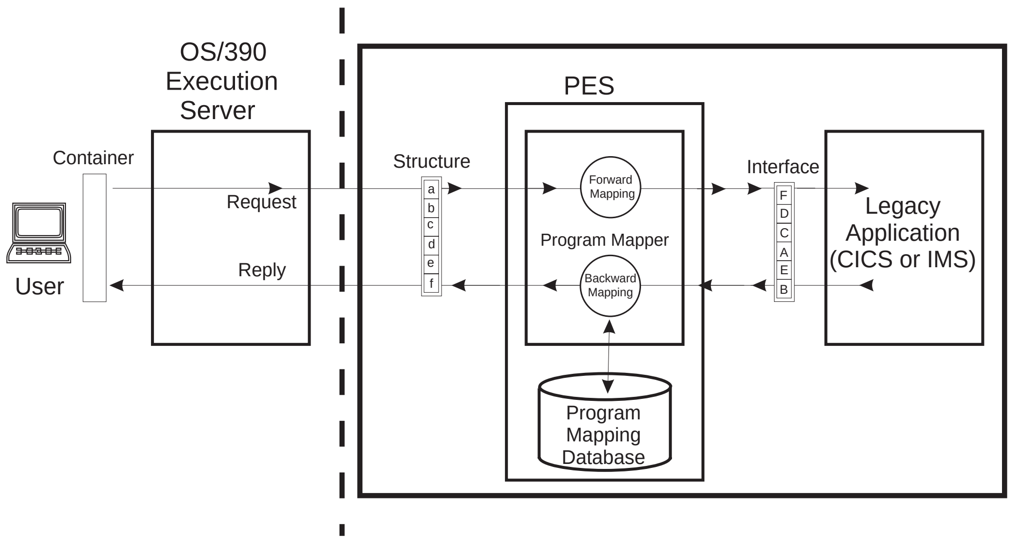


Figure 4. LAN - OS/390 structure.

In order to make the format of workflow containers acceptable by legacy applications the content of the workflow containers is put to an interface called **structure**. The input/output interface for the legacy application is called **interface**. The task for the program mapper is to convert the data between the structure and interface. Mapping from MQSeries Workflow to the legacy application (a to A, b to B...) is called **forwardmapping** and mapping from legacy applications to MQSeries

Workflow (A to a, B to b...) **backwardmapping**. If special conversion between structure and interface elements is needed a usertype exit (which will be explained later) can be used.

Mapping is not necessary if the called application is using Workflow API's to extract data from the containers.

The way the mapping should be done between structures and interfaces is defined with a **mapping definition language (MDL)**.

To create a mapping you will have to write the definition of the structure and interface elements first. Then you connect these structures and interfaces with forward/backwardmapping program mapping definitions, compile it with a parser and load it into the mapping database. The elements of the mapper are explained in detail in the *MQSeries Workflow for OS/390: Customization & Administration*. The following graphic illustrates the process:

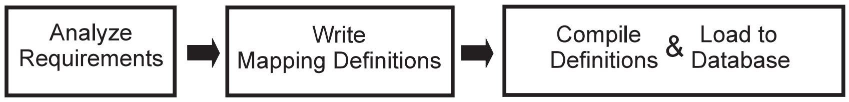

Analyze Requirements
Write Mapping Definitions
Compile Definitions & Load to Database

Figure 5. How to create a program mapping.

Chapter 17. Program mapping definitions

In this chapter the mapping definitions will be explained more detailed. For every definition a simple example is given additionally.

Structure Definition

A structure defines the MQSeries Workflow container structure which is passed into the program execution server (PES). The structure definition syntax is identical to the structure definition syntax used in the Flowmark definition language (FDL). This allows to export container definitions from Buildtime into a flat file and copy these structure definitions into the mapping definition language (MDL). A structure mainly consists of a collection of members (structure elements) who have a type and cardinality.

Example: This example shows a container in MQSeries Workflow representing an account representative structure. May there be the name from the holder of the account (first name and last name defined as a string), the corresponding zip (zip = postal code, defined as long), salary and tax. The last part of the container shall be filled with the data of some customers belonging to the holder of the account. The example for the definition of the CustomerStructure is given later. In order to define the structure you have to define every single element of the MQSeries Workflow container which shall be passed to the legacy application (The code for an example legacy application is given in “Chapter 25. Additional Mapping Examples” on page 139).

```
STRUCTURE AccountRepStructure
          LastName:   STRING;
          FirstName:  STRING;
          Zip:        LONG;
          Salary:     FLOAT;
          Tax:        FLOAT;
          Customers:  CustomerStructure(3);
END AccountRepStructure
```

You will find a more detailed example “Simple datastructure with default name mapping” on page 142 and the structure definition grammar “Structure definition” on page 118.

Interface Definition

An interface defines the layout and type of the data accepted by a legacy application. Every interface element has a fixed size and place (offset) and will be filled with converted container elements. There is no way to verify whether the size place and type of the elements actually matches the size place and type expected by the legacy application. This means the interface definitions have to be created carefully. Otherwise conversion results are unpredictable and also mapping runtime errors may occur because of invalid data. Every element of an interface is mapped to a structure element with the same name. If there is no element with the same name the interface element is skipped and the container element is untouched. It is also possible to define a constant for an interface element. See “Constants” on page 107 for more details.

Example: This example shows an interface of a legacy application representing an account representative structure. In this case the name from the holder of the account (first name and last name defined as a string with a maximum of 50 characters, terminated by hex zero and justified left with a pad char ″ ″), the corresponding zip (defined as an unsigned integer with 16 bits), salary and tax (defined as float number with 32 bits). The last part of the container shall be filled with the data of some customers belonging to the holder of the account. The example for the definition of the CustomerStructure (Array for 3 customers using another structure ″CustomerInterfaceForCpp″) is given later. In order to define the interface you have to define every single element of the container used by the legacy application. The definition of the interface should read as follows:

```
INTERFACE AccountRepInterfaceForCpp
         LastName:   CHAR(50) TERMINATEDBY "<H00>" JUSTIFY LEFT PAD " ";
         FirstName:  CHAR(50) TERMINATEDBY "<H00>" JUSTIFY LEFT PAD " ";
         Zip:        UNSIGNED INTEGER 16;
         Salary:     FLOAT 32;
         Tax:        FLOAT 32;
         Customers:  ARRAY(3) CustomerInterfaceForCpp;
END AccountRepInterfaceForCpp
```

You will find a more detailed example “Simple datastructure with default name mapping” on page 142 and the interface definition grammar “Interface definition” on page 119.

If you have an interface element and no corresponding structure element no mapping will be done by the default mapper. If it is required to have some constants on the legacy application side every interface element may optionally have a **constant** statement which defines the constant to create for the legacy application. The constant is converted in a forwardmapping whether there is a matching structure element or not. If a backwardmapping occurs a structure element is set to this constant only if there is an element with the same name or a mapping rule between the structure and the interface with a constant. See “Constants” on page 107 for more detailed information.

Forward/Backwardmapping Definition

The connection between a structure and an interface is done via a forwardmapping and backwardmapping definition. Forwardmapping is used to map a structure into a format accepted by a legacy application and backwardmapping is used to map legacy application data into a structure. Mapping done for structure and interface elements with identical names is called **default mapping**. In addition it is possible to do **explicit mapping** of elements which have different names via rules. Structures are mapped as a whole into interfaces and arrays are mapped as a whole if both the structure and interface array have the same size. It is not possible to map array elements individually. If more powerful mappings are required use container mapping (See the *Buildtime* manual).

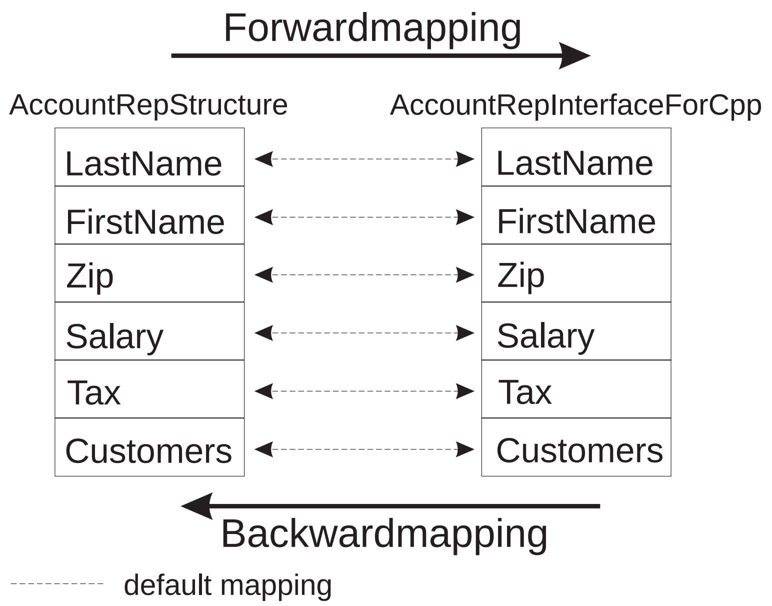


Figure 6. Default Forward/Backwardmapping

To create a forwardmapping you have to define which structure shall be mapped to which interface. To create a backwardmapping you have to define which interface shall be mapped to which structure. Optional you can use rules to map elements with different names. See "Chapter 18. Mapping algorithm" on page 105 for more detailed information about default and explicit mapping.

The coding for the mapping according to the diagram would follow as:

```
/* Mapping from MQSeries Workflow (structure) to legacy appl.(interface) */

FORWARDMAPPING Forward
        FROM AccountRepStructure TO AccountRepInterfaceForCpp
END

/* Mapping from legacy appl. (interface) to MQSeries Workflow (structure) */

BACKWARDMAPPING Backward
        FROM AccountRepInterfaceForCpp TO AccountRepStructure
END
```

Note: All structure and interface elements are mapped because they have identical names (default mapping).

You will find one complete mapping in "Example" on page 109.

Usertype Definition

A usertype can be used by the program mapper whenever the interface types offered by a default mapper do not offer the required conversion. In this case the actual data conversion has to be done by a usertype exit which has to reside in a DLL. See chapter "Chapter 21. Usertype" on page 129.

Example: In order to assign a number to a currency with the corresponding symbol you need to define a usertype (every mapping type would map the number to the exact number in a different format but not assign it to a special currency). It is also possible to define a usertype which calculates a value of currency A used in the structure to a currency B used in the interface (for example US dollars to British pound or the new European currency, the Euro).

```
USERTYPE SampleUsertype LENGTH(4)
        DLL "SAMPUTY","SampleUsertypeExit"
END

INTERFACE SampleUsertypeInterface
         DESCRIPTION "Sample Usertype Interface"
         SampleElement: USERTYPE SampleUsertype PARMS "$";
END
```

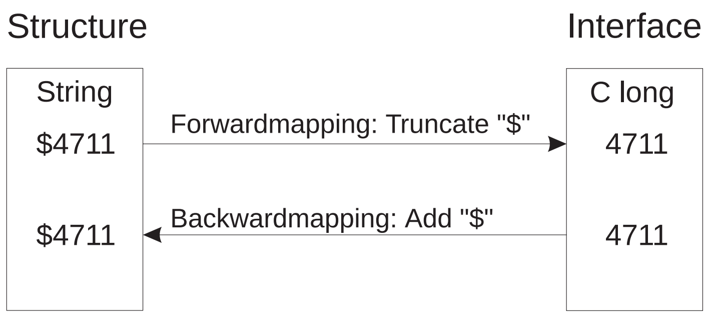


Figure 7. UserType example

This example shows the functionality of a usertype. An interface element (defined as C long) is mapped to a structure element (defined as a string). In backwardmapping the C long ″4711″ is converted to a string and prefixed with ″$″. In forwardmapping the string ″$4711″ is truncated to ″4711″ and then converted to a C long.

Chapter 18. Mapping algorithm

All elements in the MQSeries Workflow container have names. The interface elements also have to have names. Mapping is done per default on name by name base if elements have the same name. If element names are different mapping rules can be used to do explicit mapping.

Structures are mapped as a whole to interfaces if there names are identical. Arrays are also mapped as a whole. It is also possible to define constants which are inserted on the legacy application or container side.

If structure or interface elements are not mapped the data in the structure element or interface element is not modified.

Note: Every structure element can only be mapped to one interface element and vice versa.

In this example you have 4 structure and interface elements which should be mapped by the default mapper.

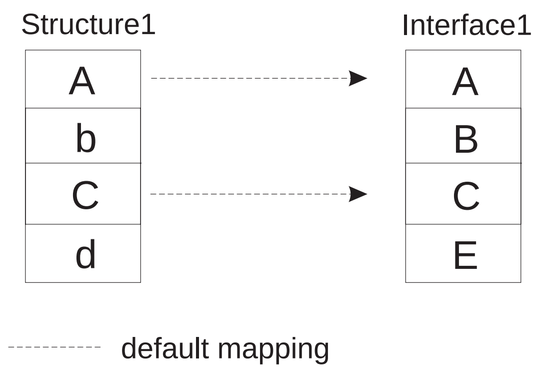


Figure 8. Default Forwardmapping illustration.

```
FORWARDMAPPING Forward
    FROM Structure1 TO Interface1
END
```

The default mapper maps structure element A to interface element A and structure element C to interface element C. It does **not** map structure element b to interface element B or structure element d to interface element E, because of the different names. See also “Simple datastructure with default name mapping” on page 142 for a more detailed example.

If the corresponding structure and interface elements do not have identical names the mapping has to be defined explicitly. In this case you have to define an additional mapping rule for the mapping. The mapping definition language (MDL) and the corresponding grammar is explained in “Chapter 20. Grammar” on page 115. The following graphic displays a simple forwardmapping with some non-identical names of the structure and interface elements.

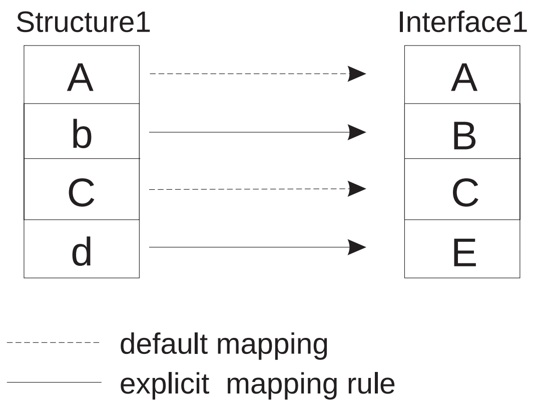


Figure 9. Non-Default Forwardmapping Forward2 illustration.

The mapping rules would follow as:

```
FORWARDMAPPING Forward2 FROM Structure1 TO Interface1
        MAP b TO B;
        MAP d TO E;
END
```

Structure elements A and C do not have to be mapped to interface elements A and C explicitly, this will be done be the default mapper automatically. Refer to "Complex datastructure with non default name mapping" on page 143 for a more detailed example.

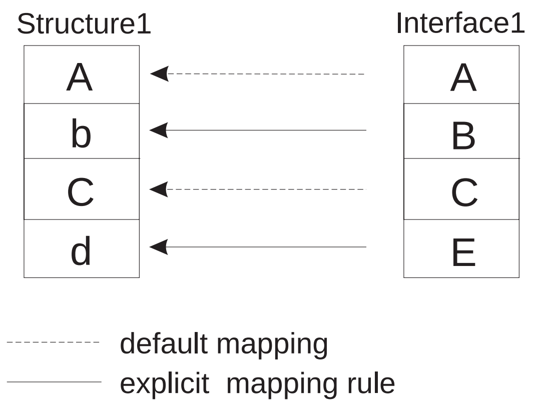


Figure 10. Non-Default Backwardmapping Backward1 illustration.

If you would like a backwardmapping according to the above diagram the mapping rules would be:

```
BACKWARDMAPPING Backward1 FROM Interface1 TO Structure1
        MAP B TO b;
        MAP E TO d;
END
```

How you map elements to each other only depends on the definition as long as you do not violate some conversion rules (See "Valid conversions between MQSeries Workflow container program mapping element types and program mapping interface types" on page 112). For example it is not allowed to map an integer to a binary.

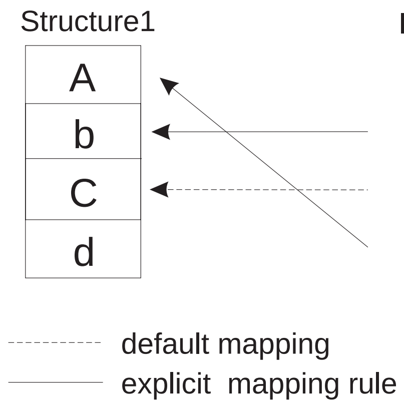


Figure 11. Explicit mapping Backward2 illustration.

In this case the interface element A will not be mapped to structure element A because there is a rule from interface element E to structure element A. Interface element B is mapped because of the explicit rule. Interface element C is mapped to structure element C because they have the same name (default mapping) and no explicit mapping for interface element C is defined. Interface element E will be mapped to structure element A because of a mapping rule for interface element E.

```
BACKWARDMAPPING Backward2 FROM Interface1 TO Structure1
         MAP B TO b;
         MAP E TO A;
END
```

Table 9. Rule mapping with no constant definition

	BACKWARDMAPPING	**FORWARDMAPPING**
There exists a definition rule for forward/ backwardmapping	Map interface element to structure element	Map structure element to interface element
There exists **no** definition rule for forward/ backwardmapping	Interface element not mapped to structure element	Interface element undefined

Note:

- If the mapping rules use invalid or non existent interface element names these rules are ignored during actual mapping. Make sure you use the right names in forwardmapping and backwardmapping. By contrast structure element names used in definition rules have to exist. Otherwise runtime errors occur.
- Do not map structure elements to interface elements which are used in other arrays. The structure element will contain the interface elements with the largest dimension.

Constants

If it is required to have some constants on the legacy application or structure side every **interface** element may optionally have a constant statement which defines the constant to create for the legacy application or structure element. The constant is converted in a forwardmapping whether there is a matching structure element or not. If a backwardmapping occurs the structure element is set to this constant only if there is an element with the same name or a rule is defined for this structure element.

Table 10. Mapping with constant definition

	Backwardmapping	**Forwardmapping**
There exists a definition rule for forward/ backwardmapping	Structure element set to constant	Interface element set to constant
There exists **no** definition rule for forward/ backwardmapping	Structure element **not** set to constant	Interface element set to constant

Example for non-default forwardmapping with constant definitions:

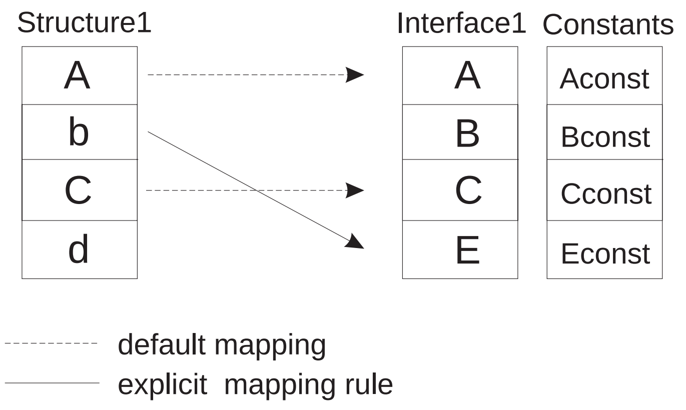


Figure 12. Forwardmapping with constants.

Because structure elements A, C and b are mapped to interface elements A, C and E (who all have a constant definition) the interface elements A,C and E are set to Aconst, Cconst and Econst respectively. The interface element B is set to Bconst because no structure element was mapped to B. So all interface elements are set to their corresponding constant values. In forwardmapping all interface elements with a constant definition are set to their constant regardless if there is a mapping to this element or not. Only if the interface element has **no** constant definition a mapping may change the value. Refer to “Simple datastructure with all interface types with CONSTANTS and usertypes” on page 144 for a more complex example.

Example for default backwardmapping with constant definitions:

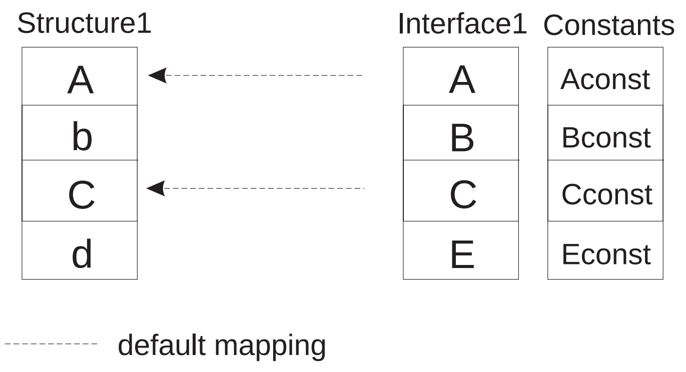


Figure 13. Backwardmapping with constants.

Because interface elements B and E are not mapped to structure elements b and d, b and d are **not** set to the constant values of Bconst and Econst. Because interface element A is mapped to structure element A and the interface element A has a constant Aconst, the structure element A is set to Aconst (same as structure element C is set to Cconst). Assuming there would be no constant definition for C on the legacy application side, interface element C would have been mapped to structure element C as usual.

Example

In this example the structure elements and interface elements do not have the same names. Therefore they have to be mapped explicitly in the forward/backwardmapping definition. If they would not be mapped explicitly no mapping would be done at all because all structure and interface elements have different names.

```
STRUCTURE AccountRepStructure
          LastName:   STRING;
          FirstName:  STRING;
          Zip:        LONG;
          Salary:     FLOAT;
          Tax:        FLOAT;
          Customers:  CustomerStructure(3);
END AccountRepStructure

INTERFACE AccountRepInterfaceForCpp
          L: CHAR(50) TERMINATEDBY "<H00>" JUSTIFY LEFT PAD " ";
          F: CHAR(50) TERMINATEDBY "<H00>" JUSTIFY LEFT PAD " ";
          Z: UNSIGNED INTEGER 16;
          S: FLOAT 32;
          T: FLOAT 32;
          C: ARRAY(3) CustomerInterfaceForCpp;
END AccountRepInterfaceForCpp

/* Mapping from MQSeries Workflow (structure) to legacy appl.(interface) */

FORWARDMAPPING Forward FROM AccountRepStructure TO AccountRepInterfaceForCpp
         MAP LastName TO L;
         MAP FirstName TO F;
         MAP Zip TO Z;
         MAP Salary TO S;
         MAP Tax TO T;
         MAP Customers TO C;
END

/* Mapping from legacy appl. (interface) to MQSeries Workflow (structure) */

BACKWARDMAPPING Backward FROM AccountRepInterfaceForCpp TO AccountRepStructure
         MAP L TO LastName;
         MAP F TO FirstName;
         MAP Z TO Zip;
         MAP S TO Salary;
         MAP T To Tax;
         MAP C TO Customers;
END
```

Chapter 19. Supported program mapping definition element types

Program mapping structure definition element types

All MQSeries Workflow element types are supported:
- LONG
- FLOAT
- STRING
- BINARY

Program mapping interface definition element types

Characters

Characters have a size in bytes, an optional termination character (that means hex 0 for C/C++) and justification and padding.

Integer Numbers

Integers have a sign or are unsigned and a size in bits. Supported sizes are 16 and 32 bits.

Float Numbers

Floats have a size in bits. Supported sizes are 32 and 64 bits.

Packed Numbers

Packed numbers have a size, are either signed with a character for plus and a character for minus or unsigned with an unsigned character and have a scale.

Zoned Numbers

Zoned numbers have a size, are either signed with a character for plus and a character for minus or unsigned with an unsigned character and have a scale.

Interface

Interfaces only have a name and define another interface used as interface element. In this way it is possible to structure the interface in the same way like structures may be defined to contain other structures.

Usertypes

Whenever the previous interface types do not match the required types it is possible to define a usertype. For more details see “Chapter 20. Grammar” on page 115. Usertypes have a name and an optional parameter string which can be used to pass additional information to the usertype exit. In order to be able to use usertype this usertype has to be defined and a usertype DLL with an usertype exit has to be provided. (See “Chapter 21. Usertype” on page 129 for more details):

Valid conversions between MQSeries Workflow container program mapping element types and program mapping interface types

The following table lists all possible combinations of structure elements and interface elements. If they are arrays they have to have the same size. There is one exception: A character of size 1 can be mapped to a MQSeries Workflow LONG. If there is an invalid combination used a runtime error will be created (see *Messages* for detailed information).

Table 11. Mapping combinations

Workflow Type Interface Type	**String**	**Binary**	**Long**	**Float**
CHAR	*	*	*	
INTEGER	*		*	*
FLOAT				*
PACKED	*		*	*
ZONED	*		*	*
USERTYPE	*1	*1	*1	*1
Note: 1 Only available if this type of combination is supported by the user type exit				

Table 12. C/C++ data type mappings (legacy application (C/C++) to FDL types (structure))

C/C++ type	**Interface**	**Structure**	**Comment**
char	CHAR(1) JUSTIFY LEFT PAD ″ ″	STRING	No imbedded x′00′
char [5]	CHAR(5) JUSTIFY LEFT PAD ″ ″ or CHAR(5) TERMINATEDBY ″<h00>″ JUSTIFY LEFT PAD ″ ″	STRING	No imbedded x′00′
char	CHAR(1) JUSTIFY LEFT PAD ″ ″	BINARY	
char [5]	CHAR(5) JUSTIFY LEFT PAD ″ ″ or CHAR(5) TERMINATEDBY ″<h00>″ JUSTIFY LEFT PAD ″<h00>″	BINARY	
char	CHAR(1) JUSTIFY LEFT PAD ″<h00>″	LONG	
short	SIGNED INTEGER 16	LONG, STRING	
unsigned short	UNSIGNED INTEGER 16	LONG, STRING	
int	SIGNED INTEGER 32	LONG, STRING	
unsigned int	UNSIGNED INTEGER 32	LONG, STRING	
long	SIGNED INTEGER 32	LONG, STRING	
unsigned long	UNSIGNED INTEGER 32	LONG, STRING	
float	FLOAT 32	FLOAT	
double	FLOAT 64	FLOAT	

Table 13. Cobol data type mappings (legacy application (COBOL) to FDL types (structure))

Cobol	Interface	Structure	Comment
PIC X(n)	CHAR(n) JUSTIFY LEFT PAD ″ ″	STRING	No imbedded x′00′
PIC X(n)	CHAR(n) JUSTIFY LEFT PAD ″<h00>″	BINARY	
PIC X(1)	CHAR(1) JUSTIFY LEFT PAD ″<h00>″	LONG	
PIC S999 PACKED-DECIMAL or COMP-3	PACKED(3) SIGNED MINUS ″<h0d>″ PLUS ″<h0c>″ SCALE 0	LONG, STRING, FLOAT	
PIC S999PP PACKED-DECIMAL or COMP-3	PACKED(3) SIGNED MINUS ″<h0d>″ PLUS ″<h0c>″ SCALE 2	LONG, STRING, FLOAT	
PIC S99V9 PACKED-DECIMAL or COMP-3	PACKED(3) SIGNED MINUS ″<h0d>″ PLUS ″<h0c>″ SCALE -1	LONG, STRING, FLOAT	
PIC S9999 BINARY or COMP-4	SIGNED INTEGER 16	LONG, STRING, FLOAT	up to 4 digits[1]
PIC S9(6) BINARY or COMP-4	SIGNED INTEGER 32	LONG, STRING, FLOAT	between 5 and 9 digits[1]
COMP-1	FLOAT 32	FLOAT	
COMP-2	FLOAT 64	FLOAT	
PIC S9(n)V9(m) DISPLAY	ZONED(n+m) SIGNED LAST MINUS ″-″ PLUS ″+″ SCALE -m	LONG, STRING, FLOAT	
PIC S9(n)V9(m) DISPLAY SIGN LEADING	ZONED(n+m) SIGNED FIRST MINUS ″-″ PLUS ″+″ SCALE -m	LONG, STRING, FLOAT	
PIC S9(n)V9(m) DISPLAY SIGN TRAILING SEPARATE	ZONED(n+m) SIGNED LAST MINUS ″-″ PLUS ″+″ SEPARATE SCALE -m	LONG, STRING, FLOAT	
PIC 9(n)V9(m) DISPLAY	ZONED(n) UNSIGNED SCALE -m	LONG, STRING, FLOAT	

Note: [1] 8 bytes BINARY with 10-18 digits are not supported.

Chapter 20. Grammar

This chapter describes the program mapper grammar: Base elements of the grammar are tokens, keywords in uppercase, constants and comments. Combining the base elements you are able to define mapping elements: forward/backwardmapping (FM/BM) and structure/interface definitions (IF,ST). The forward/backwardmapping consists of rules (RL) which combine the structure and interface elements (IFE,STE). The following graphic shall illustrate the relationship between all these elements.

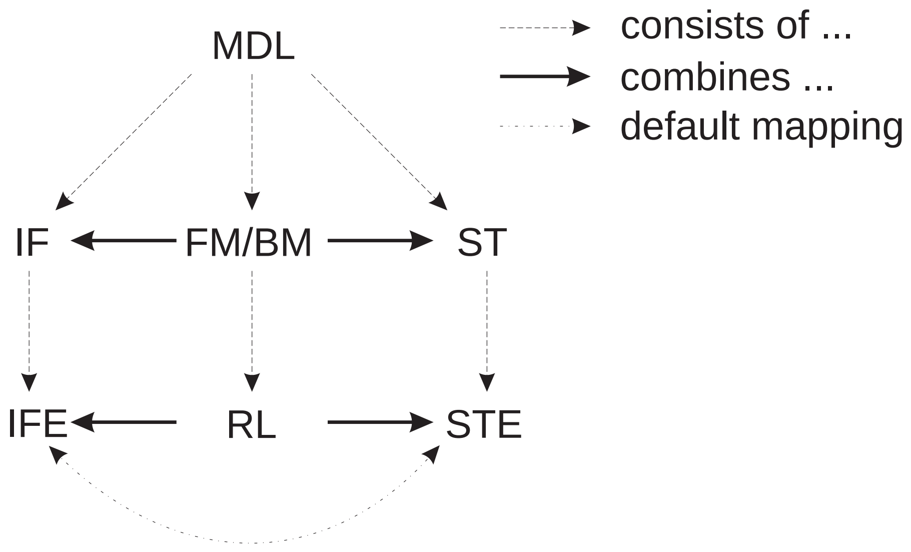


Figure 14. Relationship between mapping elements.

Grammar elements

Comments

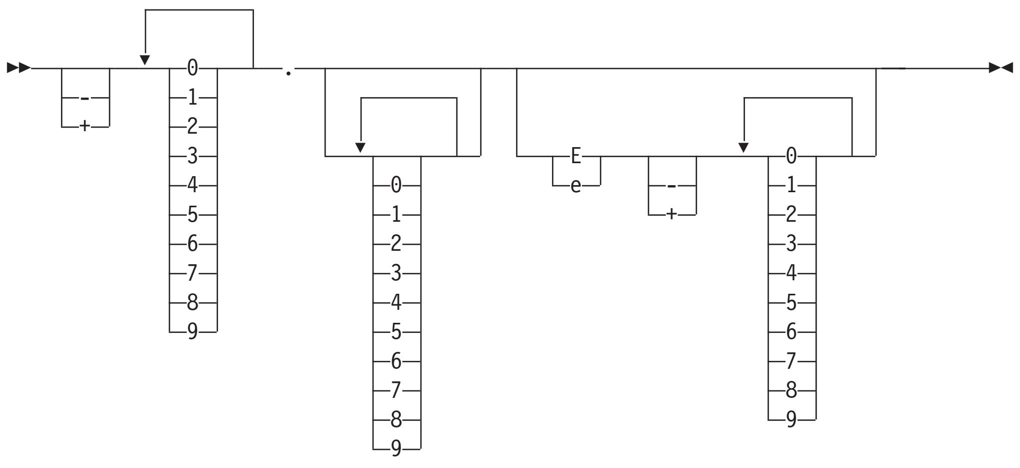
Comments should be used to document the mapping definition. There exist two types of comments: C style, for example /* 'comment' */ and C++ style, for example // 'comment' at the end of a line.

Comments started with '/*' and ended with '*/' can be at any place between syntax tokens. Comments starting with // can be at the end of every line. Nesting of '/* ... */' is not allowed.

Tokens

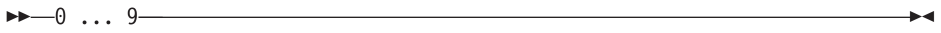
Tokens are the base element of the grammar and therefore every token is explained in detail. For every token at least a syntax diagram and an example is given.

FLOAT_TOKEN



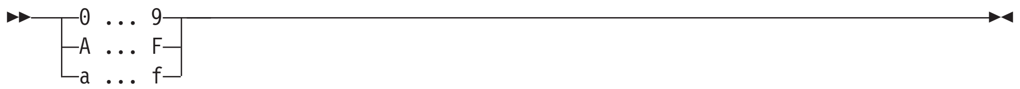
Example: -7.42E-4 which would equal -0.000742.

Note: In order to keep the diagrams more simple the possibility to choose a single number from 0 to 9 will be displayed with



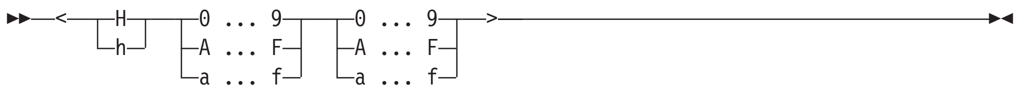
from now on.

Hex_digit



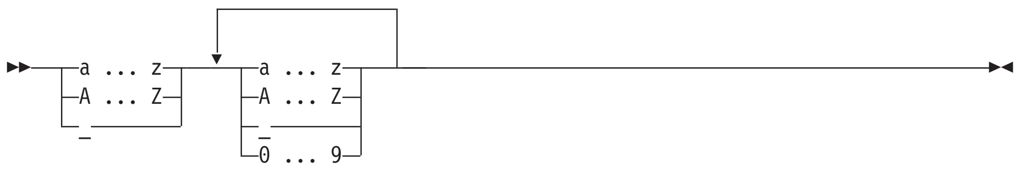
Examples: 3, F.

Hex_token



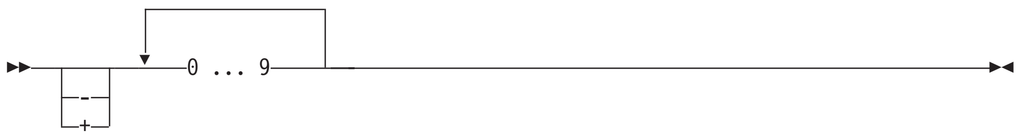
Example: <H4F>

IDENTIFIER



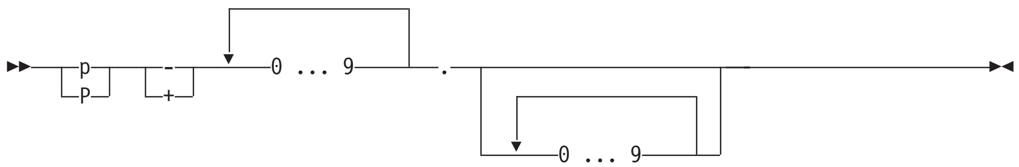
Examples: a_b4_h4, _Z97_bfsd

INT_TOKEN



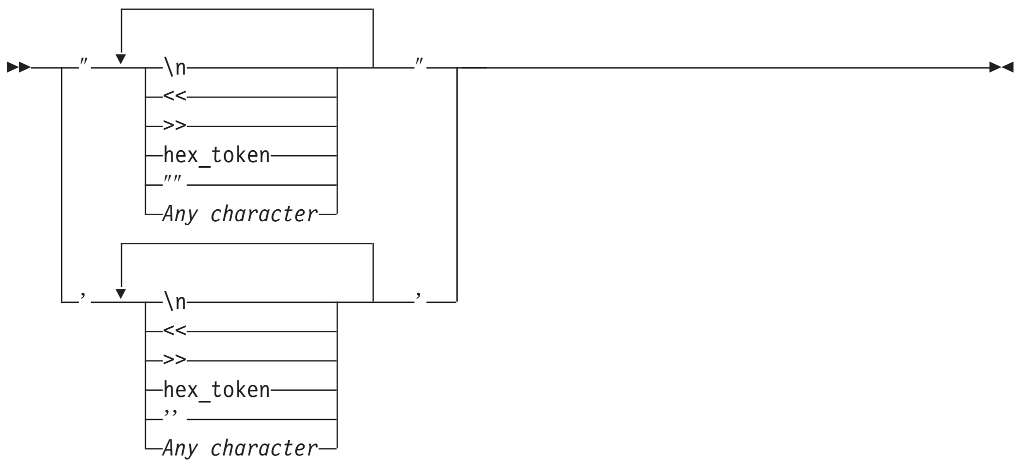
Examples: -89432, 412

PACKED_TOKEN



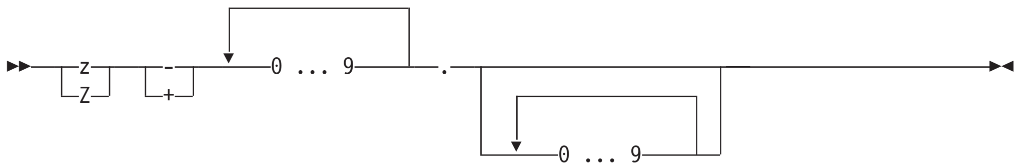
Examples: p+212.2 equals 212.2, p-142.8 equals -142.8

STRING_TOKEN



Example: ″AlbertEinstein″, ″xyz’a″, ’xyz″a’, ’other_example<h15><h12>’

ZONED_TOKEN



Example: z+412.8

Keywords

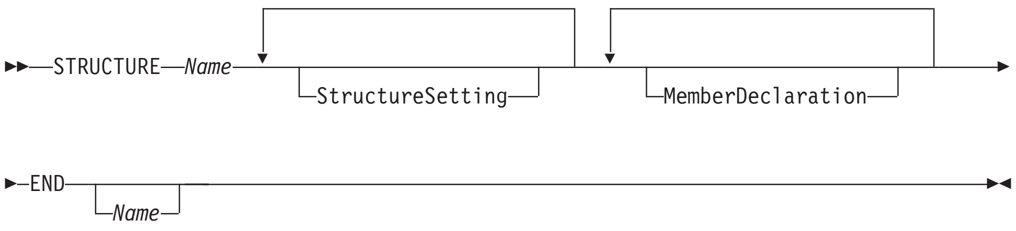
Listed below are all available keywords. Do not name variables with this reserved keywords, because this would cause problems when doing the mapping (for example do not name a structure element 'STRUCTURE').

ARRAY	BACKWARDMAPPING	BINARY	CHAR
CONSTANT	DESCRIPTION	DLL	DOCUMENTATION
END	FIRST	FLOAT	FORWARDMAPPING
FROM	IGNORE	INTEGER	INTERFACE
JUSTIFY	LAST	LEFT	LENGTH
LONG	MAP	MAPPING	MINUS
PACKED	PAD	PARMS	PLUS
RIGHT	SCALE	SEPARATE	SIGNED
STRING	STRUCTURE	TERMINATEDBY	TO
UNSIGNED	USERTYPE	ZONED	

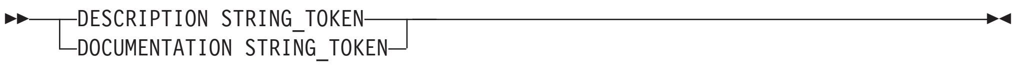
Note: All keywords have to be in uppercase!

Structure definition

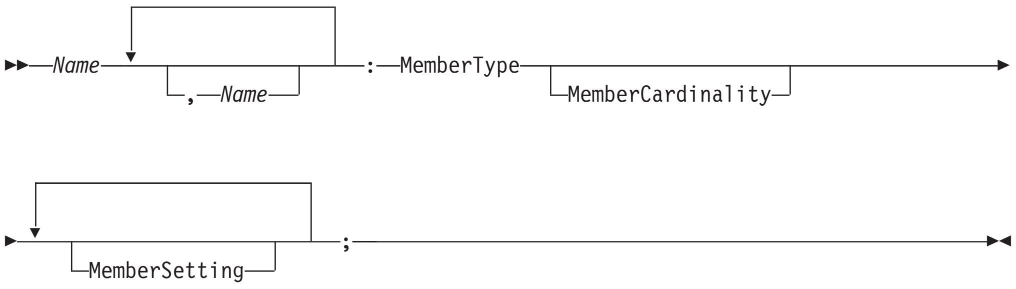
Structure



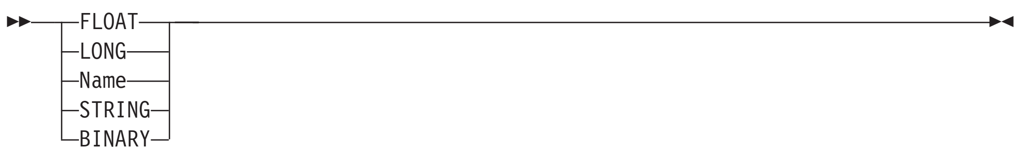
StructureSetting



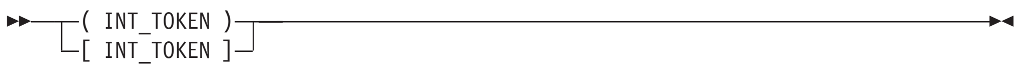
MemberDeclaration



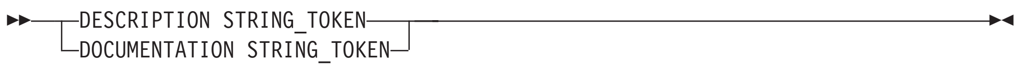
MemberType



MemberCardinality



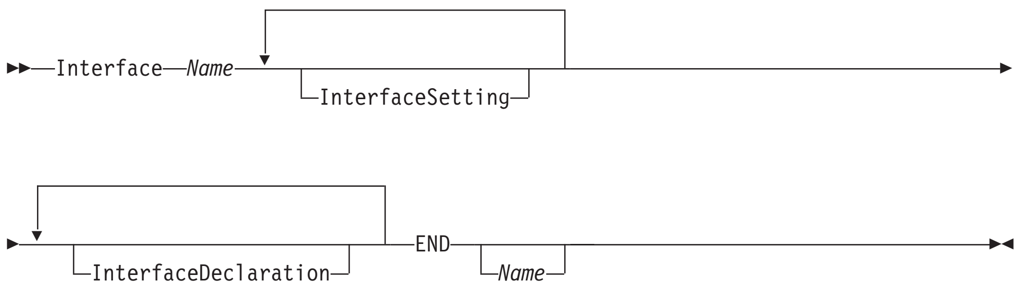
MemberSetting



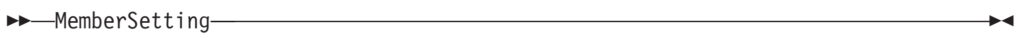
Note:
- Same syntax like structure definitions in FDL.

Interface definition

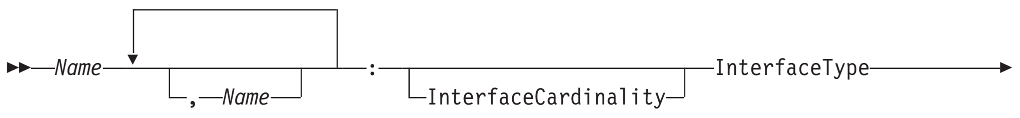
Interface



InterfaceSetting



InterfaceDeclaration



```
►─MemberSetting──;──────────────────────────────────────────────────────────►◄
```

InterfaceType

```
►►──┬─CharInterfaceType────────────────────────┬──────────────────────────►◄
    ├─CharInterfaceType CONSTANT STRING_TOKEN──┤
    ├─IntegerInterfaceType─────────────────────┤
    ├─IntegerInterfaceType CONSTANT INT_TOKEN──┤
    ├─FloatInterfaceType───────────────────────┤
    ├─FloatInterfaceType CONSTANT FLOAT_TOKEN──┤
    ├─PackedInterfaceType──────────────────────┤
    ├─PackedInterfaceType CONSTANT PACKED_TOKEN┤
    ├─ZonedInterfaceType───────────────────────┤
    ├─ZonedInterfaceType CONSTANT ZONED_TOKEN──┤
    ├─Name─────────────────────────────────────┤
    ├─UserInterfaceType────────────────────────┤
    └─UserInterfaceType CONSTANT STRING_TOKEN──┘
```

InterfaceCardinality

```
►►──ARRAY──┬─[ INT_TOKEN ]─┬──────────────────────────────────────────────►◄
           └─( INT_TOKEN )─┘
```

Note:

- The sequence of elements is significant and defines the sequence of the elements in the data area used for forwardmapping and backwardmapping. Every element has a fixed offset from the start of the data area. Make sure the interface elements have the size and type of the data the legacy application expects (See “Valid conversions between MQSeries Workflow container program mapping element types and program mapping interface types” on page 112 for size informations).
- The sequence of member definitions in the structure is not relevant for the mapping. Mapping is done by name from interface elements to structure elements.

Interface types

PackedInterfaceType

```
►►──PACKED──( INT_TOKEN )──PackedAttributeList─────────────────────────────►◄
```

PackedAttributeList

```
►►──┬─SIGNED──MINUS STRING_TOKEN──PLUS STRING_TOKEN─┬──SCALE INT_TOKEN────►◄
    └─UNSIGNED STRING_TOKEN──────────────────────────┘
```

Examples: p+212.2 equals 212.2, p-142.8 equals -142.8

Scale is used to define the decimal point of the packed number and defines the factor used by conversion. The packed number is multiplied by 10^scale in BACKWARDMAPPING and divided by 10^scale in FORWARDMAPPING. From the plus character, minus character and unsigned character only the right nibble is used that means the value has to be <= x'0f'.

Example:
Packed number is 4711, scale is 0. Decimal number is 4711.
Packed number is 4711, scale is 2. Decimal number is 471100.
Packed number is 4711, scale is -2. Decimal number is 47.11.

Format: PACKED(5) SIGNED MINUS ″<h0d>″ PLUS ″<h0c>″ SCALE 1;

Byte0 Byte1 Byte2

DD DD DS

where D is a digit 0-9 and S is the positive or negative sign

Note:
- The size in bytes used by the packed number is packed number (size + 1) / 2 rounded up to the next integer.
- Packed numbers can create runtime conversion errors if the digits are > 9 or the sign does not match the sign defined for the interface element.

ZonedInterfaceType

```
►►──ZONED──( INT_TOKEN )──PackedAttributeList───────────────────────────────►◄
```

ZonedAttributeList

```
►►─┬─SIGNED───┬─FIRST─┬───MINUS STRING_TOKEN──PLUS STRING_TOKEN──┬──────────┬─┬────►
   │          └─LAST──┘                                           └─SEPARATE─┘ │
   └─UNSIGNED──────────────────────────────────────────────────────────────────┘

►─SCALE──────────────────────────────────────────────────────────────────────►◄
```

Examples: z+471.1 equals 471.1, z-142.8 equals -142.8.

The size defines the number of significant digits used by the zoned number. Scale is used to define the decimal point of the zoned number and defines the factor used by conversion. The zoned number is multiplied by 10^scale in backwardmapping and divided by 10^scale in forwardmapping. FIRST and LAST define where the sign is located in the number. From the plus character, minus character and unsigned character only the right nibble is used if the sign is not separate that means the value has to be <= x'0f'. If the sign is separate the first character as a whole is used, that means the character has to be <= x'ff'. FIRST and LAST define the location of the sign (see examples below).

Example:
Zoned number is 4711, scale is 0. Decimal number is 4711.

Zoned number is 4711, scale is 2. Decimal number is 471100.
Zoned number is 4711, scale is -2. Decimal number is 47.11.

Format:
ZONED(3) SIGNED LAST LAST MINUS ″<h0d>″ PLUS ″<h0c>″ SCALE 2

```
Byte0 Byte1 Byte2
 FD    FD    SD
```

where D is a digit 0-9 and S is the positive, negative or unsigned sign
and F are the zoned bits.

Format:
ZONED(3) SIGNED FIRST LAST MINUS ″<h0d>″ PLUS ″<h0c>″ SCALE 2

```
Byte0 Byte1 Byte2
 SD    FD    FD
```

where D is a digit 0-9 and S is the positive, negative or unsigned sign
and F are the zoned bits.

Format:
ZONED(3) SIGNED LAST SEPARATE MINUS ″-″ PLUS ″+″ SCALE 2

```
Byte0 Byte1 Byte2 Byte3
 FD    FD    FD    XX
```

where D is a digit 0-9 and S is the positive, negative or unsigned sign
and F are the zoned bits and XX is the sign (either x′4E′ or x′60′).

Note:
- The size in bytes used by the zoned number is zoned number size. If the sign is separate one additional byte is used.
- Zoned numbers can create runtime conversion errors if the digits are > 9 and the zone does not contain x′f′ or the sign does not match the sign defined for the interface element.
- FIRST and LAST define where to append the sign.

IntegerInterfaceType

```
►►──┬─SIGNED───┬──INTEGER──┬─16─┬──────────────────────────────────►◄
    └─UNSIGNED─┘           └─32─┘
```

FloatInterfaceType

```
►►──FLOAT──┬─32─┬───────────────────────────────────────────────────►◄
           └─64─┘
```

CharacterInterfaceType

```
►►──CHAR──( INT_TOKEN )──┬──────────────────────────┬───────────────►
                         └─TERMINATEDBY STRING_TOKEN─┘
```

```
►─┬──────────────────────────┬──────────────────────────────────────────────►◄
  └─CharInterfaceAttributeList─┘
```

CharInterfaceAttributeList:

```
►►──JUSTIFY──┬─LEFT──┬──PAD STRING_TOKEN────────────────────────────────────►◄
             └─RIGHT─┘
```

The termination character is inserted in forwardmapping and stripped off in backwardmapping. Padding, alignment and truncation occurs in forwardmapping. The data is not modified in backwardmapping. The character size in bytes has to include the termination character and the number of bytes converted is one less than the specified character size.

Examples for justification:

Length	Content	Length	JUSTIFY(Left)	JUSTIFY(RIGHT)
3	'ABC'	4	'ABC '	' ABC'
3	'AB '	4	'AB '	' AB '
3	' BC'	4	' BC '	' BC'
4	'ABCD'	4	'ABCD'	'ABCD'
4	' BCD'	4	' BCD'	' BCD'
4	'ABC '	4	'ABC '	'ABC '
5	'ABCDE'	4	'ABCD'	'BCDE'
5	' BCDE'	4	' BCD'	'BCDE'
5	'ABCD '	4	'ABCD'	'BCD '

UserInterfaceType

```
►►──USERTYPE──Name──┬────────────────────┬──────────────────────────────────►◄
                    └─PARMS STRING_TOKEN─┘
```

Note: STRING_TOKEN is passed to the UserType exit.

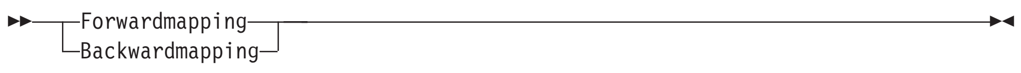
Mapping elements

This chapter illustrates the formal definition and grammar for the MDL. For every mapping element (structure, interface, forward/backwardmapping etc.) exists a syntax diagram which explains how to use the elements correctly. Examples: "Example" on page 109ff and "MDL examples" on page 141ff.

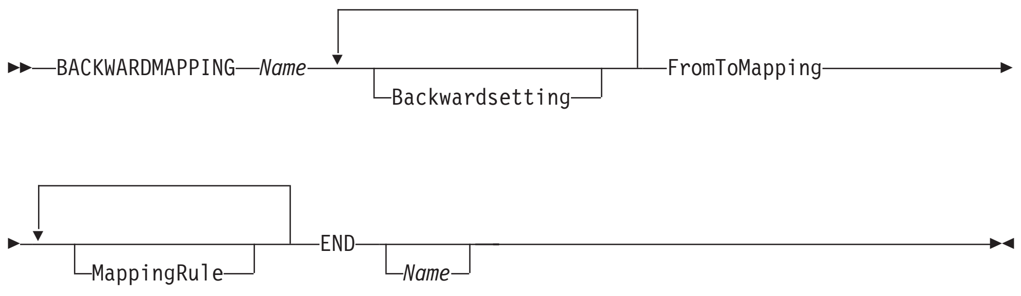
MappingElement

```
►►──┬─Structure─┬───────────────────────────────────────────────────────────►◄
    ├─Interface─┤
    ├─Usertype──┤
    └─Mapping───┘
```

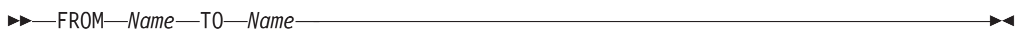
Mapping



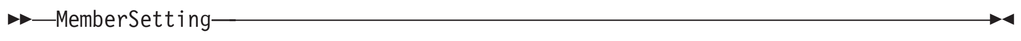
Backwardmapping



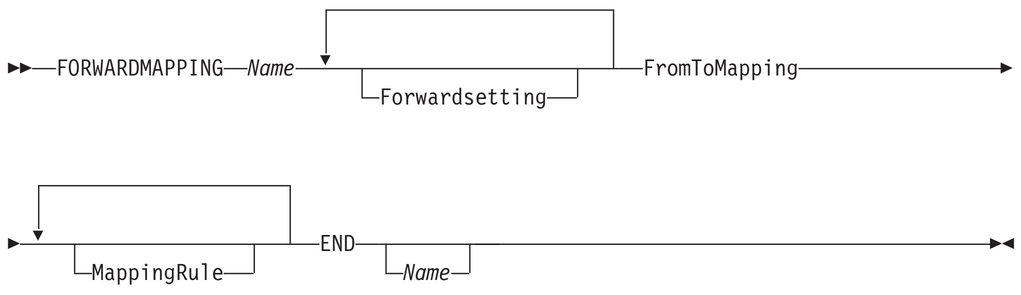
FromToMapping



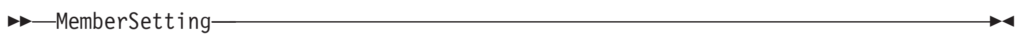
BackwardSetting



Forwardmapping



ForwardSetting



MappingRule

```
►►──MAP──First Name──TO──Second Name──;──────────────────────────────────────►◄
```

Note:
- The first name has to be the interface name in backwardmapping and the structure element name in forwardmapping. The second name has to be the interface name in forwardmapping and the structure element name in backwardmapping (see “Chapter 18. Mapping algorithm” on page 105

UserType definition

UserType

```
                        ┌──────────────────────────┐
                        ▼                          │
►►──UserType──Name──────┴──┬──────────────────┬────┴──UserTypeLength──────────────►
                           └─UserTypeSetting──┘

►─UserTypeDeclaration──END──┬──────┬──────────────────────────────────────────►◄
                            └─Name─┘
```

UserTypeLength

```
►►──LENGTH──(──INT_TOKEN──)──────────────────────────────────────────────────►◄
```

Note: The usertype length defines the size of the usertype in bytes.

UserTypeSetting

```
►►──┬─DESCRIPTION STRING_TOKEN───┬─────────────────────────────────────────────►◄
    └─DOCUMENTATION STRING_TOKEN─┘
```

UserTypeDeclaration

```
►►──DLL──STRING_TOKEN──,──STRING_TOKEN──────────────────────────────────────►◄
```

Note: The first string_token in usertype declaration defines the DLL name, the second defines the exit entry name so it is possible to use one DLL for multiple usertypes.

Sample MDL for C/C++ and COBOL

In this example every definition is shown in detail and the used variables have the same name (interface and structure). Therefore the forward and backwardmapping definition is as simple as possible and an explicit mapping as in the previous example is unnecessary.

```
/*
   --- Structure definition ---
*/

STRUCTURE AccountRepStructureBackw
   LastName:  STRING;
   FirstName:  STRING;
   Zip: LONG;
   Salary: FLOAT;
   Tax: FLOAT;
   Customers: CustomerStructure [3];
END AccountRepStructureBackw

STRUCTURE AccountRepStructureForw
   LastName:  STRING;
   FirstName:  STRING;
   Zip: LONG;
   Salary: FLOAT;
   Tax: FLOAT;
END AccountRepStructureForw

/* In this example the CustomerStructure contains 3 elements
   (last name, first name and phonenumber which are defined as
   string) */

STRUCTURE CustomerStructure
   LastName: STRING;
   FirstName: STRING;
   PhoneNumber: STRING;
END CustomerStructure

/*
   --- Interface definition for Cobol ---
*/

INTERFACE AccountRepInterfaceForCobol
        LastName: CHAR(50) JUSTIFY LEFT PAD ' ';
        FirstName: CHAR(50) JUSTIFY LEFT PAD ' ';
        Zip: UNSIGNED INTEGER 16;
        Salary: PACKED(8) UNSIGNED '<h0c>' SCALE -2;
        Tax: PACKED(2) UNSIGNED '<h0c>' SCALE -2;
        CustomersOpt: ARRAY(3) CustomerInterfaceForCobol;
END AccountRepInterfaceForCobol
INTERFACE CustomerInterfaceForCobol
        LastName: CHAR(50) JUSTIFY LEFT PAD ' ';
        FirstName: CHAR(50) JUSTIFY LEFT PAD ' ';
        PhoneNumber: CHAR(10) JUSTIFY LEFT PAD ' ';

END CustomerInterfaceForCobol

/*
   --- Interface definition for C++ ---
*/

INTERFACE AccountRepInterfaceForCpp
           LastName:  CHAR(50) TERMINATEDBY "<H00>" JUSTIFY LEFT PAD " ";
           FirstName: CHAR(50) TERMINATEDBY "<H00>" JUSTIFY LEFT PAD " ";
           Zip:       UNSIGNED INTEGER 16;
           Salary:    FLOAT 32;
           Tax:       FLOAT 32;
           Customers: ARRAY(3) CustomerInterfaceForCpp;
END AccountRepInterfaceForCpp

INTERFACE CustomerInterfaceForCpp
           LastName:    CHAR(50) TERMINATEDBY "<H00>" JUSTIFY LEFT PAD " ";
           FirstName:   CHAR(50) TERMINATEDBY "<H00>" JUSTIFY LEFT PAD " ";
           PhoneNumber: CHAR(10) TERMINATEDBY "<H00>" JUSTIFY LEFT PAD " ";
END CustomerInterfaceForCpp

/*
  -- Forward/backward mapping definition for Cobol ---
*/

FORWARDMAPPING ForwardSampleForCobol
    FROM AccountRepStructure TO AccountRepInterfaceForCobol
END ForwardSampleForCobol

BACKWARDMAPPING BackwardSampleForCobol
    FROM AccountRepInterfaceForCobol TO AccountRepStructure
END BackwardSampleForCobol

/*
 --- Forward/backward mapping definition for C++ ---
*/

FORWARDMAPPING ForwardSampleForCpp
    FROM AccountRepStructure TO AccountRepInterfaceForCpp
END ForwardSampleForCpp

BACKWARDMAPPING BackwardSampleForCpp
    FROM AccountRepInterfaceForCpp TO AccountRepStructure
END BackwardSampleForCpp
```

This sample is distributed as FMCEMDL in SFMCDATA.

Chapter 21. Usertype

A usertype allows to convert MQSeries Workflow program mapping structure definition elements if the available interface types do not fulfill the required conversion. The program mapper will call a user exit every time a conversion for a usertype is required. It is possible to pass up to 256 characters to the user exit which have to be defined where the interface element is mapped to the usertype. This allows to use the same usertype for different conversions and control the functionality of the exit via passed parameters. In addition it is possible to define parameters at the same time the forward and backward mapping format are defined during buildtime called ’forward mapping parameters’ and ’backward mapping parameters’ and the user exit has access to these parameters. Usertypes have to have a fixed length.

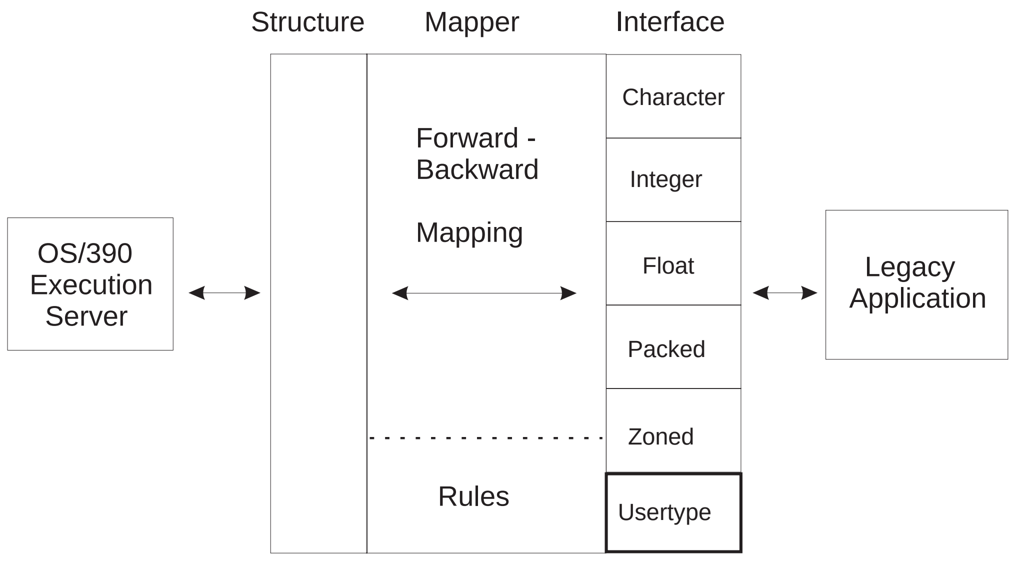


Figure 15. Usertype exit.

Exit interface

The exit interface and passed datastructures are defined in the file fmcxmeut.h.

Note: The exit entry point has to have C linkage.

Following parameters are passed:

1. direction of mapping required by PES

   Use defined constants to check whether forward or backward mapping should be done.

   #define FMC_PROGRAMMAPPING_USERTYPE_BACKWARDMAPPING 0

   #define FMC_PROGRAMMAPPING_USERTYPE_FORWARDMAPPING 1
2. InterfaceDescriptor

   Allows to access the interface element data via a pointer to the data buffer. In addition the length of the usertype in bytes is passed.

```
typedef struct {
        char* elementData;                   // pointer to raw data
        unsigned long elementDataLength;     // length of usertype in bytes
} FmcProgrammMappingInterfaceDescriptor;
```

Warning: Make sure that the data written into the data buffer during forwardmapping is not longer than the size of the usertype in bytes. Otherwise unpredictable effects may occur.

3. StructureDescriptor

   Allows to access the structure element. The elementname and length is passed in addition to a handle to the MQSeries Workflow container. The element can be accessed with the MQSeries Workflow container API by passing the element name.

```
typedef struct {
     const char* elementName;
        //container element name (zero terminated)
     unsigned long elementNameLength;
        // container element name length in bytes
     FmcjContainerHandle containerHandle;
        // container handle
} FmcProgrammMappingStructureDescriptor;
```

   The element name contains the qualified element name and can be used to get or set the container element with the container API. The name is zero terminated.

4. BuildTimeParameter, buildTimeParameterLength

   During buildtime it's possible to insert forward and backward mapping parameters. The length in bytes is passed via buildTimeParameterLength. The name is zero terminated.

5. InterfaceParameter, interfaceParameterLength

   It's possible to insert a parameter in the interface where the usertype is used. The length in bytes is passed via interfaceParameterLength. The name is zero terminated.

   Example:

```
INTERFACE SampleUsertypeInterface
       SampleElement: USERTYPE SampleUsertype PARMS "$";
END
```

6. Return value

   A return value <> 0 signals an error to the program mapper and the return code is set for the activity implementation.

   For a sample see sample usertype exit FMCHSMUT in SFMCDATA.

Creation of DLL

The usertype exit has to be available during runtime of the PES. Any entry name and DLL name can be used but these names have to be identical to the names used in the usertype definition (See below). There exists a sample JCL FMCHJMUT in SFMCDATA to build the sample usertype DLL. Please note that it's possible to have multiple usertype exits in one DLL if they use different function names.

Usertype definition

The usertype definition defines the name of a usertype and the length of the usertype in bytes. In addition the DLL name and exit function name have to be specified.

```
USERTYPE SampleUsertype LENGTH(4)
             DLL "SAMPUTY","SampleUsertypeExit"
END
```

See FMCHEMUT in SMFCDATA for a sample usertype definition.

Chapter 22. Size of program mapping interface definition elements

The interface definition has to match exactly the layout of the data the legacy application expects. If there is only one byte mismatch the results are unpredictable!

The following list summarizes which number of bytes are used by the interface definition element types.

Table 14. Interface elements size

Type	Length	Example
Char	Size equals length in bytes.	Char(2) has a length of 2 bytes.
Integer	2 bytes for INTEGER 16 and 4 bytes for INTEGER 32.	-
Float	4 bytes for FLOAT 32 and 8 bytes for FLOAT 64.	-
Packed	Size used to define the (packed number + 1) divided by 2 and rounded up	Packed(2) equals 2 bytes, Packed(3) equals 2 bytes and Packed(4) equals 3 bytes.
Zoned	Size used to define the zoned number if no separate sign is defined. Otherwise it is one larger than the size used to define the zoned number.	Zoned(4) equals 4 bytes as Zoned(4) SEPERATE equals 5 bytes.
UserType	Size of usertype.	USERTYPE SampleUsertype LENGTH(4) equals 4 bytes.

Note: If there is any alignment done by the compiler who is used to compile the legacy application this alignment also has to be done in the interface definition.

Example: A C structure defined as follows:

```
struct S {
          int x;
          char y;
          int z;
         };
```

might be aligned on 4-byte boundary so that

```
x x x x y       z z  z  z
0 1 2 3 4 5 6 7 8 9 10 11 Byte
```

and therefore needs following interface definition

```
INTERFACE i
          x:   SIGNED INTEGER 32;
          y:   CHAR(1) JUSTIFY LEFT PAD " ";
          pad: CHAR(3) JUSTIFY LEFT PAD "<h00>";
          z:   SIGNED INTEGER 32;
END
```

Chapter 23. Activation of Program Mapping Definitions

The activation of a program mapping definition is described in detail in the *MQSeries Workflow for OS/390: Customization & Administration*, chapter *Administering Program Mappings*. Below you will find a short summary about how to activate a program mapping definition:

1. Copy sample job to FMCHJMPR.
2. Create an MDL.
3. Update control statements for the input utility.
4. Run and compile MDL and insert MDL into mapping database.
5. If existing MDL elements were modified restart PES to activate modifications. For new elements there is no PES restart needed.

Chapter 24. Troubleshooting

In order to provide as much help to you as possible this part of the chapter will give a list of usual problems and typical solutions for it. For a detailed list of error messages refer to the book *Messages* (SC33-7032-00).

Common Errors

Element data mapped is incorrect

Most commonly this is because a mismatch between legacy application data layout and interface definiton.

There is no way for the program mapper to check whether the interface maps correctly to the data format and layout the legacy application expects. Every interface should be carefully created and double checked. If there occur runtime conversion errors (only for packed and zoned interface types) this is mostly caused by this. In addition reflect alignment on the legacy side in the interface (see "Chapter 22. Size of program mapping interface definition elements" on page 133 for more details). If the size of one interface element is incorrect (for example integer 16 instead of integer 32) all the following data will be incorrect.

Elements not mapped

Either the element names are different and no mapping rule was specified or a mappingrule uses the wrong element names.

Note: If a mapping rule is used in both directions the mapping rule arguments have to be switched.

Modified mapping definition is not activated

Mapping definitions are reloaded whenever the PES is restarted. It is not sufficient to import the definition into the program mapping database. New definitions will be used without a PES restart.

Chapter 25. Additional Mapping Examples

Application examples

CICS C++ Application

This C++ application under CICS displays the data, creates new customers and increases the salary by 8 percent. It corresponds to the forward/backwardmapping example in “Forward/Backwardmapping Definition” on page 102.

```
#pragma XOPTS(SP)

#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>

/* --- CustomerStructure --- */
#pragma pack(1)

struct CustomerStructure {
   char LastName[50];
   char FirstName[50];
   char PhoneNumber[10];
};

/* --- AccountRepStructure --- */

struct AccountRepStructure {
   char LastName[50];
   char FirstName[50];
   short Zip;
   float Salary;
   float Tax;
   struct CustomerStructure Customers[3];
};

#pragma pack(1)
int main()
{

   struct AccountRepStructure *commarea;

   EXEC CICS ADDRESS COMMAREA(commarea) EIB(dfheiptr);

   if (dfheiptr->eibcalen <= 0) {
      cout << "??? Empty commarea ???" << endl;
      EXEC CICS RETURN;
   }

// Display all data

   cout << "LastName:   " << commarea->LastName << endl;
   cout << "FirstName:  " << commarea->FirstName << endl;
   cout << "Zip:        " << commarea->Zip << endl;
   cout << "Salary:     " << commarea->Salary << endl;
   cout << "Tax:        " << commarea->Tax << endl;

// Create customers

   strcpy(commarea->Customers[0].LastName,"EINSTEIN");
   strcpy(commarea->Customers[0].FirstName,"ALBERT");
   strcpy(commarea->Customers[0].PhoneNumber,"3048");
   strcpy(commarea->Customers[1].LastName,"NEWTON");
   strcpy(commarea->Customers[1].FirstName,"ISAAK");
   strcpy(commarea->Customers[1].PhoneNumber,"4041");
   strcpy(commarea->Customers[2].LastName,"HAWKINGS");
   strcpy(commarea->Customers[2].FirstName,"STEVEN");
   strcpy(commarea->Customers[2].PhoneNumber,"5154");

   for (int i=0; i<3; i++) {
      cout << "Customer LastName : "
         << commarea->Customers[i].LastName << endl;
      cout << "Customer FirstName : "
         << commarea->Customers[i].FirstName << endl;
      cout << "Customer PhoneNumber : "
         << commarea->Customers[i].PhoneNumber << endl;
   }

// Increase salary by 8%

   commarea->Salary *= 1.08;

   cout << "New Salary: " << commarea->Salary << endl;

   EXEC CICS RETURN;

}
```

CICS COBOL Application

This COBOL application under CICS is doing the same like “CICS C++ Application” on page 139 (displays the data, creates new customers and increases the salary by 8 percent). It corresponds to the forward/backwardmapping example in “Forward/Backwardmapping Definition” on page 102.

```
        IDENTIFICATION DIVISION.
        PROGRAM-ID. "SAMPCBL".

        DATA DIVISION.

          WORKING-STORAGE SECTION.

          01 PRINT-SALARY PIC Z(5)9.9(2).
          01 PRINT-TAX PIC Z9.99.

          LINKAGE SECTION.

          01  DFHCOMMAREA.
      *
      *     AccountRepStructure
      *
             02  LASTNAME PIC X(50).
             02  FIRSTNAME PIC X(50).
             02  ZIP  PIC 9999 COMP-4.
             02  SALARY PIC 9(6)V9(2) COMP-3.
             02  TAX PIC V99 COMP-3.
      *
      *     CustomerStructure
      *
             02  CUSTOMERS OCCURS 3 TIMES
                    INDEXED BY CUSTOMER-INDEX.
                03 LAST-NAME PIC X(50).
                03 FIRST-NAME PIC X(50).
                03 PHONE-NUMBER PIC X(10).
```

```
 PROCEDURE DIVISION.

      IF EIBCALEN <= 0
         DISPLAY "??? EMPTY COMMMAREA ???"
         EXEC CICS RETURN
         END-EXEC
      END-IF

*
*     Display all data
*

      DISPLAY "Lastname:  " LASTNAME
      DISPLAY "Firstname: " FIRSTNAME
      DISPLAY "Zip:       " ZIP

      MOVE TAX TO PRINT-TAX
      MOVE SALARY TO PRINT-SALARY

      DISPLAY "Salary:    " PRINT-SALARY
      DISPLAY "Tax:       " PRINT-TAX

*
*     Create some customers
*

      MOVE "EINSTEIN" TO LAST-NAME(1)
      MOVE "ALBERT" TO FIRST-NAME(1)
      MOVE "3048" TO PHONE-NUMBER(1)
      MOVE "NEWTON" TO LAST-NAME(2)
      MOVE "ISAAK" TO FIRST-NAME(2)
      MOVE "4041" TO PHONE-NUMBER(2)
      MOVE "HAWKINGS" TO LAST-NAME(3)
      MOVE "STEPHEN" TO FIRST-NAME(3)
      MOVE "5154" TO PHONE-NUMBER(3)

      PERFORM
         VARYING CUSTOMER-INDEX FROM 1 BY 1
            UNTIL CUSTOMER-INDEX > 3
         DISPLAY "Customer LastName : "
            LAST-NAME(CUSTOMER-INDEX)
         DISPLAY "Customer FirstName: "
            FIRST-NAME(CUSTOMER-INDEX)
         DISPLAY "Customer PhoneNumber: "
            PHONE-NUMBER(CUSTOMER-INDEX)
      END-PERFORM

*     Increase salary by 8%

      COMPUTE SALARY = SALARY * 1.08

      MOVE SALARY TO PRINT-SALARY
      DISPLAY "New Salary: " PRINT-SALARY

      EXEC CICS RETURN
      END-EXEC

      GOBACK.
```

MDL examples

This chapter illustrates some examples how to use the mapper. There are examples coded in C, Cobol and for simple and complex datastructure.

Simple datastructure with default name mapping

In this example the mapping is defined for a simple data structure and the used mapping is the default one (which means that every element of a container is mapped to the element with the **same name** in the other container).

```
STRUCTURE SimpleDataStructure
        element1: STRING;
        element2: STRING;
        element3: LONG;
        element4: FLOAT;
        element5: BINARY;
        element6: BINARY;
        element7: LONG(20);
        END SimpleDataStructure
INTERFACE SimpleDataInterface
        DESCRIPTION 'This is an example of a simple interface mapping'
        element1: CHAR(10) TERMINATEDBY "<h00>" JUSTIFY LEFT PAD ' ';
        element2: CHAR(20) JUSTIFY LEFT PAD "<h00>";
        element3: SIGNED INTEGER 16;
        element4: FLOAT IBM 32;
        element5: CHAR(500);
        element6: CHAR(200);
        element7: ARRAY (10) SIGNED INTEGER 8;
        END SimpleDataInterface
BACKWARDMAPPING SimpleMapping
        FROM SimpleDataInterface
        TO SimpleDataStructure
        END SimpleMapping
FORWARDMAPPING SimpleMapping
        FROM SimpleDataStructure
        TO SimpleDataInterface
        END SimpleMapping
```

Complex datastructure with default name mapping

In this example the mapping is defined for a complex data structure and the used mapping is the default one (which means that every element of a container is mapped to the element with the **same name** in the other container).

```
STRUCTURE ComplexDataStructure1
        element1: STRING (10);
        element2: FLOAT;
        END ComplexDataStructure1
STRUCTURE ComplexDataStructure2
        element1: STRING (20);
        element2: ComplexDataStructure (5);
        END ComplexDataStructure2
INTERFACE ComplexDataInterface1
        element1: CHAR(10) TERMINATEDBY "<h00>" JUSTIFY LEFT PAD ' ';
        element2: FLOAT IBM 32";
        END ComplexDataInterface1
INTERFACE ComplexDataInterface2
        element1: CHAR(20) JUSTIFY RIGHT PAD ' ';
        element2: ARRAY(5) ComplexDataInterface1;
        END ComplexDataInterface2
BACKWARDMAPPING ComplexMapping
        FROM ComplexDataInterface2
        TO ComplexDataStructure2
            /*
             * Implicitly element1 and element2 are mapped
             */
        END ComplexMapping
FORWARDMAPPING ComplexMapping
        FROM ComplexDataStructure2
        TO ComplexDataInterface2
            /*
             * Implicitly element1 and element2 are mapped
             */
        END ComplexMapping
```

Complex datastructure with non default name mapping

In this example the mapping is defined for a complex data structure and the used mapping is not default one (which means that the structure elements of ComplexDataStructure1 do not have identical names in the interface ComplexStructure1 and are mapped explicitly).

```
STRUCTURE ComplexDataStructure1
        strs: STRING (10);
        flts: FLOAT;
        END ComplexDataStructure1
INTERFACE ComplexDataInterface1
        stri: CHAR(10) TERMINATEDBY "<h00>" JUSTIFY LEFT PAD ' ';
        flti: FLOAT IBM 32;
        END ComplexDataInterface1
BACKWARDMAPPING ComplexMapping
        FROM ComplexDataInterface1
        TO ComplexDataStructure1
           MAP stri TO strs;
           MAP flti TO flts;
        END ComplexMapping
FORWARDMAPPING ComplexMapping
        FROM ComplexDataStructure1
        TO ComplexDataInterface1
           MAP strs TO stri;
           MAP flts TO flti;
        END ComplexMapping
```

Complex datastructure with non default name mapping with arrays and structures

In this example the mapping is defined for a complex data structure and the used mapping is not default one (which means that the elements (in this case arrays!) of structure ComplexDataStructure1 do not have identical names in the Interface ComplexStructure1 and are mapped explicitly).

```
STRUCTURE ComplexDataStructure1
        element1: STRING (10);
        element2: FLOAT;
        END ComplexDataStructure1
STRUCTURE ComplexDataStructure2
        element1: STRING (20);
        element2: ComplexDataStructure1 (5);
        specials: FLOAT;
        END ComplexDataStructure2
STRUCTURE ComplexDataStructure3
        element1: STRING (5);
        element2: ComplexDataStructure2 (4);
        element3: ComplexDataStructure1 (4);
        END ComplexDataStructure3
INTERFACE ComplexDataInterface1
        element1: CHAR(10) TERMINATEDBY "<h00>" JUSTIFY LEFT PAD ' ';
        element2: FLOAT IBM 32;
        END ComplexDataInterface1
INTERFACE ComplexDataInterface2
        element1: CHAR(20) TERMINATEDBY "<h00>" JUSTIFY RIGHT PAD '*';
        element2: ARRAY (5) ComplexDataInterface1;
        speciali: FLOAT IBM 32
        END ComplexDataInterface2
INTERFACE ComplexDataInterface3
        element1: CHAR(5) JUSTIFY RIGHT PAD '*';
        element2: ARRAY (4) ComplexDataInterface1;
        elementx: ARRAY (4) ComplexDataInterface2;
        END ComplexDataInterface3
BACKWARDMAPPING ComplexMapping
        FROM ComplexDataInterface3
        TO ComplexDataStructure3
         /* Interface element elementx is explicitly mapped to structure
            element element2. All structure and interface elements of this
            structure are mapped per default; interface element element2 is
            also explicitly mapped with all its subelements. */
           MAP 'elementx' to 'element2';
           MAP 'element2' to 'element3';

           /* element speciali is not mapped per default and an explicit rule
              for this element is required */

           MAP 'elementx.speciali' to 'element2.specials';

           /* Per default mapping element1 with all subelements is mapped */

END ComplexMapping

FORWARDMAPPING ComplexMapping
        FROM ComplexDataStructure1
        TO ComplexDataInterface1
         /* Structure element element2 is explicitly mapped to interface
            element elementx. All structure and interface elements of this
            structure are mapped per default; structure element element3 is
            also explicitly mapped with all its subelements. */
           MAP 'element2' to 'elementx';
           MAP 'element3' to 'element2';

           /* element specials is not mapped per default and an explicit rule
              for this element is required */

           MAP 'element2.specials' to 'elementx.speciali';

           /* Per default mapping element1 with all subelements is mapped */

        END ComplexMapping
```

Simple datastructure with all interface types with CONSTANTS and usertypes

In this example the mapping is defined for a simple data structure and the used mapping is not default one (which means that the elements of structure ComplexDataStructure1 do not have identical names in the interface ComplexStructure1 and are mapped explicitly). Additional there is a usertype defined, which converts a 4 byte integer into a Workflow string, seperates every three digits by a '.' and prefixes the string with a currency symbol, for example $1.234.567.

```
/* A usertype which converts a 4 byte integer into a Workflow string, seperates
 * every three digits by a '.' and prefixes the string with a currency
 * symbol, for example $1.234.567 */
USERTYPE user1 LENGTH(4)
                DLL "dlluser","user2Inbound"
         END user1
STRUCTURE SimpleDataStructure
        element1: LONG;
        element2: STRING;
        element3: LONG;
        element7: LONG(20);
        element8: STRING;
        END SimpleDataStructure
INTERFACE SimpleDataInterface
        DESCRIPTION 'This is an example of a simple interface mapping'
        /* following integer constant is inserted in forwardmapping
         * and removed in backwardmapping */
        insert:   SIGNED INTEGER 16
                  CONSTANT 4711
        element3: SIGNED INTEGER 16;
        element7: ARRAY (100) SIGNED INTEGER 8;
        element1: USERTYPE user2 PARMS 'DM'; /* three digits */
        element8: USERTYPE user2 PARMS '$'; /* for example $ 12.345 */
        element9: CHAR(5) CONSTANT "This is a string constant with some
                        hex chars <h47><h11>" JUSTIFY RIGHT PAD '*'
        element10: PACKED (10) CONSTANT p47.11
                        SIGNED FIRST MINUS "<h0d>" PLUS "<h0c>" UNSIGNED "<h0c>"
                        SCALE 5
        element11: ZONED (10) CONSTANT z47.11
                        SIGNED FIRST MINUS "<h0d>" PLUS "<h0c>" SCALE 5
        element12: FLOAT IBM 8
                   CONSTANT +47E11
        END SimpleDataInterface
BACKWARDMAPPING SimpleMapping
        FROM SimpleDataInterface
        TO SimpleDataStructure
        END SimpleMapping
FORWARDMAPPING SimpleMapping
        FROM SimpleDataStructure
        TO SimpleDataInterface
        END SimpleMapping
```

Part 5. Program Execution Server Exits

Chapter 26. Introduction

For extensibility of MQSeries Workflow for OS/390 the PES uses exits in following areas:

- Application invocation
- Legacy application program mapping

There are exits provided by MQSeries Workflow for OS/390 for program mapping and program invocation. Whenever there are new invocation exits or mapping exits needed these exits can be used instead of the IBM supplied exits or in parallel.

Both kind of exits have an interface defined to which every user written exit has to conform.

Every exit type has to provide an init function which is called from the PES when the exit is needed the first time. Later the exit specific functions are called (See the specific exit chapters for more details). A shutdown request for the PES triggers a call to the deinit function of the exits.

Usually the init function does all the initialization needed for the exit. If there are informations needed further on in following calls a handle can be filled in the initialization call which is passed to all following functions (for example DB handles, connection informations, states, ...). Deinit which is called last normally deallocates and frees all resources allocated during init.

The exit DLL is loaded by the PES when the exit is needed the first time and unloaded when the PES terminates.

The main difference between the two types of exits is the following:

- **Mapping** exits do a data conversion between MQSeries Workflow for OS/390 containers and data acceptable by legacy applications.
- **Invocation** exits do invoke applications on the application side.

Note:

- Whenever you modify an exit you have to shutdown and reboot the PES in order to make your changes effective.
- All PES exits have to be reentrant.

Return codes and error messages

All exits use a return code to signal availability of error informations from the exit functions to the PES. If the return code is not OK 4 parameters used in every function contain more detailed error information. Errors can be recoverable or unrecoverable. A recoverable error is passed by the PES to the program activity and the program activity is set into an error state. An recoverable error causes the PES instance to terminate. An errorId and errorDescription are written to FMCERRxx (DDname).

Parameters

char * errorIdBuffer

- Character buffer provided by PES which is 4 characters long and is used to pass an error number. Has to be set accordingly if the id is shorter than 4 characters.
- Input/output parameter

long * errorIdBufferLength

- Length of the message number in errorIdBuffer which has to be set by the exit. The maximum available number of characters is passed in. Valid lengths are between 0 and the passed in value in errorIdBufferLength.
- Input/output parameter

char * errorDescriptionBuffer

- Character buffer provided by PES which is 512 characters long and is used for an error message. Has to be set accordingly if the id is shorter than 512 characters.
- Input/output parameter

long * errorDescriptionBufferLength

- Length of the description in errorDescriptionBuffer which has to be set by the exit. The maximum available number of characters is passed in. Valid lengths are between 0 and the passed in value in errorDescriptionBufferLength.
- Input/output parameter

The error id may consist of up to 4 digits and is prefixed by the PES by a character identifying the exit type (I for invocation, M for mapping). In addition the errorDescription is prefixed by the PES with the dllname of the exit so that every exit can use the message numbers and the dllname is the identifier of the exit.

Return codes

FMC_EXIT_OK
Function was successful.

FMC_EXIT_RECOVERABLE_ERROR
Function was unsuccessful but recoverable. The PES will return message FMC32204 (see MQSeries Workflow for OS/390 Messages) with the passed error information. The PES continues processing. errorIdBuffer, errorIdBufferLength, errorDescriptionBuffer and errorDescriptionBufferLength have to be set accordingly .

FMC_EXIT_NONRECOVERABLE_ERROR
Function was unsuccessful and unrecoverable. The errorId and errorDescription are written to STDERR. The PES instance terminates, errorIdBuffer, errorIdBufferLength, errorDescriptionBuffer and errorDescriptionBufferLength have to be set accordingly.

Chapter 27. Interfaces for all exits

Note: All interfaces must have C linkage!

Init

Header files

FMCXMIF.H (program mapping exit) and FMCXIEP.H (invocation exit).

Function

- Initialize exit.
- It is called once when the exit is used the first time.

Interface

long Init

```
( void **  exitHandle,
  void *   initializationParameter,
  long     initializationParameterLength,
  char *   errorIdBuffer,
  long *   errorIdBufferLength,
  char *   errorDescriptionBuffer,
  long *   errorDescriptionBufferLength)
```

Parameters

void ** exitHandle

- Pointer passed into the init function which is not used by the PES but passed to any function called later on. Used to pass exit environment data between all exit functions
- Input/output parameter

void * initializationParameter

- Parameters defined in the PES directory entry for the exit which can be used to customize the exit initialization. The parameter is terminated by zero.
- Input parameter

long initializationParameterLength

- Length of the initializationParameter in bytes.
- Input parameter

char * errorIdBuffer

- See "Return codes and error messages" on page 149 for detailed information.

long * errorIdBufferLength

- See "Return codes and error messages" on page 149 for detailed information.

char * errorDescriptionBuffer

- See “Return codes and error messages” on page 149 for detailed information.

long * errorDescriptionBufferLength

- See “Return codes and error messages” on page 149 for detailed information.

Return codes

See “Return codes and error messages” on page 149 for the return codes.

Deinit

Header files

FMCXMIF.H (program mapping exit) and FMCXIEP.H (invocation exit).

Function

- Deinitialize exit.
- It is called once when the PES terminates.

Interface

long Deinit

```
( void **  exitHandle,
  char *   errorIdBuffer,
  long *   errorIdBufferLength,
  char *   errorDescriptionBuffer,
  long *   errorDescriptionBufferLength)
```

Parameters

void ** exitHandle

- Pointer passed into the init function which is not used by the PES but passed to any function called later on. Used to pass exit environment data between all exit functions.
- Input/output parameter

char * errorIdBuffer

- See “Return codes and error messages” on page 149 for detailed information.

long * errorIdBufferLength

- See “Return codes and error messages” on page 149 for detailed information.

char * errorDescriptionBuffer

- See “Return codes and error messages” on page 149 for detailed information.

long * errorDescriptionBufferLength

- See “Return codes and error messages” on page 149 for detailed information.

Return codes

See “Return codes and error messages” on page 149 for the return codes.

Chapter 28. Program mapping exit

There exist a IBM supplied program mapping exit which can be used to convert and translate data between legacy applications and MQSeries Workflow for OS/390. Every other mapping exit who is conform to the mapping exit interface can be used in parallel to the IBM supplied mapping exit or replace the IBM supplied mapping exit.

The program mapping exit has to be available in a DLL which is loaded by the PES and used to translate the MQSeries Workflow for OS/390 container data into data accepted by a legacy application. This DLL has to have two exit functions: init and deinit (see above) and one exit specific function *translate*. It is used to do the actual conversion and translation for forward mappings and backward mappings every time a legacy application is called. The raw buffer for the legacy application is available and the container can be accessed via MQSeries Workflow for OS/390 container API calls.

In forward mapping calls the program mapping exit has to extract the data from the container and translate the data into a raw buffer which is passed to the legacy application. In backward mapping calls the program mapping exit has to translate the raw buffer and assign the data to the container.

Additional program mapping exit specific interfaces

A mapping exit has to follow the general rules for PES exits as described in “Chapter 27. Interfaces for all exits” on page 151.

Translate

Header files

FMCXMIF.H (program mapping exit).

Function

- **Translate** is called every time by the PES in direction FMC_PROGRAMMAPPING_BACKWARDMAPPING if backward mapping has to be done and in direction FMC_PROGRAMMAPPING_FORWARDMAPPING if forward mapping has to be done.

Interface

long Translate

```
(void *  exitHandle,
 short   direction,
 char *  mappingName,
 long    mappingNameLength,
 char *  buildTimeParameter,
 long    buildTimeParameterLength,
 void *  containerHandle,
 char**  buffer,
 long *  bufferLength,
 char *  errorIdBuffer,
 long *  errorIdBufferLength,
 char *  errorDescriptionBuffer,
 long *  errorDescriptionBufferLength)
```

Parameters

void * exitHandle

- Pointer passed into the init function which is not used by the PES but passed to any function called later on. Used to pass exit environment data between all exit functions
- Input/output parameter

short direction

- Used to identify the translation direction. Either FMC_PROGRAMMAPPING_BACKWARDMAPPING or FMC_PROGRAMMAPPING_FORWARDMAPPING
- Input parameter

char * mappingName

- Forward mapping format or backward mapping format defined in the OS/390 program properties. The name is zero terminated.
- Input parameter

long mappingNameLength

- Length of the mappingNameLength in bytes.
- Input parameter

char * buildTimeParameter

- Forward mapping parameter or backward mapping parameter defined in the OS/390 program properties. Can be used to customize the translation process. The parameter is zero terminated.
- Input parameter

long buildTimeParameterLength

- Length of the buildTimeParameter in bytes.
- Input parameter

void * containerHandle

- Containerhandle to the MQSeries Workflow for OS/390 container used for translation. Argument in MQSeries Workflow for OS/390 container APIs to access the container.
- Input parameter

char ** buffer

- A valid buffer address for forward mapping calls or has to be set by the program mapping exit to a valid buffer address in backward mapping calls. This buffer is passed by the PES to the legacy application in forward mapping calls and passed to the program mapping exit for data conversion and setting of container elements in backward mapping calls.
- Input parameter for forward mapping
- Output parameter for backward mapping

long * bufferLength

- Length of the buffer in bytes. Is already set in forward mapping calls and has to be set in backward mapping calls by the program mapping exit.

- Input parameter for forward mapping
- Output parameter for backward mapping

char * errorIdBuffer

- See “Return codes and error messages” on page 149 for detailed information.

long * errorIdBufferLength

- See “Return codes and error messages” on page 149 for detailed information.

char * errorDescriptionBuffer

- See “Return codes and error messages” on page 149 for detailed information.

long * errorDescriptionBufferLength

- See “Return codes and error messages” on page 149 for detailed information.

Return codes

See “Return codes and error messages” on page 149 for the return codes.

See the program mapping sample “Program mapping exit sample” for more details.

Enabling the PES to use a program mapping exit

In order to use a program mapping exit the exit has to reside in a DLL which has to be available in a linklibrary used by the PES. Customize sample JCL FMCHJMEX in *InstHLQ*.SFMCCNTL and submit JCL.

Then the PES has to know about the program mapping exit by a definition of this exit in the PES directory. See *MQSeries Workflow for OS/390: Customization and Administration* for more informations.

Program mapping exit sample

There exists a C sample program mapping exit (FMCHSMEX in *InstHLQ*.SFMCSRC) with a corresponding JCL to compile and link the exit (FMCHJMEX in *InstHLQ*.SFMCCNTL).

The sample exit gives an example how a program mapping exit should work in general. It also shows how the exit can use all the different parameters passed (exit initialization parameter, forward/backward mapping format and forward/backward mapping parameters), how error messages can be created and how the container elements can be accessed.

The exit is able to convert MQSeries Workflow for OS/390 container elements of type LONG and FLOAT into C types long and double. The elements are converted in the same sequence that the elements are defined in the container structure.

The type of the element is derived from the first character of the element name and has to be defined in the PES directory entry for the mapping exit (exitParameters).

ExitParameter syntax: LONG=x FLOAT=y

where x is the character used to identify LONG container elements and y is the character to identify FLOAT container elements. All elements starting with an undefined character are ignored.

In addition TRACE=YES can be defined during initialization which enables some trace print-outs.

Mapping example:

```
STRUCTURE S
   LE1: LONG(2);
   FE1: FLOAT;
   LE2: LONG;
END;
```

will be converted so that the following structure is filled with conversion data:

```
struct S {
   long LE1[2];
   double FE1;
   long LE2;
};
```

The following three different mapping formats are available:

1. DEFAULT
2. INCREASE_INCOME
3. DECREASE_INCOME

The first format does normal conversion of the values. The second one increases all values by 8% (default) whereas the third format decreases all values by 8% (default). If other percentages are needed they can be defined as forward mapping parameters or backward mapping parameters using the Buildtime tool when the program properties for OS/390 are defined.

To use the mapping sample you have to

1. compile and link the mapping sample (See JCL FMCHJMEX in *InstHLQ*SFMCCNTL).
2. update the PES directory with following definition (See *MQSeries Workflow for OS/390: Customization and Administration*: Importing a PES directory source file)

   (KEYTOMAPPING2)

   ```
   type           =SAMPLE
   exitName       =SAMPEXT
   exitParameters =LONG=L FLOAT=F
   ```
3. define a program in BuildTime which uses the sample mapping type SAMPLE and optionally provide a different increase/decrease amount for forward/backward mapping parameters.

Chapter 29. Program invocation exit

A program invocation exit is used by the program execution server in order to run a requested application on a service system like CICS or IMS. The corresponding invocation request is issued to the program execution server by an MQSeries Workflow for OS/390 execution server.

To handle an invocation request the program execution server uses an exit containing the implementation of an invocation type like *EXCI*. This allows application developers using MQSeries Workflow for OS/390 to attach their own invocation types to MQSeries Workflow for OS/390. An invocation exit has to follow the general rules for program execution server exits as described in "Part 5. Program Execution Server Exits" on page 147 in this book. As all program execution server exits invocation exits are available as dynamic link libraries providing entry points.

Synchronous and asynchronous invocation exits

MQSeries Workflow for OS/390 supports synchronous and asynchronous invocation exits. They will be characterized by the following. Asynchronous invocations must base on MQSeries queues.

Synchronous invocation exit

A synchronous invocation exit

- builds up the connection to the service where request should run.
- runs the requested application on that service by use of the invocation protocol.
- removes the connection to the service after the application terminated.
- passes back the output data from the application or an error message if the invocation failed to the program execution server.
- if - depending on the invocation protocol - connections can be reused by subsequent calls, the connections are cached after being used a first time and removed at deinitialization of the invocation exit.

The program execution server 'waits' for the output from a synchronous invocation exit.

Asynchronous invocation exit

Within processing a request the program execution server calls an asynchronous invocation exit twice. First to handle a request, and second to handle a reply. If an asynchronous invocation exit is called by the program execution server with parameters describing a request it does not execute a request from the program execution server directly on the target service system. It only creates a message consisting of an invocation specific header followed by the data for the requested application and passes it back to the program execution server. Also it creates a message descriptor - which is an MQSeries message descriptor MQMD since only MQSeries based invocations are supported- and passes it also back.

The program execution server then uses message descriptor and message together with connection parameters to put the message to the input queue as specified by the connection parameters. The program execution server does not wait until the requested application has finished but continues with the next request.

If the program execution server gets the reply message consisting of message descriptor and message data it calls all invocation exits in order to get one which can deal with that message. The first of these exits recognizing the message will be called again in order to handle and analyze the reply. Therefore it is important that reply messages from asynchronous invocations are recognized uniquely. That means only invocation protocols should be used where the message contains protocol specific data at the beginning for example CICS bridge header MQCIH or IMS bridge header MQIIH, so that a reply is not handled by an exit which is not the right one. Also it is recommended to reflect that format in the message descriptor (by the Format field in the MQMD structure).

If an exit recognizes the reply it is called to handle it and passes the application output data back to the program execution server or an error message.

Note: An asynchronous invocation exit does **not** connect itself to a service. The MQSeries queues from and to the service are always served by the PES.

Additional invocation exit specific interfaces

Each asynchronous and synchronous invocation exit has to provide the following additional entry points beside the interfaces to be provided by all program execution server exits:

HdlRequ

Header files

FMCXIEP.H (invocation exit).

Function

- For a synchronous invocation this method executes an application program passed as **executableName** on a service to which it connects as defined by **connectionParameters**. In case no error occurred this function returns the output from the application in a buffer passed back by the **programOutput** parameter.
- For an asynchronous invocation this method either creates a request message or it treats the passed-in parameters as reply message. The parameters represent a request message if an application name is passed as **executableName**. If zero is passed as **executableName** the parameters represent a reply message whose data are passed as **executableParameters** and **executableParametersLength** together with a message descriptor MQMD as **messageDescriptor** and **messageDescriptorLength**.

Interface

long HdlRequ

```
(void * exitHandle,
 void * invocationContext,
 char * serviceName,
 long serviceNameLength,
 char * connectionParameters,
 long connectionParametersLength,
 char * executableName,
 long executableNameLength,
 char * executableType,
 long executableTypeLength,
 char * executionParameters,
 long executionParametersLength,
 char ** programOutput,
 long * programOutputLength,
 char ** messageDescriptor,
 char * messageDescriptorLength,
 char * errorIdBuffer,
 long * errorIdBufferLength,
 char * errorDescriptionBuffer,
 long * errorDescriptionBufferLength)
```

Parameters

void * exitHandle

- Reference to the invocation environment for this exit
- Input parameter

void * invocationContext

- Address of data describing the context in which the invocation of the request should be done.
- For the context information provided by MQSeries Workflow for OS/390 see “Invocation Context” on page 164
- input parameter

char * serviceName

- Name of service as known to MQSeries Workflow for OS/390 where the requested application has to be performed.
- Input parameter

long serviceNameLength

- Length of serviceName character string
- Input parameter

char * connectionParameters

- Character string providing the parameters needed by the invocation to connect to the service where the requested program should run. These parameters are defined for the respective service in the PES directory.
- Input parameter
- See also “Connection Parameters” on page 166

long connectionParametersLength

- Length of connectionParameters
- Input parameter

char * executableName

- Address of buffer containing the name of the executable of the request
- For asynchronous invocations only: the program execution server passes zero to indicate that this function is called to handle a reply message from an asynchronously invoked program execution request.
- Input parameter

long executableNameLength

- Length of executableName
- Input parameter

char * executableType
- Type of the program specified by the executableName
- Input parameter

long executableTypeLength
- Length of executableType
- Input parameter

char * executionParameters
- Parameters for the program specified by the executable name
- Reply from an asynchronously performed execution request
- Input parameter

long executionParametersLength
- Length of executionParameters or the asynchronous reply
- Input parameter

char ** programOutput
- Address to return buffer containing the output of the executable
- Asynchronous invocations use this parameter also to return the protocol conformal message to the program execution server corresponding to the execution request passed in as **executableName** and **executionParameters**.
- Input/output parameter

long * programOutputLength
- Address to length of output data or the protocol conformal message
- Input/output parameter

char ** messageDescriptor
- Address to pass/return a buffer containing a message descriptor
- For asynchronous invocations only
- Input/output parameter

long * messageDescriptorLength
- Address to buffer containing the length of the message descriptor
- For asynchronous invocations only
- Input/output parameter

char * errorIdBuffer
- See "Return codes and error messages" on page 149 for detailed information.

long * errorIdBufferLength
- See "Return codes and error messages" on page 149 for detailed information.

char * errorDescriptionBuffer
- See "Return codes and error messages" on page 149 for detailed information.

long * errorDescriptionBufferLength

- See “Return codes and error messages” on page 149 for detailed information.

Return Codes

FMC_EXIT_OK

- See “Return codes and error messages” on page 149 for detailed information.

FMC_EXIT_RECOVERABLE_ERROR

- See “Return codes and error messages” on page 149 for detailed information.

FMC_EXIT_NONRECOVERABLE_ERROR

- See “Return codes and error messages” on page 149 for detailed information.

Recogn

Header files

FMCXIEP.H (invocation exit).

Function

- Checks whether the reply message passed in as message descriptor and message data is recognized by the invocation type this exit represents.
- This method applies to exits for asynchronous invocations. However this entry point must also be provided by synchronous invocation exits. For these exits always FMC_INV_NOT_RECOGNIZED has to be returned.
- This function is called against all asynchronous invocation exits when an invocation reply message is received. It is assumed that there are no ambiguities, so that the first invocation exit where this function recognizes this message is the correct one to handle it.

Interface

long Recogn

```
(void * exitHandle
 char * messageDescriptor,
 long   messageDescriptorLength,
 char * messageData,
 long   messageDataLength)
```

Parameters

void * exitHandle

- Reference to the invocation environment for this exit
- Input parameter

char * messageDescriptor

- Descriptor of the reply message
- Input parameter

long * messageDescriptorLength

- Length of messageDescriptor
- Input parameter

char * messageData

- The data of the reply message
- Input parameter

long messageDataLength

- Length of messageData
- Input parameter

Return Codes

FMC_INV_NOT_RECOGNIZED
The message is not recognized

FMC_INV_RECOGNIZED
The message is recognized

IsAsync

Header files

FMCXIEP.H (invocation exit).

Function

- Returns whether the exit represents an asynchronous invocation.

Interface

long Recogn

```
(void* exitHandle)
```

Parameters

void * exitHandle

- Reference to the invocation environment for this exit
- Input parameter

Return Codes

FMC_INV_SYNCHRONOUS
The invocation is synchronous.

FMC_INV_ASYNCHRONOUS
The invocation is asynchronous.

Invocation Context

The PES passes an invocation context to the invocation exit for HdlRequ. It contains information about the context in which the request should be executed. The context consists of four different context types: Workflow, Security, Transaction, and Performance. The context is created by the PES. The content depends on server and program settings. The workflow context is passed for internal reasons and is not intended to be used by any invocation exit. The security context is either set to the PES user ID or the execution user ID resolved for the request. The transaction context is set to a non-zero value if the request is to be executed as safe application. The performance context is set to the WLM enclave token, if the PES is running WLM managed.

An invocation accesses the context by using a C-type interface defined in the included file fmcxiinv.h.

GetContext

Header files

FMCXIINV.H (invocation context).

Function

- Extracts the value of the context type as specified by a passed in context name.

Interface

int GetContext

```
(void *  invContext,
 char *  invContextName,
 char ** invContextValue,
 long *  invContextValueLength)
```

Parameters

void * invContext

- Address of the whole invocation context containing all context types
- Input parameter

char * invContextName

- Name of the requested context type
- Can be one of the character string constants as listed in the *Name* column in the *Context types* table below.
- Input parameter

char ** invContextValue

- Address to which this function puts the pointer to the value of the requested context type
- Input/output parameter

long * invContextValueLength

- Address to which this function puts the length of the value of the requested context type
- Input/output parameter

Return Codes

FMC_INV_CTX_ON
Context successfully retrieved

FMC_INV_CTX_NOT_SET
Context has not been set

FMC_INV_CTX_NOT_DEFINED
Context not found

Table 15. Context types

Type	Meaning	Used by	Name
Workflow	contains Workflow context	internal use only	FMC_WORKFLOW_CTX
Security	contains the user identification the request should be executed for	invocation exit	FMC_SECURITY_CTX
Transaction	contains indication that the request should be executed in transactional context	invocation exit	FMC_TRANSACT_CTX
Performance	contains WLM enclave token, if the PES is running WLM managed	invocation exit	FMC_PERFMGMT_CTX

To link your exit correctly you will have include definition side deck FMCH0XIC from *InstHLQ*.SFMCDSD.

Connection Parameters

The parameter **connectionParameters** of the entry point **HdlRequ** represents a character string of up to 254 printable characters. This string contains the parameters needed by the invocation to connect to the service system in order to execute an application there.

Connection parameters for synchronous invocations

For synchronous invocations these parameters and the syntax how to specify them must be defined by the developer of an invocation exit. In most cases it is recommended to use the following syntax:

```
<keyword_1>=<value_1>;<keyword_2>=<value_2>;...;<keyword_n>=<value_n>
```

where the parameters are represented by key-value pairs separated by semicolons. Nevertheless it is up to you how the syntax of the connection parameters should be for your invocation exit. Connection parameters and the syntax how to specify them must be part of the documentation on your invocation exit since an MQSeries Workflow for OS/390 administrator has to specify this string when defining this exit to the PES directory.

Connection parameters for asynchronous invocation

Because only asynchronous invocations based on MQSeries queues are supported the connection parameters for asynchronous invocations and the syntax how to specify them must be

```
QUEUEMANAGER=<queuemanagername>;INPUTQUEUE=<inputqueuename>
```

where <queuemanagername> and <inputqueuename> represent MQSeries queue manager respectively MQSeries queue and have to follow the corresponding MQSeries naming rules. These parameters define the MQSeries queue to which the request message created by an asynchronous invocation exit has to be put.

Enabling MQSeries Workflow for OS/390 to use an invocation exit

In order to use a program invocation exit the exit has to reside in a DLL which has to be available in a linklibrary used by the PES.

Note:
If there is more then one asynchronous invocation exit recognizing the same kind of reply messages, it is unpredictable which of the exits will handle a reply message. This may lead to wrong results without any chance of noticing it!

See also *MQSeries Workflow for OS/390 Customization and Administration Guide - Adding a new Invocation Type*.

Invocation exit coding example

Since an invocation exit assumes there is a special kind of service system available where an application should run there are no compiling sources provided as samples but a skeleton for synchronous invocations FMCHSIVS and one for asynchronous invocations FMCHSIVA both in *InstHLQ*.SFMCSRC written in C syntax.

Part 6. Using the MQSeries Workflow APIs

Chapter 30. Using the MQSeries Workflow Runtime API

Overview of the Runtime API

There are various tasks which you typically want to address by writing an MQSeries Workflow application program:

- You can write a client application in order to:
  - Manage process instances
  - Handle worklists and/or work items
  - Administrate process instances or work items
  - Monitor the progress of execution
- You can write a program that implements an activity or support tool in your workflow process (model).

These programs typically use only a subset of the MQSeries Workflow API. For example, an activity implementation typically only accesses its containers, that is, only uses the so-called "Container API". The MQSeries Workflow API and its header files and library structures takes this fact into account (see “Compiling and linking” on page 12).

In order to ask for Runtime services, a communication must be established between the client application and an MQSeries Workflow execution server.

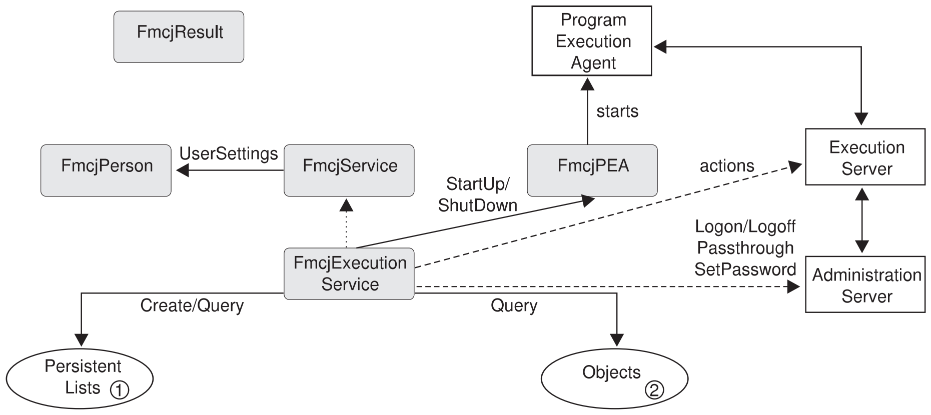


Figure 16. Setting up client/server communication. Legend: --▸ Inheritance (C++); ⟶ provides for access; — ⟶ sends messages to

As a first step, an FmcjExecutionService object must be constructed/allocated. An FmcjExecutionService object represents a session between a user and an MQSeries Workflow execution server. It essentially provides the basic functions/subprograms to set up a communication path to the specified MQSeries Workflow execution server and to establish the user session (Logon() respectively Passthrough()), and finish it (Logoff()). To log on, not only the execution server but also the administration server must be up and running so that authentication can be done. This is, however, transparent to you.

When the session to an execution server has been established, you can:

- Query objects for which you are authorized: process templates, process instances, items (work items, activity instance notifications, process instance notifications), or lists containing such objects.
- Create persistent lists, that is, persistent views on objects contained in the MQSeries Workflow database.
- Query information about the logged-on user or change that user's password.
- Start up respectively shut down a program execution agent associated to the logged-on user. This becomes necessary when work items are to be executed by MQSeries Workflow specific means.

All function/subprogram calls update a so-called result object. Detailed information about an erroneous request can be obtained from there. See "Chapter 4. Handling errors" on page 15 for more information.

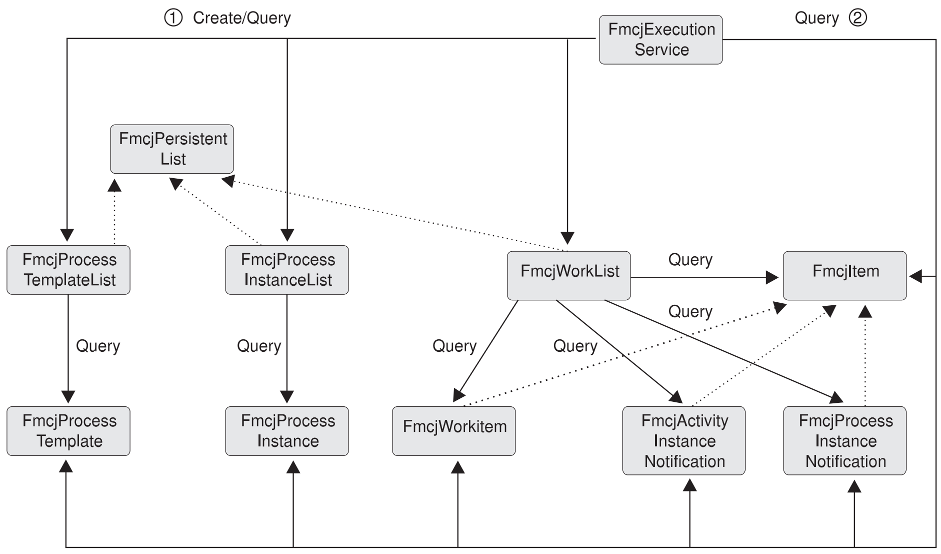


Figure 17. Querying objects. Legend: --▸ Inheritance (C++); ⟶ provides for access

When the session to an execution server has been established, you can create or query persistent lists (process template lists, process instance lists, worklists) or query other objects for which you are authorized. Note that in Runtime you can retrieve the currently valid version of a process template only; you cannot see any future or past versions.

A persistent list represents a set of objects the user is authorized for. It is a view on those objects. All objects which are accessible through the list have the same characteristics. These characteristics are specified by a filter. For example, depending on the filter specified, a worklist can contain a set of work items only. No activity instance notifications or process instance notifications are accessible through that list. The worklist content, the work items, can be queried and their attributes can be accessed. As soon as a work item has been read from the execution server, further actions can be called, for example, starting a work item.

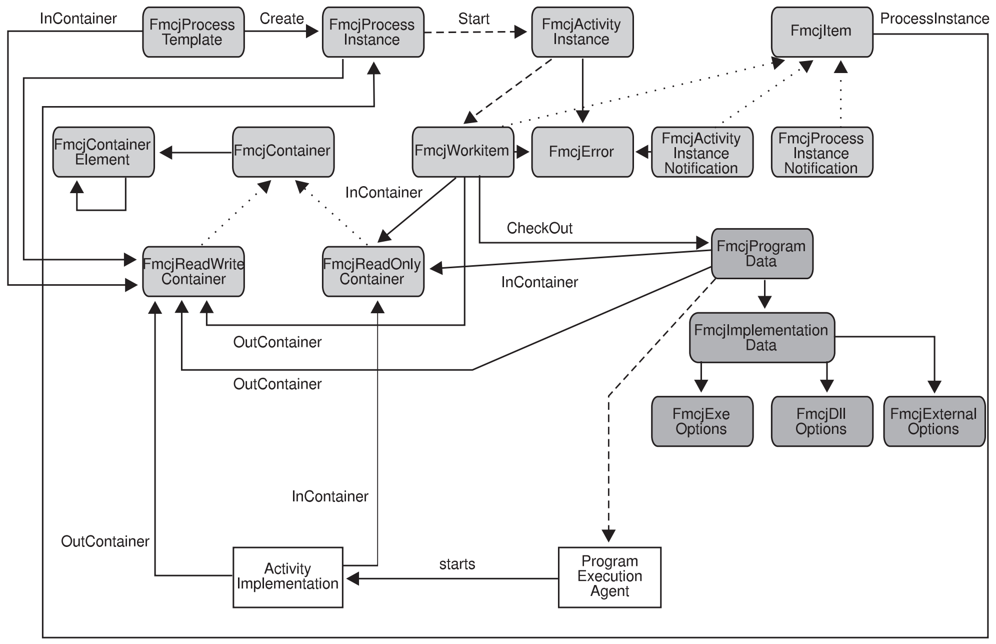


Figure 18. Dealing with process instances and (work) items. Legend: --▸ Inheritance (C++); ⟶ provides for access; — ⟶ data is passed to or results in

When (a valid version of) a process template has been retrieved, a process instance can be created and started. Starting a process instance can require input data. You can use the container functions/subprograms for reading and writing values. See "Chapter 9. Handling Containers" on page 39 for more information.

Starting a process instance triggers the scheduling of activity instances and, as a result of that, the creation of a set of work items and possibly activity instance notifications or process instance notifications when they are not worked on in time. A work item implemented by a program can then be executed either by MQSeries Workflow-specific means or by user-specific means.

When executed by user-specific means, the work item is to be checked out. Checking out provides for all information needed to execute the underlying program, the program data and its description of the implementing options and the input container data.

When executed by MQSeries Workflow-specific means, that program data is automatically sent to the program execution agent which starts the appropriate activity implementation. The activity implementation can then access its input and output containers via an appropriate request to the program execution agent. The same container accessor functions/subprograms are applicable whether called from a client application program or from an activity implementation program.

When a work item and thus the associated activity instance has not been executed successfully, the FmcjError object provides for analyzing the cause of the state InError.

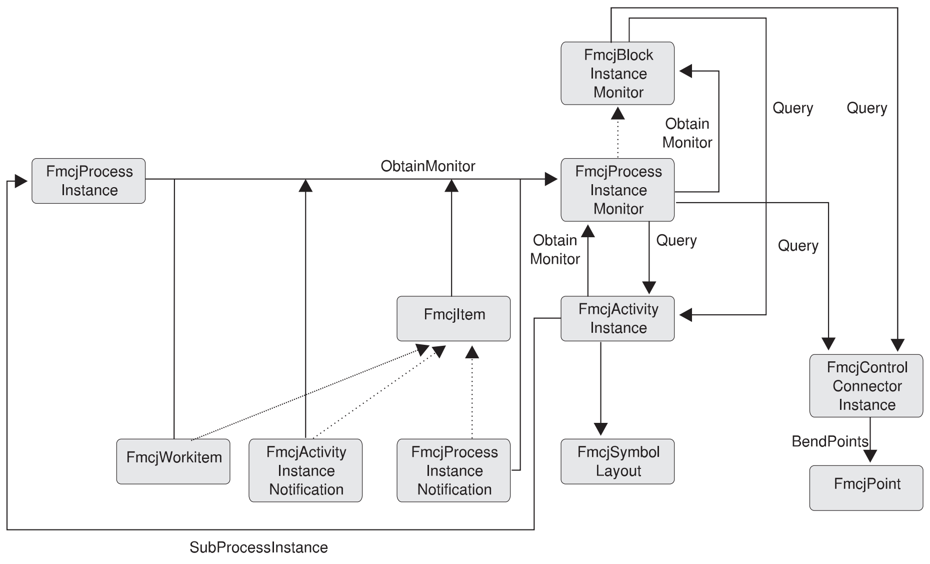


Figure 19. Monitoring a process instance. Legend: --▸ Inheritance (C++); ⟶ provides for access

When a process instance or item, that is, a work item, an activity instance notification, or a process instance notification, has been retrieved, you can obtain the associated process instance monitor. The process instance monitor then allows for analyzing the states of activity instances and control connector instances. The path taken through the process instance can thus be determined. In case you want to present this information graphically, the activity instance symbol layout and the control connector instance positions and bend points offer support.

Once a process instance monitor has been obtained, you can iterate into the process model by obtaining block instance monitors for activities of type Block or process instance monitors for activities of type Process, that is, for subprocess instances. See “Chapter 10. Monitoring a process instance” on page 47 for more information.

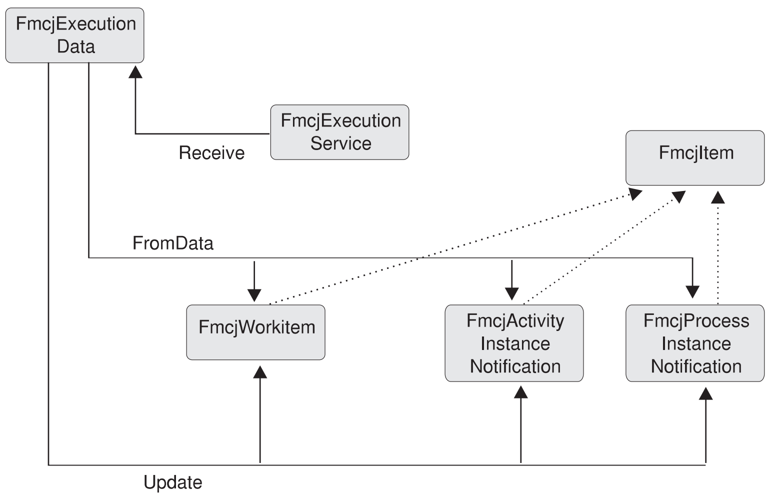


Figure 20. Handling data sent by an MQSeries Workflow server. Legend: --▸ Inheritance (C++); ⟶ provides for access

When the process setting specifies a *push refresh policy*, then the MQSeries Workflow execution server pushes changes on work items or notifications to a present client. A client application should then set up a means in order to receive such execution data. Once received, the appropriate item can be updated, created, or deleted depending on the information sent. See "The push data access model" on page 23 for more information.

API classes/objects

An alphabetical list of C++ classes respectively a list of function prefixes in the C language follows. All functions/subprograms following this list are valid calls on the respective C++ object. To become valid C language function calls, they are to be prefixed by the respective scope name. For example, if you have access to a work item whose C++ object name is wi or whose C language handle is wi, then a valid C++ call is wi.Start() and the corresponding C language call is FmcjWorkitemStart(wi).

The class and function names in Cobol have in several cases been shortened due to the 30 characters per word limit, see Table 7 on page 88.):

Class/Object	Description
FmcjActivityInstance	An instance of a workflow process template activity.
FmcjActivityInstanceNotification	A notification associated with an activity instance.
FmcjActivityInstanceNotificationVector	The C language result of a query for activity instance notifications.
FmcjActivityInstanceVector	The C language result of a query for activity instances.
FmcjBlockInstanceMonitor	The monitor for an activity instance of kind Block.
FmcjContainer	The data container of a work item or a process instance.
FmcjContainerElement	An element of a data container.
FmcjContainerElementVector	The C language result of a query for container elements.
FmcjControlConnectorInstance	The instance of a control connector between two activities.
FmcjControlConnectorInstanceVector	The result of a query for control connector instances.
FmcjCDateTime	FmcjCDateTime is the representation of date and time values.
FmcjDllOptions	The program implementation definitions for a dynamic link library.
FmcjError	Describes the cause of a state InError.
FmcjExecutionData	Information pushed by an MQSeries Workflow execution server.
FmcjExecutionService	The representation of a session between a user and an MQSeries Workflow exection server so that services can be requested.
FmcjExeOptions	The program implementation definitions for an executable.
FmcjExternalOptions	The program implementation definitions for an external service.

Class/Object	Description
FmcjGlobal	A means to group functions/subprograms which are global API functions/subprograms.
FmcjImplementationData	The program implementation definitions.
FmcjItem	An item associated to a user; can be a work item or notification.
FmcjItemVector	The C language result of a query for items.
FmcjMessage	A means to request an NLS regarding formatted message for a known message ID.
FmcjPersistentList	A list definition stored persistently.
FmcjPerson	User-specific settings for the user logged on to an MQSeries Workflow execution server.
FmcjPerson	User-specific settings for the user logged on to an MQSeries Workflow execution server.
FmcjPoint	Describes the bend points of a control connector instance.
FmcjPointVector	The result of a query for bend points.
FmcjProcessInstance	An instance of a workflow process template.
FmcjProcessInstanceList	A list to group process instances.
FmcjProcessInstanceListVector	The C language result of a query for process instance lists.
FmcjProcessInstanceMonitor	The monitor for a process instance.
FmcjProcessInstanceNotification	A notification associated with a process instance.
FmcjProcessInstanceNotificationVector	The C language result of a query for process instance notifications.
FmcjProcessInstanceVector	The C language result of a query for process instance notifications.
FmcjProcessTemplate	A workflow process template consisting of activities and containers and their control and data flow.
FmcjProcessTemplateList	A list to group process templates.
FmcjProcessTemplateListVector	The C language result of a query for process template lists.
FmcjProcessTemplateVector	The C language result of a query for process templates.
FmcjProgramData	The program definitions of an activity implementation.
FmcjReadOnlyContainer	A data container that can only be read.
FmcjReadWriteContainer	A data container that can be read and written to.
FmcjResult	The detailed result of a request.
FmcjService	Provides for common aspects of MQSeries Workflow services.
FmcjStringVector	The C language result of a query resulting in a list of strings or the C language means of providing a list of strings.
FmcjSymbolLayout	Describes the graphical layout of an activity instance.

Class/Object	Description
FmcjWorkitem	A user assigned activity instance to be worked on.
FmcjWorkitemVector	The C language result of a query for work items.
FmcjWorklist	A list to group work items or notifications.
FmcjWorklistVector	The C language result of a query for worklists.

FmcjActivityInstance

An activity instance represents an instance of a process template activity.

Refer to “Basic functions/subprograms” on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Copy()	Allocates and initializes the storage for an activity instance object by copying.
Deallocate()	Deallocates the storage for an activity instance object.
Equal()	Compares two activity instances.
IsComplete()	Indicates whether the complete activity instance information is available.
Kind()	States the kind of the activity instance, whether it is a program, a process, or a block.

Refer to “Accessor function/subprograms” on page 62 for detailed descriptions of accessor functions/subprograms.

Note: The value in the **Set** column shows if this attribute is a primary attribute (P) and set immediately when activity instances are queried or if this attribute is a secondary attribute (S) and set only after the refresh of a specific activity instance.

Accessor methods	Set	Description
ActivationTime()	P	Returns the activation time of the activity instance.
ActivationTimeIsNull()	P	Indicates whether an activation time is set.
Category()	P	Returns the process category of the activity instance.
CategoryIsNull()	P	Indicates whether a category is set.
Description()	P	Returns the description of the activity instance.
DescriptionIsNull()	P	Indicates whether a description is set.
Documentation()	S	Returns the documentation of the activity instance.
DocumentationIsNull()	S	Indicates whether a documentation is set.

Accessor methods	Set	Description
EndTime()	S	Returns the ending time of the activity instance.
EndTimeIsNull()	S	Indicates whether an end time is set.
ErrorReason()	S	Returns an error object describing the reason why the activity instance is in state InError.
ErrorReasonIsNull()	S	Indicates whether an error reason is set.
ExitCondition()	S	Returns the exit condition of the activity instance.
FirstNotificationTime()	S	Returns the time the first notification for the activity instance is to occur or has occurred.
FirstNotificationTimeIsNull()	S	Indicates whether a first notification time is set.
FirstNotifiedPersons()	S	Returns the persons who received a first notification for the activity instance.
FullName()	P	Returns the fully qualified name of the activity instance (dot notation).
Icon()	P	Returns the icon associated with the activity instance.
Implementation()	P	Returns the name of the implementing program of the activity instance.
ImplementationIsNull()	P	Indicates whether an implementation is set.
InContainerName()	S	Returns the name of the input container of the activity instance.
LastModificationTime()	P	Returns the last time a primary attribute of the activity instance was changed.
LastStateChangeTime()	P	Returns the last time the state of the activity instance changed.
ManualExitMode()	S	Returns whether the exit mode of the activity instance is manual.
ManualStartMode()	S	Returns whether the start mode of the activity instance is manual.
Name()	P	Returns the name of the activity instance.
OutContainerName()	S	Returns the name of the output container of the activity instance.
PersistentOid()	P	Returns a representation of the object identification of the activity instance.
Priority()	P	Returns the priority of the activity instance.

Accessor methods	Set	Description
PriorityIsNull()	P	Indicates whether a priority is set.
ProcessAdmin()	S	Returns the process administrator of the activity instance.
ProcessAdminIsNull()	S	Indicates whether a process administrator is set.
ProcessInstanceName()	P	Returns the name of the process instance the activity instance is part of.
ProcessInstanceState()	P	Returns the state of the process instance the activity instance is part of.
ProcessInstanceSystemGroupName()	S	Returns the name of the system group of the process instance the item is part of.
ProcessInstanceSystemName()	S	Returns the name of the system of the process instance the activity instance is part of.
SecondNotificationTime()	S	Returns the time the second notification for the activity instance is to occur or has occurred.
SecondNotificationTimeIsNull()	S	Indicates whether a second notification time is set.
SecondNotifiedPersons()	S	Returns the persons who received a second notification for the activity instance.
Staff()	S	Returns all persons a work item for the activity instance has been assigned to.
StartCondition()	S	Returns the start condition of the activity instance.
Starter()	P	Returns the starter of the activity instance.
StarterIsNull()	P	Indicates whether a starter is set.
StartTime()	P	Returns the start time of the activity instance.
StartTimeIsNull()	P	Indicates whether a start time is set.
State	P	Returns the state of the activity instance.
StateOfNotification()	S	Returns the notification state of the activity instance.
SupportTools()	P	Returns the support tools associated with the activity instance.
SupportToolsIsNull()	P	Indicates whether support tools are set.
SymbolLayout()	S	Returns the symbol layout of the activity instance.

Refer to “Action functions/subprograms” on page 76 for detailed descriptions of action functions/subprograms.

Action methods	Description	Page
ObtainProcessInstanceMonitor()	Retrieves the process instance monitor for the process instance the activity instance is part of.	215
SubProcessInstance()	Retrieves the process instance implementing the activity instance of type Process.	217

FmcjActivityInstanceNotification

An activity instance notification represents a notification for an activity instance. All functions/subprograms of FmcjItem are also applicable to activity instance notifications.

Refer to “Basic functions/subprograms” on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Copy()	Allocates and initializes the storage for an activity instance notification object by copying.
Deallocate()	Deallocates the storage for an activity instance notification object.
Kind()	States that the object is an activity instance notification.
Equal()	Compares two activity instance notifications.

Refer to “Accessor function/subprograms” on page 62 for detailed descriptions of accessor functions/subprograms.

Note: The value in the **Set** column shows if this attribute is a primary attribute (P) and set immediately when activity instance notifications are queried or if this attribute is a secondary attribute (S) and set only after the refresh of a specific activity instance notification.

Accessor methods	Set	Description
ActivityKind()	P	Returns the kind of the associated activity instance, whether it is a program or process and so on.
ErrorReason()	S	Returns an error object describing the reason why teh associated activity instacne is in state InError.
ErrorReasonIsNull()	S	Indicates whether an error reason is set.
ExitCondition()	S	Returns the exit condition of the associated activity instance.
Expired()	P	Returns whether the associated activity instance has been started and is expired now.
FirstNotificationTime()	S	Returns the first notification time of the activity instance, that is, the time when this notification has been created.
Implementation()	P	Returns the implementing program or process name of the associated activity instance.
ImplementationIsNull()	P	Indicates whether an implementation is set.

Accessor methods	Set	Description
ManualExitMode()	S	Returns whether the exit mode of the associated activity instance is manual.
ManualStartMode()	S	Returns whether the start mode of the associated activity instance is manual.
Priority()	P	Returns the priority of the associated activity instance.
SecondNotificationTime()	S	Returns the second notification time of the associated activity instance.
SecondNotificationTimeIsNull()	S	Indicates whether a second notification time is set.
Staff()	S	Returns all persons a work item for the associated activity instance has been assigned to.
StartCondition()	S	Returns the start condition of the associated activity instance.
StartOverdue()	P	Returns whether the start of the associated activity instance is overdue.
State	P	Returns the state of the associated activity instance.
StateOfNotification()	S	Returns the notification state of the associated activity instance.
SupportTools()	P	Returns the support tools associated with the activity instance.
SupportToolsIsNull()	P	Indicates whether support tools are set.

Refer to “Action functions/subprograms” on page 76 for detailed descriptions of action functions/subprograms.

Action methods	Description	Page
PersistentObject()	Retrieves the specified activity instance notification.	221

FmcjActivityInstanceNotificationVector

An activity instance notification vector represents the result of a query for activity instance notifications.

Refer to “Handling collections” on page 30 for detailed descriptions of vector functions.

Vector methods	Description
Deallocate()	Deallocates an activity instance notification vector object.
FirstElement()	Returns the first element of the activity instance notification vector.
NextElement()	Returns the next element of the activity instance notification vector.
Size()	Returns the number of elements in the activity instance notification vector.

FmcjActivityInstanceVector

An activity instance vector represents the result of a query for activity instances.

Refer to “Handling collections” on page 30 for detailed descriptions of vector access functions.

Accessor methods	Description
Deallocate()	Deallocates an activity instance vector object.
FirstElement()	Returns the first element of the activity instance vector.
NextElement()	Returns the next element of the activity instance vector.
Size()	Returns the number of elements in the activity instance vector.

FmcjBlockInstanceMonitor

A block instance monitor object represents a monitor of an activity instance of type *Block*. All functions/subprograms of a block instance monitor are also applicable to process instance monitors.

Refer to “Accessor function/subprograms” on page 62 for detailed descriptions of accessor functions/subprograms. All properties are primary because a block instance monitor is a part of a process instance monitor.

Accessor methods	Description
ActivityInstances()	Returns the activity instances which are represented by the block instance monitor, that is, which are part of the activity instance of type *Block*.
ControlConnectorInstances()	Returns the control connector instances which are represented by the block instance monitor, that is, which are part of the activity instance of type *Block*.

Refer to “Action functions/subprograms” on page 76 for detailed descriptions of action functions/subprograms.

Action methods	Description	Page
ObtainBlockInstanceMonitor()	Returns the block instance monitor for an activity instance of type *Block*. The activity instance is part of the set of activity instances represented by the block instance monitor.	225
ObtainProcessInstanceMonitor()	Returns the process instance monitor for an activity instance of type *Process*. The activity instance is part of the set of activity instances represented by the block instance monitor.	227
Refresh()	Refreshes the block instance monitor from the MQSeries Workflow execution server.	229

FmcjContainer

A container represents an input or output data container of a process instance or work item. All functions/subprograms of a container are applicable to read-only and read/write containers.

Refer to "Accessor function/subprograms" on page 62 for detailed descriptions of accessor functions/subprograms.

Accessor methods	Description
AllLeafCount()	Returns the number of leaf elements of the container including the MQSeries Workflow predefined members.
AllLeaves()	Returns all leaf elements of the container including the MQSeries Workflow predefined members.
ArrayBinaryLength()	Returns the length of the value of the specified container leaf element in the C language. The leaf is part of an array and of type BINARY.
ArrayBinaryValue()	Returns the value of the specified container leaf element in the C language. The leaf is part of an array and of type BINARY.
ArrayFloatValue()	Returns the value of the specified container leaf element in the C language. The leaf is part of an array and of type FLOAT.
ArrayLongValue()	Returns the value of the specified container leaf element in the C language. The leaf is part of an array and of type LONG.
ArrayStringLength()	Returns the length of the value of the specified container leaf element in the C language. The leaf is part of an array and of type STRING.
ArrayStringValue()	Returns the value of the specified container leaf element in the C language. The leaf is part of an array and of type STRING.
BinaryLength()	Returns the length of the value of the specified container leaf element in the C language. The leaf is of type BINARY.
BinaryValue()	Returns the value of the specified container leaf element in the C language. The leaf is of type BINARY.
FloatValue()	Returns the value of the specified container leaf element in the C language. The leaf is of type FLOAT.
GetElement()	Provides access to a container element.
LeafCount()	Returns the number of user-defined leaf elements of the container.
Leaves()	Returns all user-defined leaf elements of the container.
LongValue()	Returns the value of the specified container leaf element in the C language. The leaf is of type LONG.
MemberCount()	Returns the number of structural members in the container.
StringLength()	Returns the length of the value of the specified container leaf element in the C language. The leaf is of type STRING.
StringValue()	Returns the value of the specified container leaf element in the C language. The leaf is of type STRING.
StructMembers()	Returns the structural members of the container.
Type()	Returns the type of the container, that is, the data structure name.

Activity implementation methods	Description	Page
InContainer()	Accesses the input container from within an activity implementation.	233
OutContainer()	Accesses the output container from within an activity implementation.	235
RemoteInContainer()	Accesses the input container from within a program started by an activity implementation.	236
RemoteOutContainer()	Accesses the output container from within a program started by an activity implementation.	238
SetRemoteOutContainer()	Sets the output container from within a program started by an activity implementation.	240
SetOutContainer()	Sets the output container from within an activity implementation.	242

FmcjContainerElement

A container element represents an arbitrary element of a container.

Refer to “Basic functions/subprograms” on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Copy()	Allocates and initializes the storage for a container element object by copying.
Deallocate()	Deallocates the storage for a container element object.
Equal()	Compares two container elements.

Refer to “Accessor function/subprograms” on page 62 for detailed descriptions of accessor functions/subprograms.

Accessor methods	Description
ArrayBinaryLength()	Returns the length of the value of the specified container element leaf element in the C language. The leaf is part of an array and of type BINARY.
ArrayBinaryValue()	Returns the value of the specified container element leaf element in the C language. The leaf is part of an array and of type BINARY.
ArrayElements()	Returns the array elements of the container element.
ArrayFloatValue()	Returns the value of the specified container element leaf element in the C language. The leaf is part of an array and of type FLOAT.
ArrayLongValue()	Returns the value of the specified container element leaf element in the C language. The leaf is part of an array and of type LONG.
ArrayStringLength()	Returns the length of the value of the specified container element leaf element in the C language. The leaf is part of an array and of type STRING.
ArrayStringValue()	Returns the value of the specified container element leaf element in the C language. The leaf is part of an array and of type STRING.

Accessor methods	Description
BinaryLength()	Returns the length of the value of the specified container element leaf element in the C language. The leaf is of type BINARY.
BinaryValue()	Returns the value of the specified container element leaf element in the C language. The leaf is of type BINARY.
Cardinality()	Returns the number of array elements of the container element.
FloatValue()	Returns the value of the specified container element leaf element in the C language. The leaf is of type FLOAT.
FullName()	Returns the fully-qualified dotted name of the container element.
GetElement()	Provides access to an element of the container element.
IsArray()	Indicates whether the container element is an array.
IsLeaf()	Indicates whether the container element is a leaf.
IsStruct()	Indicates whether the container element is a structure itself.
LeafCount()	Returns the number of leaf elements of the container element.
Leaves()	Returns all leaf elements of the container element.
LongValue()	Returns the value of the specified container element leaf element in the C language. The leaf is of type LONG.
MemberCount()	Returns the number of structural members in the container element.
Name()	Returns the name of the container element.
StringLength()	Returns the length of the value of the specified container element leaf element in the C language. The leaf is of type STRING.
StringValue()	Returns the value of the specified container element leaf element in the C language. The leaf is of type STRING.
StructMembers()	Returns the structural members of the container element.
Type()	Returns the type of the container element, that is, the data structure name.

FmcjContainerElementVector

A container element vector represents the result of a query for container elements in the C language.

Refer to “Handling collections” on page 30 for detailed descriptions of vector functions.

Vector methods	Description
Deallocate()	Deallocates a container element vector object.
FirstElement()	Returns the first element of the container element vector.
NextElement()	Returns the next element of the container element vector.
Size()	Returns the number of elements in the container element vector.

FmcjControlConnectorInstance

A control connector instance object represents a control connector between two activity instances and its state.

Refer to “Basic functions/subprograms” on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Copy()	Allocates and initializes the storage for a control connector instance object by copying.
Deallocate()	Deallocates the storage for a control connector instance object.
Equal()	Compares two control connector instance objects on the basis of their source and target activity instances.
Kind()	States the kind of the control connector instance, whether it is a transition condition or the "otherwise" connector.

Refer to “Accessor function/subprograms” on page 62 for detailed descriptions of accessor functions/subprograms. All properties are primary properties.

Accessor methods	Description
BendPoints()	Returns the bend points of the control connector instance.
Name()	Returns the name associated with the control connector instance.
NameIsNull()	Indicates whether a name is set.
PersistentOidOfSource()	Returns the object ID of the activity instance which is the source of this control connector instance.
PersistentOidOfTarget()	Returns the object ID of the activity instance which is the target of this control connector instance.
State()	Returns the state of the control connector instance, whether it is evaluated, and the result of evaluation.
TransitionCondition()	Returns the transition condition of the control connector instance.
TransitionConditionIsNull()	Indicates whether a transition condition is set.

FmcjControlConnectorInstanceVector

A control connector instance vector represents the result of a query for control connector instances in the C language.

Refer to “Handling collections” on page 30 for detailed descriptions of vector access functions.

Accessor methods	Description
Deallocate()	Deallocates a control connector instance vector object.
FirstElement()	Returns the first element of the control connector instance vector.

Accessor methods	Description
NextElement()	Returns the next element of the control connector instance vector.
Size()	Returns the number of elements in the control connector instance vector.

FmcjCDateTime

An FmcjCDateTime structure represents date and time values in the C and Cobol language.

Accessor functions	Description
FmcjDateTimeAsString	Returns the string representation of the date/time structure.
FmcjDateTimeCurrentTime	Returns the current date/time.
FmcjDateTimeIsValid	Indicates whether the passed date/time is a valid date/time.

FmcjDllOptions

A DllOptions object represents the program implementation definitions for a dynamic link library. These program implementations can not be executed on the OS/390 version of MQSeries.

Refer to “Basic functions/subprograms” on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Copy()	Allocates and initializes the storage for a DLL options object by copying.
Deallocate()	Deallocates the storage for a DLL options object.

Refer to “Accessor function/subprograms” on page 62 for detailed descriptions of accessor functions/subprograms.

Accessor methods	Description
EntryPointName()	Returns the name of the entry point of the DLL.
ExecuteFenced()	States whether the DLL should run in a separate address space.
ExecuteFencedIsNull()	Indicates whether execute fended is set.
KeepLoaded()	States whether the DLL should stay loaded.
KeepLoadedIsNull()	Indicates whether keep loaded is set.
PathAndFileName()	Returns the path and file name of the DLL.

FmcjError

An Error object represents a description of the reason why a work item is in state InError.

Refer to “Chapter 12. Function/subprogram types” on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
constructor()	Constructs an Error object.
Copy()	Constructs an activity instance object by copying.
Deallocate()	Destructs an activity instance object.
Equal()	Compares two activity instance objects.

Refer to “Accessor function/subprograms” on page 62 for detailed descriptions of accessor functions/subprograms. All properties are primary properties.

Accessor methods	Description
MessageText()	Returns the error as an NLS regarding formatted message.
Parameters()	Returns the parameters of the error; these are to be incorporated into the message text.
Rc()	Returns the return code remembered in the Error object.

FmcjExecutionData

An execution data object represents data sent from an MQSeries Workflow execution server.

Refer to “Basic functions/subprograms” on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Copy()	Allocates and initializes the storage for an execution data object by copying.
Deallocate()	Deallocates the storage for an execution data object.
Kind()	Returns the kind of the data, whether it is describing a work item creation, deletion, and so on.

Refer to “Accessor function/subprograms” on page 62 for detailed descriptions of accessor functions/subprograms. All properties are primary properties.

Accessor methods	Description
ActivityInstanceNotificationFromData()	Creates an activity instance notification from the execution data.
PersistentOid()	Returns a representation of the object ID of the object described by the execution data.
ProcessInstanceNotificationFromData()	Creates a process instance notification from the execution data.
WorkitemFromData()	Creates a work item from the execution data.

FmcjExecutionService

An execution service object represents a user session to an execution server. All functions/subprograms provided by FmcjService are also applicable.

Refer to “Basic functions/subprograms” on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Allocate()	Allocates the storage for an execution service object. The execution service to connect to is taken from the MQSeries Workflow user's or workstation profile.
AllocateForSystem()	Allocates the storage for the specified execution service object.
Copy()	Allocates and initializes the storage for an execution service object by copying.
Deallocate()	Deallocates the storage for an execution service object.
Equal()	Compares two execution service objects.

Action methods	Description	Page
CreateProcessInstanceList()	Creates a new process instance list on the execution server.	248
CreateProcessTemplateList()	Creates a new process template list on the execution server.	254
CreateWorklist()	Creates a new worklist on the execution server.	259
Logoff()	Logs off from the connected execution server.	266
Logon()	Logs on to the execution server.	268
QueryActivityInstanceNotifications()	Retrieves the activity instance notifications the logged-on user has access to.	272
QueryItems()	Retrieves the work items or notifications the logged-on user has access to.	277
QueryProcessInstanceLists()	Retrieves the process instance lists the logged-on user has access to.	283
QueryProcessInstanceNotifications()	Retrieves the process instance notifications the logged-on user has access to.	285
QueryProcessInstances()	Retrieves the process instances the logged-on user has access to.	291
QueryProcessTemplateLists()	Retrieves the process template lists the logged-on user has access to.	296
QueryProcessTemplates()	Retrieves the process templates the logged-on user has access to.	298
QueryWorkitems()	Retrieves the work items the logged-on user has access to.	302
QueryWorklists()	Retrieves the worklists the logged-on user has access to.	309
Receive()	Receives execution data sent by an MQSeries Workflow execution server.	311
TerminateReceive()	Places information in the client input queue to indicate that receiving execution data sent by an MQSeries Workflow execution server can end.	315

Activity implementation methods	Description	Page
Passthrough()	Establishes a session between an activity implementation and an execution server.	270

Activity implementation methods	Description	Page
RemotePassthrough()	Establishes a session between a program started by an activity implementation and an execution server.	313

FmcjExeOptions

An ExeOptions object represents the program implementation definitions for an executable. These program implementations can not be executed on the OS/390 version of MQSeries.

Refer to “Basic functions/subprograms” on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Copy()	Allocates and initializes the storage for an EXE options object by copying.
Deallocate()	Deallocates the storage for an EXE options object.

Refer to “Accessor function/subprograms” on page 62 for detailed descriptions of accessor functions/subprograms.

Accessor methods	Description
AutomaticClose()	States whether the window in which the EXE starts should close when the EXE ends.
AutomaticCloseIsNull()	Indicates whether automatic close is set.
Environment()	States the environment settings for the EXE.
EnvironmentIsNull()	Indicates whether an environment is set.
InheritEnvironment()	States whether the environment settings should be merged with the operating system environment settings.
PathAndFileName()	Returns the path and file name of the EXE.
RunInXTerm()	States whether the EXE should start in a separate xterm.
RunInXTermIsNull()	Indicates whether run in xterm is set.
StartInForeGround()	States whether the EXE should start in the foreground.
StartInForeGroundIsNull()	Indicates whether start in foreground is set.
WindowStyle()	States the initial window style.
WindowStyleIsNull()	Indicates whether a window style is set.
WorkingDirectoryName()	States the working directory for the EXE.
WorkingDirectoryNameIsNull()	Indicates whether a working directory is set.

FmcjExternalOptions

An ExternalOptions object represents the program implementation definitions for an external service.

Refer to “Basic functions/subprograms” on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
constructor()	Constructs an ExternalOptions object.
Copy()	Allocates and initializes the storage for an EXE options object by copying.
Deallocate()	Deallocates the storage for an EXE options object.

Refer to “Accessor function/subprograms” on page 62 for detailed descriptions of accessor functions/subprograms. All properties are primary properties.

Accessor methods	Description
BackwardMappingFormat()	Specifies the format of the mapping from the structure the executable uses to an MQSeries Workflow container.
BackwardMappingFormatISNull()	Indicates whether a backward mapping format is set.
BackwardMappingParameters()	Returns backward mapping parameters, if any.
BackwardMappingParameterIsNull()	Indicates whether backward mapping parameters are set.
Codepage()	Specifies the code page of the service.
CodepageIsNull()	Indicates whether a code page is set.
ExecutableName()	Specifies the executable to be invoked by the invocation type and service.
ExecutableType()	Identifies the type of executable.
ForwardMappingFormat()	Specifies the format for the mapping from an MQSeries Workflow container to the structure the executable uses.
ForwardMappingFormatIsNull()	Indicates whether a forward mapping format is set.
ForwardMappingParameters()	Returns forward mapping parameters, if any.
ForwardMappingParameterIsNull()	Indicates whether forward mapping parameters are set.
InvocationType()	Specifies the invocation mechanism to invoke the executable on the service.
IsLocalUser()	Returns whether a local user is to be resolved insteadof using the MQSeries Workflow user ID.
IsMappingRoutineCall()	Specifies wheteher forward or backward mapping routines are to be called.
IsSecurityRoutineCall()	Specifies whether a security routine is to be called.
Mapping Type()	Identifies the type of mapping that should occur.
Mapping TypeIsNull()	Identifies whether a mapping type is set.
ServiceName()	Identifies the service that is to be called.
ServiceType()	Identifies the type of service to be called, for example CICS and IMS.
TimeoutPeriod()	Specifies a timeout duration.
TimeoutIntervall()	Specifies how long the program execution agent or server should wait for response from the started service, forever, a time period or never.
TimeoutIntervallIsNull()	Indicates whether a timeout intervall is set.

FmcjGlobal

An API global object serves to group global MQSeries Workflow API functions/subprograms.

Refer to “Basic functions/subprograms” on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Connect()	Initializes the API in the current thread.
Disconnect()	Deinitializes the API in the current thread.

FmcjImplementationData

An implementation data object represents the program implementation definitions.

Refer to “Basic functions/subprograms” on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Copy()	Allocates and initializes the storage for an implementation data object by copying.
Deallocate()	Deallocates the storage for an implementation data object.
Kind()	States the actual kind of the implementation data, whether it is a DLL or an EXE.

Refer to “Accessor function/subprograms” on page 62 for detailed descriptions of accessor functions/subprograms.

Accessor methods	Description
CommandLineParameters()	Returns the command line parameters to be passed to the invoked program.
CommandLineParametersIsNull()	Indicates whether command line parameters are set.
DllOptions()	Returns the description of a DLL, if the implementation is a DLL.
ExeOptions()	Returns the description of an EXE, if the implementation is an EXE.
ExternalOptions()	Returns the description of external options, if the implementation is an ExternalService.
Platform()	Returns the operating system platform this implementation data describes.

FmcjItem

An item represents a work item, an activity instance notification, or a process instance notification. This means that all functions/subprograms of an item are also applicable to work items, activity instance notifications, and process instance notifications.

Refer to “Basic functions/subprograms” on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Copy()	Allocates and initializes the storage for an item object by copying.

Basic methods	Description
Deallocate()	Deallocates the storage for an item object.
Equal()	Compares two items.
IsComplete()	Indicates whether the complete item information is available.
Kind()	States the actual kind of the item, whether it is a work item or some kind of notification.

Refer to “Accessor function/subprograms” on page 62 for detailed descriptions of accessor functions/subprograms.

Note: The value in the **Set** column shows if this attribute is a primary attribute (P) and set immediately when items are queried or if this attribute is a secondary attribute (S) and set only after the refresh of a specific item.

Accessor methods	Set	Description
Category()	P	Returns the process category of the item.
CategoryIsNull()	P	Indicates whether a category is set.
CreationTime()	P	Returns the creation time of the item.
Description()	P	Returns the description of the item.
DescriptionIsNull()	P	Indicates whether a description is set.
Documentation()	S	Returns the documentation of the item.
DocumentationIsNull()	S	Indicates whether a documentation is set.
EndTime()	S	Returns the ending time of the item.
EndTimeIsNull()	S	Indicates whether an end time is set.
Icon()	P	Returns the icon associated with the item.
InContainerName()	S	Returns the name of the î of the item.
LastModificationTime()	P	Returns the last time a primary attribute of the item was changed.
Name()	P	Returns the name of the item.
OutContainerName()	S	Returns the name of the ô of the item.
Owner()	P	Returns the owner of the item.
PersistentOid()	P	Returns a representation of the object identification of the item.
ProcessAdmin()	S	Returns the process administrator of the item.
ProcessInstanceName()	P	Returns the name of the process instance the item is part of.
ProcessInstanceState()	P	Returns the state of the process instance the item is part of.
ProcessInstanceSystemGroupName()	S	Returns the name of the system group of the process instance the item is part of.
ProcessInstanceSystemName()	S	Returns the name of the system of the process instance the item is part of.
ReceivedAs()	P	Returns the reason why the item was received.
ReceivedTime()	P	Returns the time when the item was received.
StartTime()	P	Returns the start time of the item.
StartTimeIsNull()	P	Indicates whether a start time is set.

Mutator methods	Description	Page
Update()	Updates the item with the execution data sent by an MQSeries Workflow execution server. The object IDs of the item and of the object described by the execution data must match.	70

Refer to “Action functions/subprograms” on page 76 for detailed descriptions of action functions/subprograms.

Action methods	Description	Page
Delete()	Deletes an item.	317
ObtainProcessInstanceMonitor()	Retrieves the process instance monitor for the process instance the item is part of.	319
ProcessInstance()	Retrieves the process instance the item is part of.	321
Refresh()	Retrieves the complete information of the item.	323
SetDescription()	Sets the description of the item.	324
SetName()	Sets the name of the item.	326
Transfer()	Transfers an item to the specified user.	328

FmcjItemVector

An item vector represents the result of a query for items in the C language.

Refer to “Handling collections” on page 30 for detailed descriptions of vector access functions.

Accessor methods	Description
Deallocate()	Deallocates an item vector object.
FirstElement()	Returns the first element of the item vector.
NextElement()	Returns the next element of the item vector.
Size()	Returns the number of elements in the item vector.

FmcjMessage

A message object serves to access the MQSeries Workflow provided message catalog.

Refer to “Accessor function/subprograms” on page 62 for detailed descriptions of accessor functions/subprograms.

Accessor methods	Description
MessageText()	Returns an NLS regarding formatted message based on the message ID and the parameters passed.

FmcjPersistentList

A persistent list represents a persistent list definition. All functions/subprograms of a persistent list are also applicable to process instance lists, process template lists, and worklists.

Refer to "Accessor function/subprograms" on page 62 for detailed descriptions of accessor functions/subprograms.

Accessor methods	Description
Description()	Returns the description of the persistent list.
DescriptionIsNull()	Indicates whether a description is set.
Filter()	Returns the filter of the persistent list.
FilterIsNull()	Indicates whether a filter is set.
Name()	Returns the name of the persistent list.
OwnerOfList()	Returns the owner of the persistent list.
OwnerOfListIsNull()	Indicates whether an owner is set; a public list does not have an owner.
SortCriteria()	Returns the sort criteria of the persistent list.
SortCriteriaIsNull()	Indicates whether sort criteria are set.
Threshold()	Returns the threshold of the persistent list.
ThresholdIsNull()	Indicates whether a threshold is set.
Type()	Returns the type of the persistent list, whether it is a public or private list.

Action methods	Description	Page
Delete()	Deletes the persistent list.	331
Refresh()	Refreshes the persistent list.	333
SetDescription()	Sets the description of the persistent list.	334
SetFilter()	Sets the filter of the persistent list.	336
SetSortCriteria()	Sets the sort criteria of the persistent list.	338
SetThreshold()	Sets the threshold of the persistent list.	339

FmcjPerson

A person object represents the settings of the logged-on user.

Refer to "Basic functions/subprograms" on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Copy()	Allocates and initializes the storage for a person object by copying.
Deallocate()	Deallocates the storage for a person object.
Equal()	Compares two persons.
IsComplete()	Indicates whether the complete person information is available.

Refer to "Accessor function/subprograms" on page 62 for detailed descriptions of accessor functions/subprograms.

Note: The value in the **Set** column shows if this attribute is a primary attribute (P) and set immediately when persons are queried or if this attribute is a secondary attribute (S) and set only after the refresh of a specific person.

Note: The UserSettings() function/subprogram returns all attributes of a person, primary and secondary.

Accessor methods	Set	Description
CategoriesAuthorizedFor()	P	Returns the categories the person is authorized for.
CategoriesAuthorizedForAsAdmin()	P	Returns the categories the person is authorized for as administrator.
Description()	P	Returns the description of the person.
DescriptionIsNull()	P	Indicates whether a description is set.
FirstName()	P	Returns the first name of the person.
FirstNameIsNull()	P	Indicates whether a first name is set.
IsAbsent()	P	Indicates whether the person is absent.
IsAdminForCategory()	P	Indicates whether the person has administrator rights for the specified category.
IsAdministrator()	S	Indicates whether the person is an administrator.
IsAuthorizedForAllCategories()	P	Indicates whether the person is authorized for all categories.
IsAuthorizedForAllCategoriesAsAdmin()	P	Indicates whether the person is authorized for all categories as administrator.
IsAuthorizedForAllPersons()	P	Indicates whether the person is authorized to see the items of all persons.
IsAuthorizedForAuthorizationDefinition()	P	Indicates whether the person is authorized to define authorizations.
IsAuthorizedForOperationAdministration()	P	Indicates whether the person is authorized for operational administrations.
IsAuthorizedForProcessDefinition()	P	Indicates whether the person is authorized to define process models.
IsAuthorizedForStaffDefinition()	P	Indicates whether the person is authorized to define persons.
IsAuthorizedForTopologyDefinition()	P	Indicates whether the person is authorized to define topological data.
IsManager()	S	Indicates whether the person is a manager.
IsResetAbsence()	P	Indicates whether the absence flag should be reset when the person logs on.
LastName()	P	Returns the last name of the person.

Accessor methods	Set	Description
LastNameIsNull()	P	Indicates whether a last name is set.
Level()	P	Returns the level of the person.
Manager()	S	Returns the user identification of the person's manager.
ManagerIsNull()	S	Indicates whether the person's manager is set.
MiddleName()	P	Returns the middle name of the person.
MiddleNameIsNull()	P	Indicates whether a middle name is set.
NamesOfManagedOrganizations()	S	Returns the names of organizations the person manages.
NamesOfRoles()	P	Returns the names of roles the person belongs to.
NamesOfRolesToCoordinate()	S	Returns the names of roles the person can coordinate.
OrganizationName()	P	Returns the name of the organization the person belongs to.
OrganizationNameIsNull()	P	Indicates whether an organization name is set.
PersonID()	P	Returns the person ID of the person.
PersonIDIsNull()	P	Indicates whether a person ID is set.
PersonsAuthorizedFor()	P	Returns the persons this person is authorized for.
PersonsAuthorizedForMe()	S	Returns the persons which are authorized for this person.
PersonsToStandInFor()	S	Returns the persons this person stands in for.
Phone()	P	Returns the phone number of the person.
PhoneIsNull()	P	Indicates whether a phone is set.
SecondPhone()	P	Returns the alternate phone number of the person.
SecondPhoneIsNull()	P	Indicates whether an alternate phone is set.
Substitute()	P	Returns the substitute of the person.
SubstituteIsNull()	P	Indicates whether a substitute is set.
SystemName()	P	Returns the home system of the person.
UserID()	P	Returns the user identification of the person.

Action methods	Description	Page
Refresh()	Retrieves the complete information of the process instance.	343
SetAbsence()	Sets the absence indicator of the person.	344
SetSubstitute()	Sets the substitute of this person.	346

FmcjPoint

A point object represents a bend point of a control connector.

Refer to “Basic functions/subprograms” on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Copy()	Allocates and initializes the storage for a point object by copying.
Deallocate()	Deallocates the storage for a point object.
Equal()	Compares two point objects on the basis of their contents.

Refer to “Accessor function/subprograms” on page 62 for detailed descriptions of accessor functions/subprograms. All properties are primary properties.

Accessor methods	Description
XPosition()	Returns the x-coordinate of the point.
YPosition()	Returns the y-coordinate of the point.

FmcjPointVector

A point vector represents the result of a query for points.

Refer to “Handling collections” on page 30 for detailed descriptions of vector access functions.

Accessor methods	Description
Deallocate()	Deallocates a point vector object.
FirstElement()	Returns the first element of the point vector.
NextElement()	Returns the next element of the point vector.
Size()	Returns the number of elements in the point vector.

FmcjProcessInstance

A process instance object represents an instance of a workflow process template.

Refer to “Basic functions/subprograms” on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Copy()	Allocates and initializes the storage for a process instance object by copying.
Deallocate()	Deallocates the storage for a process instance object.
Equal()	Compares two process instances.
IsComplete()	Indicates whether the complete process instance information is available.

Refer to “Accessor function/subprograms” on page 62 for detailed descriptions of accessor functions/subprograms.

Note: The value in the **Set** column shows if this attribute is a primary attribute (P) and set immediately when process instances are queried or if this attribute is a secondary attribute (S) and set only after the refresh of a specific process instance.

Accessor methods	Set	Description
AuditMode()	S	Returns the audit mode of the process instance.
Category()	P	Returns the category of the process instance.
CategoryIsNull()	P	Indicates whether a category is set.
CreationTime()	S	Returns the creation time of the process instance.
Creator()	S	Returns the creator of the process instance.
Description()	P	Returns the description of the process instance.
DescriptionIsNull()	P	Indicates whether a description is set.
Documentation()	S	Returns the documentation of the process instance.
DocumentationIsNull()	S	Indicates whether a documentation is set.
EndTime()	S	Returns the end time of the process instance.
EndTimeIsNull()	S	Indicates whether an end time is set.
Icon()	P	Returns the icon associated with the process instance.
InContainerName()	S	Returns the name of the î of the process instance.
InContainerNeeded()	P	Indicates whether an input container is needed to start the process instance.
LastModificationTime()	P	Returns the last time a primary attribute of the process instance was changed.
LastStateChangeTime()	P	Returns the last time the state of the process instance was changed.
Name()	P	Returns the name of the process instance.
NotificationTime()	S	Returns the notification time of the process instance.
NotificationTimeIsNull()	S	Indicates whether a notification time is set.
NotifiedPerson()	S	Returns the person who received the notification.
NotifiedPersonIsNull()	S	Indicates whether a notified person is set.
OrganizationName()	S	Returns the name of the organization of the process instance.
OrganizationNameIsNull()	S	Indicates whether an organization name is set.
OutContainerName()	S	Returns the name of the ô of the process instance.
ParentName()	P	Returns the name of the parent process instance of this process instance.
ParentNameIsNull()	P	Indicates whether a parent name is set.
PersistentOid()	P	Returns a representation of the object identification of the process instance.
ProcessAdmin()	S	Returns the name of the process administrator of the process instance.

Accessor methods	Set	Description
ProcessAdminIsNull()	S	Indicates whether a process administrator is set.
ProcessTemplateName()	P	Returns the name of the process template the process instance is derived from.
RoleName()	S	Returns the name of the role of the process instance.
RoleNameIsNull()	S	Indicates whether a role is set.
Starter()	S	Returns the starter of the process instance.
StarterIsNull()	S	Indicates whether a starter is set.
StartTime()	S	Returns the start time of the process instance.
StartTimeIsNull()	S	Indicates whether a start time is set.
State()	P	Returns the state of the process instance.
StateOfNotification()	S	Returns the notification state of the process instance.
SuspensionExpirationTime()	P	Returns the suspension expiration time of the process instance.
SuspensionExpirationTimeIsNull()	P	Indicates whether the suspension expiration time is set.
SuspensionTime()	P	Returns the time the process instance was suspended.
SuspensionTimeIsNull()	P	Indicates whether the suspension time is set.
SystemGroupName()	P	Returns the name of the system group where the process instance runs.
SystemName()	P	Returns the name of the system where the process instance runs.
TopLevelName()	P	Returns the name of the top level process instance of this process instance.

Action methods	Description	Page
Delete()	Deletes the process instance.	349
InContainer()	Retrieves the input container of the process instance.	351
PersistentObject()	Retrieves the process instance specified by the passed object identification.	355
Refresh()	Retrieves the complete information of the process instance.	357
Resume()	Resumes the execution of a suspended process instance.	358
SetDescription()	Sets the description of the process instance.	360
SetName()	Sets the name of the process instance.	362
Start()	Starts the process instance.	364
Suspend()	Suspends the process instance.	366
SuspendUntil()	Suspends the process instance until the specified time.	366
Terminate()	Terminates the process instance.	368

FmcjProcessInstanceList

A process instance list represents a group of process instances. All functions/subprograms of a persistent list are also applicable to process instance lists.

Refer to “Basic functions/subprograms” on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Copy()	Allocates and initializes the storage for a process instance list object by copying.
Deallocate()	Deallocates the storage for a process instance list object.
Equal()	Compares two process instance lists.

Action methods	Description	Page
QueryProcessInstances()	Retrieves the process instances qualifying via the process instance list.	371

FmcjProcessInstanceListVector

A process instance list vector represents the result of a query for process instance lists.

Refer to “Handling collections” on page 30 for detailed descriptions of vector access functions.

Accessor methods	Description
Deallocate()	Deallocates a process instance list vector object.
FirstElement()	Returns the first element of the process instance list vector.
NextElement()	Returns the next element of the process instance list vector.
Size()	Returns the number of elements in the process instance list vector.

FmcjProcessInstanceMonitor

A process instance monitor object represents a monitor of a process instance. All functions/subprograms of FmcjBlockInstanceMonitor are also applicable to process instance monitors.

Refer to “Basic functions/subprograms” on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Deallocate()	Deallocates the storage for a process instance monitor object. All block instance monitors contained are also deallocated.

FmcjProcessInstanceNotification

A process instance notification represents a notification raised for a process instance. All functions/subprograms of an FmcjItem are also applicable to process instance notifications.

Refer to “Basic functions/subprograms” on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Copy()	Allocates and initializes the storage for a process instance notification object by copying.
Deallocate()	Deallocates the storage for a process instance notification object.
Kind()	States that the object is a process instance notification.
Equal()	Compares two process instance notifications.

Refer to “Action functions/subprograms” on page 76 for detailed descriptions of action functions/subprograms.

Action methods	Description	Page
PersistentObject()	Retrieves the specified process instance notification.	375

FmcjProcessInstanceNotificationVector

A process instance notification vector represents the result of a query for process instance notifications.

Refer to “Handling collections” on page 30 for detailed descriptions of vector access functions.

Accessor methods	Description
Deallocate()	Deallocates a process instance notification vector object.
FirstElement()	Returns the first element of the process instance notification vector.
NextElement()	Returns the next element of the process instance notification vector.
Size()	Returns the number of elements in the process instance notification vector.

FmcjProcessInstanceVector

A process instance vector represents the result of a query for process instances.

Refer to “Handling collections” on page 30 for detailed descriptions of vector access functions.

Accessor methods	Description
Deallocate()	Deallocates the storage for a process instance vector object.
FirstElement()	Returns the first element of the process instance vector.

Accessor methods	Description
NextElement()	Returns the next element of the process instance vector.
Size()	Returns the number of elements in the process instance vector.

FmcjProcessTemplate

A process template object represents the Runtime equivalent of a Buildtime workflow process model.

Refer to “Basic functions/subprograms” on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Copy()	Allocates and initializes the storage for a process template object by copying.
Deallocate()	Deallocates the storage for a process template object.
Equal()	Compares two process templates.
IsComplete()	Indicates whether the complete process template information is available.

Refer to “Accessor function/subprograms” on page 62 for detailed descriptions of accessor functions/subprograms.

Note: The value in the **Set** column shows if this attribute is a primary attribute (P) and set immediately when process templates are queried or if this attribute is a secondary attribute (S) and set only after the refresh of a specific process template.

Accessor methods	Set	Description
AuditMode()	S	Returns the audit mode of the process template.
Category()	P	Returns the category of the process template.
CategoryIsNull()	P	Indicates whether a category is set.
CreationTime()	P	Returns the creation time of the process template.
Description()	P	Returns the description of the process template.
DescriptionIsNull()	P	Indicates whether a description is set.
Documentation()	S	Returns the documentation of the process template.
DocumentationIsNull()	S	Indicates whether a documentation is set.
Icon()	P	Returns the icon associated with the process template.
InContainerName()	S	Returns the name of the î of the process template.
InContainerNeeded()	P	Indicates whether an input container is needed to start an instance of the process template.
LastModificationTime()	P	Returns the last time a primary attribute of the process template was changed.
Name()	P	Returns the name of the process template.
OrganizationName()	S	Returns the name of the organization of the process template.

Accessor methods	Set	Description
OrganizationNameIsNull()	S	Indicates whether an organization name is set.
OutContainerName()	S	Returns the name of the ô of the process template.
PersistentOid()	P	Returns a representation of the object identification of the process template.
ProcessAdmin()	S	Returns the name of the process administrator of an instance of the process template.
ProcessAdminIsNull()	S	Indicates whether a process administrator is set.
RoleName()	S	Returns the name of the role of the process template.
RoleNameIsNull()	S	Indicates whether a role is set.
ValidFromTime()	P	Returns the time when the process template becomes valid.

Action methods	Description	Page
CreateAndStartInstance()	Creates and starts an instance of the process template.	379
CreateInstance()	Creates an instance of the process template.	385
CreateAndSuspendInstance()	Creates and suspends an instance of the process template.	382
Delete()	Deletes the process template.	388
InContainer()	Retrieves the input container of the process template.	390
PersistentObject()	Retrieves the process template specified by the passed object identification.	391
Refresh()	Retrieves the complete information of the process template.	393

FmcjProcessTemplateList

A process template list represents a group of process templates. All functions/subprograms of a persistent list are also applicable to process template lists.

Refer to “Basic functions/subprograms” on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Copy()	Allocates and initializes the storage for a process template list object by copying.
Deallocate()	Deallocates the storage for a process template list object.
Equal()	Compares two process template lists.

Action methods	Description	Page
QueryProcessTemplates()	Retrieves the process templates qualifying via the process template list.	395

FmcjProcessTemplateListVector

A process template list vector represents the result of a query for process template lists.

Refer to "Handling collections" on page 30 for detailed descriptions of vector access functions.

Accessor methods	Description
Deallocate()	Deallocates a process template list vector object.
FirstElement()	Returns the first element of the process template list vector.
NextElement()	Returns the next element of the process template list vector.
Size()	Returns the number of elements in the process template list vector.

FmcjProcessTemplateVector

A process template vector represents the result of a query for process templates.

Refer to "Handling collections" on page 30 for detailed descriptions of vector access functions.

Accessor methods	Description
Deallocate()	Deallocates the storage for a process template vector object.
FirstElement()	Returns the first element of the process template vector.
NextElement()	Returns the next element of the process template vector.
Size()	Returns the number of elements in the process template vector.

FmcjProgramData

A program data object represents the program implementation definitions.

Refer to "Basic functions/subprograms" on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Copy()	Allocates and initializes the storage for a program data object by copying.
Deallocate()	Deallocates the storage for a program data object.
Equal()	Compares two program data objects.

Refer to "Accessor function/subprograms" on page 62 for detailed descriptions of accessor functions/subprograms.

Accessor methods	Description
Description()	Returns the description of the implementing program.
DescriptionIsNull()	Indicates whether a description is set.
Icon()	Returns the icon associated with the implementing program.

Accessor methods	Description
Implementations()	Returns the implementation definitions of the program.
InContainer()	Returns the î of the program.
IsUnattended()	States whether the program can run unattended.
OutContainer()	Returns the ô of the program.

FmcjReadOnlyContainer

A read-only container represents an input data container of a work item. All functions/subprograms of a container are applicable to read-only containers.

Refer to “Basic functions/subprograms” on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Copy()	Allocates and initializes the storage for a read-only container object by copying.
Deallocate()	Deallocates the storage for a read-only container object.
Equal()	Compares two read-only containers.

FmcjReadWriteContainer

A read/write container represents an î of a process instance or an ô of a work item. All functions/subprograms of a container are applicable to read/write containers.

Refer to “Basic functions/subprograms” on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Copy()	Allocates and initializes the storage for a read/write container object by copying.
Deallocate()	Deallocates the storage for a read/write container object.
Equal()	Compares two read/write containers.

Refer to “Accessor function/subprograms” on page 62 for detailed descriptions of accessor functions/subprograms.

Accessor methods	Description
SetArrayBinaryValue()	Sets the value of the specified container leaf element in the C language. The leaf element is part of an array and of type BINARY.
SetArrayFloatValue()	Sets the value of the specified container leaf element in the C language. The leaf element is part of an array and of type FLOAT.
SetArrayLongValue()	Sets the value of the specified container leaf element in the C language. The leaf element is part of an array and of type LONG.

Accessor methods	Description
SetArrayStringValue()	Sets the value of the specified container leaf element in the C language. The leaf element is part of an array and of type STRING.
SetBinaryValue()	Sets the value of the specified container leaf element in the C language. The leaf element is of type BINARY.
SetFloatValue()	Sets the value of the specified container leaf element in the C language. The leaf element is of type FLOAT.
SetLongValue()	Sets the value of the specified container leaf element in the C language. The leaf element is of type LONG.
SetStringValue()	Sets the value of the specified container leaf element in the C language. The leaf element is of type STRING.

FmcjResult

A result object represents the result of a function/subprogram call.

Refer to “Accessor function/subprograms” on page 62 for detailed descriptions of accessor functions/subprograms.

Accessor methods	Description
MessageText()	Returns the result as an NLS regarding formatted message.
ObjectOfCurrentThread()	Returns the result object associated with the thread from where this function/subprogram is called.
Origin()	Returns the origin of the result, that is, file, line, function.
Parameters()	Returns the parameters of the result; these are to be incorporated into the message text.
Rc()	Returns the return code remembered in the result object.

FmcjService

A service object represents common aspects of MQSeries Workflow service objects. All functions/subprograms of a service are also applicable to execution services.

Refer to “Accessor function/subprograms” on page 62 for detailed descriptions of accessor functions/subprograms.

Accessor methods	Description
IsLoggedOn()	Indicates whether a successful logon request has been issued.
SetTimeOut()	Sets the time the client will wait for a server to answer (see “Setting a value of type long” on page 68).
SystemGroupName()	Returns the name of the system group where the server resides.
SystemName()	Returns the name of the system where the server resides.
Timeout()	Returns the time the client will wait for a server to answer.
UserID()	Returns the user identification of the logged-on user.

Action methods	Description	Page
SetPassword()	Sets the password of the logged-on user.	405
UserSettings()	Retrieves the user settings of the logged-on user.	407

FmcjStringVector

A string vector serves to represents a set of string information. For example, a string vector is returned to show the categories the logged-on user is authorized for. Or, a string vector must be used to specify the persons to stand in for.

Refer to "Handling collections" on page 30 for detailed descriptions of vector access functions.

Accessor methods	Description
AddElement()	Adds a string to the string vector.
Allocate()	Allocates the storage for a string vector.
Deallocate()	Deallocates the storage for a string vector.
FirstElement()	Returns the first element of the string vector.
FirstResultParmElement()	Returns the first element of a string vector representing the parameters of a result object; calling this function does not change the result object and thus allows for a consistent read.
NextElement()	Returns the next element of the string vector.
NextResultParmElement()	Returns the next element of a string vector representing the parameters of a result object; calling this function does not change the result object and thus allows for a consistent read.
RemoveElement()	Removes a string from the string vector.
ResultParmDeallocate()	Deallocates the storage for a string vector representing the parameters of a result object; calling this function does not change the result object and thus allows for a consistent read.
ResultParmSize()	Returns the number of elements in a string vector representing the parameters of a result object; calling this function does not change the result object and thus allows for a consistent read.
Size()	Returns the number of elements in the string vector.

FmcjSymbolLayout

A symbol layout object represents graphical information of a named icon.

Refer to "Basic functions/subprograms" on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Copy()	Allocates and initializes the storage for a symbol layout object by copying.
Deallocate()	Deallocates the storage for a symbol layout object.
Equal()	Compares two symbol layout objects on the basis of their contents.

Refer to “Accessor function/subprograms” on page 62 for detailed descriptions of accessor functions/subprograms. All properties are primary properties.

Accessor methods	Description
XPosition()	Returns the x-coordinate of the named icon.
XPositionOfName()	Returns the x-coordinate of the name associated to the icon.
YPosition()	Returns the y-coordinate of the named icon.
YPositionOfName()	Returns the y-coordinate of the name associated to the icon.

FmcjWorkitem

A work item represents an activity instance assigned to a user in order to be worked on. All functions/subprograms of FmcjItem are also applicable to work items.

Refer to “Basic functions/subprograms” on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Copy()	Allocates and initializes the storage for a work item object by copying.
Deallocate()	Deallocates the storage for a work item object.
Kind()	States that the object is a work item.
Equal()	Compares two work items.

Refer to “Accessor function/subprograms” on page 62 for detailed descriptions of accessor functions/subprograms.

Note: The value in the **Set** column shows if this attribute is a primary attribute (P) and set immediately when work items are queried or if this attribute is a secondary attribute (S) and set only after the refresh of a specific work item.

Accessor methods	Set	Description
ActivityKind()	P	Returns the kind of the associated activity instance, whether it is a program or process and so on.
ErrorReason()	S	Returns an error object describing the reason why the associated activity instance is in state InError.
ErrorReasonIsNull()	S	Indicated whether an error reason is set.
ExitCondition()	S	Returns the exit condition of the work item.
FirstNotificationTime()	S	Returns the first notification time of the work item.
FirstNotificationTimeIsNull()	S	Indicates whether a first notification time is set.
Implementation()	P	Returns the name of the implementing program of the associated activity instance.
ImplementationIsNull()	P	Indicates whether an implementation is set.
ManualExitMode()	S	Returns whether the exit mode of the work item is manual.
ManualStartMode()	S	Returns whether the start mode of the work item is manual.

Accessor methods	Set	Description
Priority()	P	Returns the priority of the work item.
SecondNotificationTime()	S	Returns the second notification time of the work item.
SecondNotificationTimeIsNull()	S	Indicates whether a second notification time is set.
Staff()	S	Returns all persons a work item for the associated activity instance has been assigned to.
StartCondition()	S	Returns the start condition of the work item.
State	P	Returns the state of the work item.
StateOfNotification()	S	Returns the notification state of the work item.
SupportTools()	P	Returns the support tools associated with the work item.
SupportToolsIsNull()	P	Indicates whether support tools are set.

Action methods	Description	Page
CheckIn()	Checks in the work item.	411
CheckOut()	Checks out the work item.	413
Finish()	Finishes a manual exit work item.	415
ForceFinish()	Force finishes the work item.	417
ForceRestart()	Force restarts the work item.	419
InContainer()	Retrieves the input container of the work item.	421
OutContainer()	Retrieves the output container of the work item.	422
PersistentObject()	Retrieves the specified work item.	424
Restart()	Restarts the work item.	426
Start()	Starts the work item.	427
Terminate()	Terminates the work item.	429

FmcjWorkitemVector

A work item vector represents the result of a query for work items.

Refer to “Handling collections” on page 30 for detailed descriptions of vector access functions.

Accessor methods	Description
Deallocate()	Deallocates the storage for a work item vector object.
FirstElement()	Returns the first element of the work item vector.
NextElement()	Returns the next element of the work item vector.
Size()	Returns the number of elements in the work item vector.

FmcjWorklist

A worklist represents a group of items. All functions/subprograms of a persistent list are also applicable to worklists.

Refer to "Basic functions/subprograms" on page 53 for detailed descriptions of basic functions/subprograms.

Basic methods	Description
Copy()	Allocates and initializes the storage for a worklist object by copying.
Deallocate()	Deallocates the storage for a worklist object.
Equal()	Compares two worklists.

Refer to "Accessor function/subprograms" on page 62 for detailed descriptions of accessor functions/subprograms.

Accessor methods	Description
BeepOption()	Indicates whether a beep should sound when the contents of the worklist changes.

Action methods	Description	Page
QueryActivityInstanceNotifications()	Retrieves the activity instance notifications qualifying via the worklist.	431
QueryItems()	Retrieves all items qualifying via the worklist.	433
QueryProcessInstanceNotifications()	Retrieves the process instance notifications qualifying via the worklist.	436
QueryWorkitems()	Retrieves the work items qualifying via the worklist.	438

FmcjWorklistVector

A worklist vector represents the result of a query for worklists.

Refer to "Handling collections" on page 30 for detailed descriptions of vector access functions.

Accessor methods	Description
Deallocate()	Deallocates a worklist vector object.
FirstElement()	Returns the first element of the worklist vector.
NextElement()	Returns the next element of the worklist vector.
Size()	Returns the number of elements in the worklist vector.

Part 7. Application programming interfaces

This chapter describes the MQSeries Workflow application programming interfaces in alphabetical order.

Each entry contains a functional description of the API function/subprogram followed by subsections:

Usage notes
Points to general information about the nature of this call.

Authorization
States the authority required to have the API call executed.

Required connection
States the MQSeries Workflow server a session must have been established with.

API include file
States the name of the file to be included for the API function/subprogram declaration.

C language signature
Shows the C language syntax of the API call.

Cobol language signature
Shows the Cobol language syntax of the API call.

Parameters
Describes each of the parameters together with an indicator whether the parameter is an input or output parameter.

Return type
Describes the value returned by the call.

Return codes
Lists all possible return codes which may be raised by this call.

Examples
Points to an example of the call.

Chapter 31. FmcjActivityInstance functions/subprograms

An FmcjActivityInstance object represents an instance of an activity of a process template. An activity instance is uniquely identified by its object identifier or by its fully qualified name within the process instance. The fully qualified name of an activity instance is a name in dot notation where the hierarchy of nested activities of type *Block* is presented from left to right, and their names are separated by a dot.

FmcjActivityInstanceObtainProcessInstanceMonitor()

This function/subprogram retrieves the process instance monitor for the process instance the activity instance is part of from the MQSeries Workflow execution server (action call).

When the deep option is specified, all activity instances of type Block are resolved, that is, their block instance monitors are also fetched from the server.

Note: Deep is currently not supported.

The application is completely responsible for the ownership of objects, that is, it is not checked whether the process instance monitor handle already points to some object when a new one is assigned.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.
- ’’

Authorization

One of:

- Process authorization
- Process administration authorization
- Be the process administrator
- Be the process creator
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjActivityInstanceObtainProcessInstanceMonitor(
       FmcjActivityInstanceHandle            hdlInstance,
       bool                                  deep,
       FmcjProcessInstanceMonitorHandle *     monitor)
```

Cobol language signature

Cobol language signature

```
            PERFORM FmcjAINObtainProcInstMon.

         FmcjAINObtainProcInstMon.

            CALL    "FmcjItemObtainProcessInstanceMonitor"
                           USING
                           BY VALUE
                              hdlItem
                              deep
                           BY REFERENCE
                              monitor
                           RETURNING
                              intReturnValue.
```

Parameters

hdlInstance
Input. The activity instance whose process instance monitor is to be retrieved.

deep Input. An indicator whether activity instances of type Block are to be resolved, that is, their monitor is also to be provided. Note, deep is currently ignored.

monitor
Input/Output. The address of the handle to the process instance monitor respectively the process instance monitor object to be set.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_EMPTY(122)
The object has not yet been read from the database, that is, does not yet represent a persistent one.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is incorrect; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The activity instance does no longer exist.

FMC_ERROR_NOT_AUTHORIZED(119)
Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; the server to which the connection should be established is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjActivityInstanceSubProcessInstance()

This function/subprogram retrieves the process instance which is implementing the activity instance from the MQSeries Workflow execution server (action call).

All information about the process instance, primary and secondary, is retrieved.

The application is completely responsible for the ownership of objects, that is, it is not checked whether the process instance monitor handle already points to some object when a new one is assigned.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.
- ’’

Authorization

One of:

- Process authorization
- Process administration authorization
- Be the process creator
- Be the process administrator
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjActivityInstanceSubProcessInstance(
       FmcjActivityInstanceHandle   hdlInstance,
       FmcjProcessInstanceHandle *  instance )
```

Cobol language signature

Cobol language signature

```
            PERFORM FmcjAISubProcInst.

         FmcjAISubProcInst.

            CALL    "FmcjActivityInstanceSubProcessInstance"
                           USING
                           BY VALUE
                              hdlInstance
                           BY REFERENCE
                              instance
                           RETURNING
                              intReturnValue.
```

Parameters

hdlInstance
Input. The handle of the activity instance object to be queried.

instance
Input/Output. The subprocess instance object to be retrieved (initialized).

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_EMPTY(122)
The object has not yet been read from the database, that is, does not yet represent a persistent one.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is incorrect; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The activity instance does no longer exist.

FMC_ERROR_NOT_AUTHORIZED(119)
Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; the server to which the connection should be established is not defined in your profile.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
: Timeout has occurred.

Chapter 32. FmcjActivityInstanceNotification functions/subprograms

An FmcjActivityInstanceNotification object represents a notification on an activity instance assigned to a user.

Other items assigned to users are process instance notifications and work items. FmcjItem represents the common properties of such items. In the C++ implementation, FmcjActivityInstanceNotification is thus a subclass of the FmcjItem class. Similarly, in the C and Cobol language, it takes common implementations of functions from FmcjItem.

An activity instance notification is uniquely identified by its object identifier.

FmcjActivityInstanceNotificationPersistentObject()

This function/subprogram retrieves the activity instance notification identified by the passed object identifier from the MQSeries Workflow execution server (action call).

The MQSeries Workflow execution server from which the activity instance notification is to be retrieved is identified by the service object. The activity instance notification handle to be initialized must be a null pointer, respectively the activity instance notification object to be initialized must be empty. The transient object is then updated with all information (primary and secondary) of the activity instance notification.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjActivityInstanceNotificationPersistentObject(
       FmcjExecutionServiceHandle                service,
       char const *                              oid,
       FmcjActivityInstanceNotificationHandle *  hdlitem )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjAINPersistentObj.

        FmcjAINPersistentObj.

           CALL    "FmcjActivityInstanceNotificationPersistentObject"
                          USING
                          BY VALUE
                             serviceValue
                             oid
                          BY REFERENCE
                             hdlItem
                          RETURNING
                             intReturnValue.
```

Parameters

service
: Input. The service object representing the session with the execution server.

oid
: Input. The object identifier of the activity instance notification to be retrieved.

hdlItem
: Input/Output. The address of the handle to the activity instance notification object to be set.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The activity instance notification does no longer exist.

FMC_ERROR_INVALID_OID(805)
: The provided oid is invalid.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Chapter 33. FmcjBlockInstanceMonitor functions/subprograms

An FmcjBlockInstanceMonitor object represents a monitor for an activity instance of type *Block*.

Note: The ownership of a block instance monitor stays with the embracing process instance monitor. A block instance monitor is automatically deleted when the process instance monitor is deleted. After that action, using the block instance monitor handle or object is invalid.

An FmcjBlockInstanceMonitor object represents the common aspects of monitors. In the C++ implementation, it is the superclass of the FmcjProcessInstanceMonitor class. Similarly, in the C language, it provides for common implementations of functions.

FmcjBlockInstanceMonitorObtainBlockInstance Monitor()

This function/subprogram retrieves the block instance monitor for the specified activity instance from the MQSeries Workflow execution server (action call).

The activity instance must be of type *Block*.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

One of:

- Process authorization
- Process administration authorization
- Be the process administrator
- Be the process creator
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol)

C language signature

C language signature

```
FmcjBlockInstanceMonitorHandle
  FMC_APIENTRY FmcjBlockInstanceMonitorObtainBlockInstanceMonitor(
        FmcjBlockInstanceMonitorHandle  hdlMonitor,
        FmcjActivityInstanceHandle      activity )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjBIMObtainBlockInstMon.

        FmcjBIMObtainBlockInstMon.

           CALL
                  "FmcjBlockInstanceMonitorObtainBlockInstanceMonitor"
                          USING
                          BY VALUE
                             hdlMonitor
                             activity
                          RETURNING
                             FmcjBIMHandleReturnValue.
```

Parameters

activity
: Input. The activity instance of type Block whose block instance monitor is to be retrieved.

Return types

FmcjBlockInstanceMonitor*/Handle
: A pointer to the block instance monitor object respectively the handle to the block instance monitor.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_EMPTY(122)
: The object has not yet been read from the database, that is, does not yet represent a persistent one.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is incorrect; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The specified activity instance is not described by the block instance monitor.

FMC_ERROR_WRONG_KIND(501)
: The specified activity instance is not of type Block.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; the server to which the connection should be established is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjBlockInstanceMonitorObtainProcessInstance Monitor()

This function/subprogram retrieves the process instance monitor for the specified activity instance from the MQSeries Workflow execution server (action call).

The activity instance must be of type *Process*.

When the deep option is specified, then activity instances of type Block are resolved, that is, their block instance monitors are also fetched from the server.

Note: Deep is currently not supported.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

One of:

- Process authorization
- Process administration authorization
- Be the process administrator
- Be the process creator
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol)

C language signature

C language signature

```
FmcjProcessInstanceMonitorHandle
  FMC_APIENTRY FmcjBlockInstanceMonitorObtainProcessInstanceMonitor(
       FmcjBlockInstanceMonitorHandle  hdlMonitor,
       FmcjActivityInstanceHandle      activity,
       bool                            deep      )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjBIMObtainProcInstMon.

        FmcjBIMObtainProcInstMon.

           CALL
                "FmcjBlockInstanceMonitorObtainProcessInstanceMonitor"
                          USING
                          BY VALUE
                             hdlMonitor
                             activity
                             deep
                          RETURNING
                             FmcjPIMHandleReturnValue.
```

Parameters

activity
: Input. The activity instance of type Process whose process instance monitor is to be retrieved.

deep
: Input. An indicator whether activity instances of type Block are to be resolved, that is, their monitor is also to be provided. Note, deep is currently ignored.

Return types

FmcjProcessInstanceMonitor*/Handle
: A pointer to the process instance monitor object respectively the handle to the process instance monitor.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_EMPTY(122)
: The object has not yet been read from the database, that is, does not yet represent a persistent one.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is incorrect; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The specified activity instance is not described by the block instance monitor.

FMC_ERROR_WRONG_KIND(501)
The specified activity instance is not of kind Process.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; the server to which the connection should be established is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjBlockInstanceMonitorRefresh()

This function/subprogram refreshes the block instance monitor from the MQSeries Workflow execution server (action call).

All information about the block instance monitor is retrieved.

When the deep option is specified, then activity instances of type Block are resolved, that is, their block instance monitors are also refreshed from the server.

Note: Deep is currently not supported.

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

One of:
- Process authorization
- Process administration authorization
- Be the process administrator
- Be the process creator
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjBlockInstanceMonitorRefresh(
       FmcjBlockInstanceMonitorHandle hdlMonitor,
       bool                           deep      )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjBIMRefresh.

        FmcjBIMRefresh.

           CALL    "FmcjBlockInstanceMonitorRefresh"
                          USING
                          BY VALUE
                             hdlMonitor
                             deep
                          RETURNING
                             intReturnValue.
```

Parameters

hdlMonitor
: Input. The handle of the block instance monitor to be refreshed.

deep
: Input. An indicator whether activity instances of type Block are to be resolved, that is, their monitor is also to be provided. Note, deep is currently ignored.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_EMPTY(122)
: The object has not yet been read from the database, that is, does not yet represent a persistent one.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is incorrect; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The process instance does no longer exist.

FMC_ERROR_NOT_AUTHORIZED(119)
: Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; the server to which the connection should be established is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Chapter 34. FmcjContainer functions/subprograms

An FmcjContainer object represents a data container of a process template, process instance, work item, activity implementation, or support tool. A container can be a read-only input container or a read/write input or output container.

An FmcjContainer object represents the common aspects of read-only and read/write containers. In the C++ implementation, it is the superclass of the FmcjReadOnlyContainer and FmcjReadWriteContainer classes. Similarly, in the C and Cobol language, it provides for common implementations of functions.

The functions/subprograms defined on the FmcjContainer class allow to access the values of data members of a basic type (container leaves), or to get a substructure of a container, an FmcjContainerElement object.

FmcjContainerInContainer()

This function/subprogram retrieves the input container from the CICS COMMAREA/IMS I/O Area.

It can be used from within an activity implementation or support tool.

Note that this function/subprogram call will fail after the COMMAREA/ I/O Area has been changed with FmcjContainerSetOutContainer or FmcjContainerSetRemoteOutContainer.

Properties

Usage notes

- See "Activity implementation functions/subprograms" on page 76 for general information.

Authorization

Be an activity implementation or support tool

Required connection

None but active MQSeries Workflow program execution server

API include file

Runtime: fmcjccon.h (C language) or fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET  FMC_APIENTRY FmcjContainerInContainer(
        FmcjReadOnlyContainerHandle *  input )
```

Cobol language signature

Cobol language signature

```
          PERFORM FmcjCInCtnr.

       FmcjCInCtnr.

          CALL    "FmcjContainerInContainer"
                         USING
                         BY REFERENCE
                            inputValue
                         RETURNING
                            intReturnValue.
```

Parameters

input Input/Output. The address of the input container handle respectively the input container of the activity implementation or support tool to be set.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_NO_CTNR_ACCESS(1021)
The program does not have an input container.

FMC_ERROR_NOT_AUTHORIZED(119)
Not authorized to use the function/subprogram.

FMC_ERROR_PROGRAM_EXECUTION(126)
The function/subprogram was not called from within an activity implementation or support tool or the program execution server is not active.

FMC_ERROR_COMMUNICATION(13)
The specified program execution server cannot be reached.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Examples

- For a C language example see “Programming an executable (C language)” on page 465
- For a Cobol example see “Programming an executable (Cobol language)” on page 466

FmcjContainerOutContainer()

This function/subprogram retrieves the output container from the CICS COMMAREA/IMS I/O Area.

It can be used from within an activity implementation.

Note that this function/subprogram call will fail after the COMMAREA/ I/O Area has been changed with FmcjContainerSetOutContainer or FmcjContainerSetRemoteOutContainer.

Properties

Usage notes

- See "Activity implementation functions/subprograms" on page 76 for general information.

Authorization

Be an activity implementation

Required connection

None but active MQSeries Workflow program execution server

API include file

Runtime: fmcjccon.h (C language) or fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET  FMC_APIENTRY FmcjContainerOutContainer(
        FmcjReadWriteContainerHandle *   output )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjCOutCtnr.

        FmcjCOutCtnr.

           CALL    "FmcjContainerOutContainer"
                          USING
                          BY REFERENCE
                             outputValue
                          RETURNING
                             intReturnValue.
```

Parameters

output
: Input/Output. The address of the output container handle respectively the output container of the activity implementation to be set.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_NO_CTNR_ACCESS(1021)
: The program does not have an output container.

FMC_ERROR_NOT_AUTHORIZED(119)
: Not authorized to use the function/subprogram.

FMC_ERROR_PROGRAM_EXECUTION(126)
: The function/subprogram was not called from within an activity implementation or the program execution server is not active.

FMC_ERROR_COMMUNICATION(13)
: The specified program execution server cannot be reached.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
: Timeout has occurred.

Examples

- For a C language example see “Programming an executable (C language)” on page 465
- For a Cobol example see “Programming an executable (Cobol language)” on page 466

FmcjContainerRemoteInContainer()

This function/subprogram retrieves the input container from the CICS COMMAREA/IMS I/O Area.

It can be used from within a program started by an activity implementation or support tool, if the COMMAREA/IMS I/O Area got passed to the program.

Note that this function/subprogram call will fail after the COMMAREA/ I/O Area has been changed with FmcjContainerSetOutContainer or FmcjContainerSetRemoteOutContainer.

Properties

Usage notes

- See “Activity implementation functions/subprograms” on page 76 for general information.

Authorization

Be a program started by an activity implementation or support tool

Required connection

None but active MQSeries Workflow program execution server

API include file

Runtime: fmcjccon.h (C language) or fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol)

C language signature

C language signature

```
APIRET  FMC_APIENTRY FmcjContainerRemoteInContainer(
        char const *                  programID,
        FmcjReadOnlyContainerHandle * input )
```

Cobol language signature

Cobol language signature

```
            PERFORM FmcjCRemoteInCtnr.

         FmcjCRemoteInCtnr.

            CALL    "FmcjContainerRemoteInContainer"
                           USING
                           BY REFERENCE
                              programID
                              inputValue
                           RETURNING
                              intReturnValue.
```

Parameters

programID
Input. The program identification by which the activity implementation or support tool is known to the program execution server.

input
Input/Output. The address of the input container handle respectively the input container of the activity implementation or support tool to be set.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_NO_CTNR_ACCESS(1021)
The program does not have an input container.

FMC_ERROR_NOT_AUTHORIZED(119)
Not authorized to use the function/subprogram.

FMC_ERROR_PROGRAM_EXECUTION(126)
The function/subprogram was not called from within an activity implementation or the program execution server is not active.

FMC_ERROR_COMMUNICATION(13)
The specified program execution server cannot be reached.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FmcjContainerRemoteOutContainer()

This function/subprogram retrieves the output container from the CICS COMMAREA/IMS I/O Area.

It can be used from within a program started by an activity implementation or support tool, if the COMMAREA/IMS I/O Area got passed to the program.

Note that this function/subprogram call will fail after the COMMAREA/ I/O Area has been changed with FmcjContainerSetOutContainer or FmcjContainerSetRemoteOutContainer.

Properties

Usage notes

- See “Activity implementation functions/subprograms” on page 76 for general information.

Authorization

Be a program started by an activity implementation

Required connection

None but active MQSeries Workflow program execution server

API include file

Runtime: fmcjccon.h (C language) or fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol)

C language signature

C language signature

```
APIRET  FMC_APIENTRY FmcjContainerRemoteOutContainer(
        char const *                    programID,
        FmcjReadWriteContainerHandle *  output )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjCRemoteOutCtnr.

        FmcjCRemoteOutCtnr.

           CALL    "FmcjContainerRemoteOutContainer"
                          USING
                          BY REFERENCE
                             programID
                             outputValue
                          RETURNING
                             intReturnValue.
```

Parameters

programID
: Input. The program identification by which the activity implementation is known to the program execution server.

output
: Input/Output. The address of the output container handle respectively the output container of the activity implementation to be set.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_NO_CTNR_ACCESS(1021)
: The program does not have an output container.

FMC_ERROR_NOT_AUTHORIZED(119)
: Not authorized to use the function/subprogram.

FMC_ERROR_PROGRAM_EXECUTION(126)
: The function/subprogram was not called from within an activity implementation or the program execution server is not active.

FMC_ERROR_COMMUNICATION(13)
: The specified program execution server cannot be reached.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FmcjContainerSetRemoteOutContainer()

This function/subprogram returns the output container to the MQSeries Workflow program execution server (activity implementation call).

It can be used from within a program started by an activity implementation as often as required. Note, however, that the output container is not returned to the MQSeries Workflow execution server until the activity implementation ends. It is kept transiently by the CICS COMMAREA/IMS I/O Area.

Note that the functions/subprograms FmcjContainerInContainer, FmcjContainerOutContainer, FmcjContainerServicePassthrough and ″Remote″ counterparts will fail after this function was called due to an altered COMMAREA/ I/O Area.

Properties

Usage notes

- See "Activity implementation functions/subprograms" on page 76 for general information.

Authorization

Be a program started by an activity implementation

Required connection

None but active MQSeries Workflow program execution server

API include file

Runtime: fmcjccon.h (C language) or fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol)

C language signature

C language signature

```
APIRET  FMC_APIENTRY FmcjContainerSetRemoteOutContainer(
        char const *                        programID,
        FmcjReadWriteContainerHandle const  output )
```

Cobol language signature

Cobol language signature

```
          PERFORM FmcjCSetRemoteOutCtnr.

       FmcjCSetRemoteOutCtnr.

          CALL    "FmcjContainerSetRemoteOutContainer"
                         USING
                         BY REFERENCE
                            programID
                         BY VALUE
                            outputValue
                         RETURNING
                            intReturnValue.
```

Parameters

programID
: Input. The program identification by which the activity implementation is known to the program execution server.

output
: Input. The output container handle respectively the output container of the activity implementation to be passed.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_NO_CTNR_ACCESS(1021)
: The program does not have an output container.

FMC_ERROR_NOT_AUTHORIZED(119)
: Not authorized to use the function/subprogram.

FMC_ERROR_PROGRAM_EXECUTION(126)
: The function/subprogram was not called from within an activity implementation or the program execution server is not active.

FMC_ERROR_COMMUNICATION(13)
: The specified program execution server cannot be reached.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FmcjContainerSetOutContainer()

This function/subprogram returns the output container to the MQSeries Workflow program execution server (activity implementation call).

It can be used from within an activity implementation as often as required. Note, however, that the output container is not returned to the MQSeries Workflow execution server until the activity implementation ends. It is kept transiently in the CICS COMMAREA/IMS I/O Area.

Note that the functions/subprograms FmcjContainerInContainer, FmcjContainerOutContainer, FmcjContainerServicePassthrough and ″Remote″ counterparts will fail after this function was called due to an altered COMMAREA/ I/O Area.

Properties

Usage notes

- See “Activity implementation functions/subprograms” on page 76 for general information.

Authorization

Be an activity implementation

Required connection

None but active MQSeries Workflow program execution server

API include file

Runtime: fmcjccon.h (C language) or fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET  FMC_APIENTRY FmcjContainerSetOutContainer(
        FmcjReadWriteContainerHandle const   output )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjCSetOutCtnr.

        FmcjCSetOutCtnr.

           CALL    "FmcjContainerSetOutContainer"
                          USING
                          BY VALUE
                             outputValue
                          RETURNING
                             intReturnValue.
```

Parameters

output
Input. The output container handle respectively the output container of the activity implementation to be passed.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_NO_CTNR_ACCESS(1021)
The program does not have an output container.

FMC_ERROR_NOT_AUTHORIZED(119)
Not authorized to use the function/subprogram.

FMC_ERROR_PROGRAM_EXECUTION(126)
The function/subprogram was not called from within an activity implementation or the program execution server is not active.

FMC_ERROR_COMMUNICATION(13)
The specified program execution server cannot be reached.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Examples

- For a C language example see “Programming an executable (C language)” on page 465
- For a Cobol example see “Programming an executable (Cobol language)” on page 466

Chapter 35. FmcjExecutionService functions/subprograms

An FmcjExecutionService object represents a session between a user and an MQSeries Workflow execution server so that Runtime services may be asked for.

The execution service object essentially provides for the basic functions/subprograms to set up a communication path to the specified MQSeries Workflow execution server and to establish the user session (log on), and finish it (log off).

At FmcjExecutionService construction or allocation time the name of the MQSeries Workflow system and system group where the execution server resides can be specified. Default values are taken from the current user's profile or workstation profile, in this sequence, when logging on.

When the session to an execution server has been established, you can query objects for which you are authorized; that means, you can query process templates, process instances, or work items. The attributes of the queried objects can then be read and further actions can be requested. For example, once a process template has been queried, creation of a process instance can be asked for.

When the execution service object is destructed or deallocated and still represents an active session, logoff is automatically called (provided that there is no other object referencing this session). It is, however, recommended that logon and logoff calls are paired before the execution service object is deallocated.

FmcjExecutionServiceCreateActivityInstanceList()

This function/subprogram creates a new user-associated activity instance list on the MQSeries Workflow execution server so that activities can be grouped to one's own taste or for a group of users. (action call).

A activity instance list is represented by its name which is unique per type - public or private (user). It groups a set of activities which have the same characteristics. These characteristics are primarily defined via search filters. The number of activities in the list can be restricted via a threshold which specifies the maximum number of activities to be returned to the client. That threshold is applied after the activity instance list has been sorted according to sort criteria specified. Note that activities are sorted on the server, that is, the code page of the server determines the sort sequence.

The following rules apply for specifying a activity instance list name:
- You can specify a maximum of 32 characters.
- You can use any printable characters depending on your current locale, except the following:

  `* ? " ; : .`
- You can use blanks with these restrictions: no leading blanks, no trailing blanks, and no consecutive blanks.

If no name or an empty name is provided for the activity instance list, MQSeries Workflow generates a name *UserID_n*, where *UserID* is the user identification of the person logged on and n is some number.

Any name passed remains unchanged; FmcjActivityInstanceListName() respectively FmcjActivityInstanceListName() returns the name of the activity instance list.

You can filter on the following properties of the activities to be included in the activity instance list:
- Category
- Description
- LastModified
- LastStateChange
- Name
- ProcessInstanceName
- State
- Type

You can sort on the following properties of the activities to be included in the activity instance list:
- Category
- Description
- LastModified
- LastStateChange
- Name
- ProcessInstanceName
- State
- Type

Properties

Asynchronous function/subprogram considerations

None

Usage notes
- See “Action functions/subprograms” on page 76 for general information.

Authorization

Valid user session

Required connection

MQSeries Workflow execution server

API include file

fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjExecutionServiceCreateActivityInstanceList(
       FmcjExecutionServiceHandle       service,
       char const *                     name,
       char const *                     description,
       char const *                     filter,
       char const *                     sortCriteria,
       unsigned long *                  threshold,
       FmcjActivityInstanceListHandle * newList )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjESCreateActInstList.

        FmcjESCreateActInstList.

           CALL    "FmcjExecutionServiceCreateActivityInstanceList"
                          USING
                          BY VALUE
                             serviceValue
                             name
                             typeValue
                             description
                             filter
                             sortCriteria
                             threshold
                          BY REFERENCE
                             newList
                          RETURNING
                             intReturnValue.
```

Parameters

service
Input. A handle to the service object representing the session with the execution server.

name
Input. A user-defined name for the activity instance list.

description
Input. A user-defined description of the activity instance list.

filter
Input. The filter criteria which characterize the activities in the activity instance list.

sortCriteria
Input. The sort criteria to be applied to the activities in the activity instance list.

threshold
Input. The threshold which defines the maximum number of activities in the activity instance list to be passed to the client.

newList
Input/Output. The newly created activity instance list.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_INVALID_FILTER(125)
The specified filter is not applicable to activities.

FMC_ERROR_INVALID_NAME(134)
The specified activity instance list name does not comply with the syntax rules.

FMC_ERROR_INVALID_SORT(808)
The specified sort criteria are not applicable to activities.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_NOT_UNIQUE(121)
The name of the activity instance list is not unique.

Examples

- For a C language example see “Create a process instance list (C language)” on page 443 .
- For a Cobol example see “Create a process instance list (Cobol language)” on page 444.

FmcjExecutionServiceCreateProcessInstanceList()

This function/subprogram creates a process instance list on the MQSeries Workflow execution server so that process instances can be grouped to one’s own taste or for a group of users (action call).

A process instance list is identified by:
- Its name, which is unique per type
- Its type, that is, an indicator whether the list is for public or private usage
- Its owner, that is, the owner of the list when the type is private

When the process instance list is to be created for public usage or for the private usage of another user, that is, not the logged-on user itself, then the logged-on user needs to have staff definition authorization.

A process instance list groups a set of process instances which have the same characteristics. These characteristics are defined via search filters. The number of process instances in the list can be restricted via a threshold which specifies the maximum number of process instances to be returned to the client. That threshold is applied after the process instance list has been sorted according to sort criteria specified. Note that process instances are sorted on the server, that is, the code page of the server determines the sort sequence.

The following rules apply for specifying a process instance list name:

- You can specify a maximum of 32 characters.
- You can use any printable characters depending on your current locale, except the following:

  `* ? " ; : .`
- You can use blanks with these restrictions: no leading blanks, no trailing blanks, and no consecutive blanks.

The following rules apply for specifying a description:

- You can specify a maximum of 254 characters.
- You can use any printable characters depending on your current locale, including the end-of-line and new-line characters.

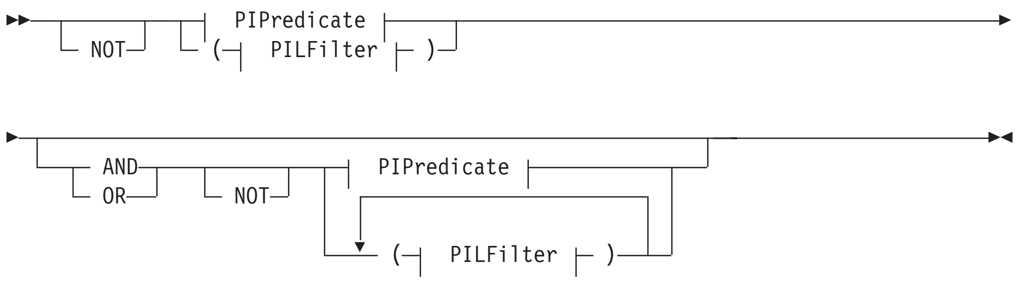
A process instance list filter is specified as a character string containing a filter on process instances (refer to “How to read the syntax diagrams” on page x):

Note: A *string* constant is to be enclosed in single quotes (′).

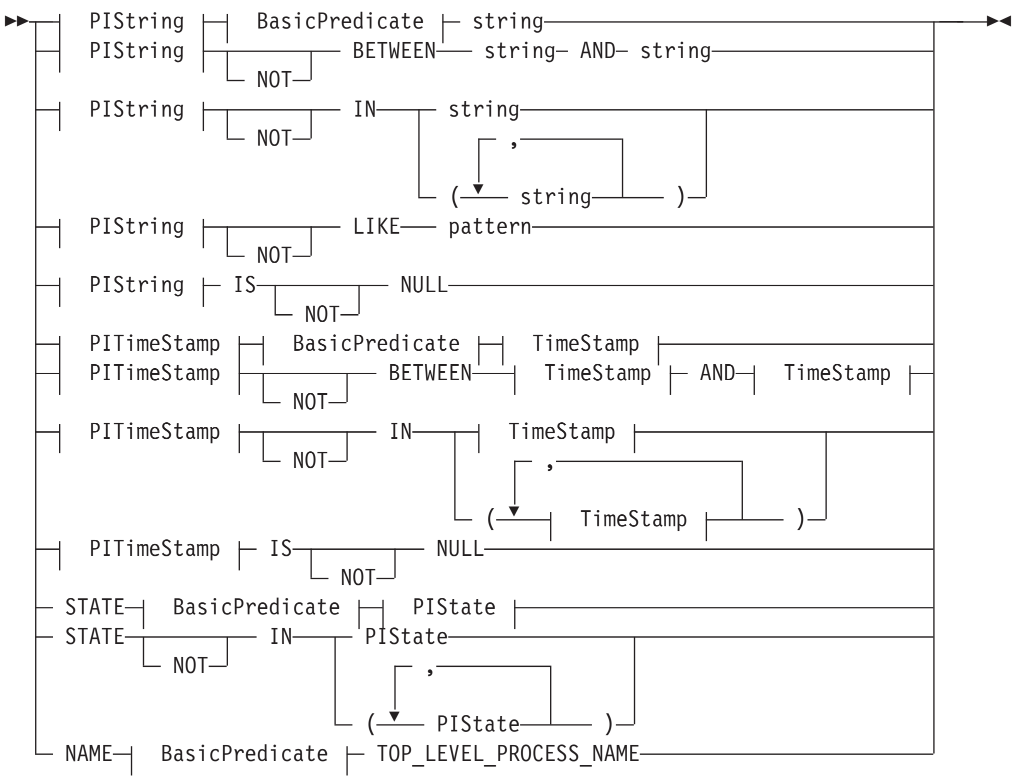
A *pattern* is a string constant in which the asterisk and the question mark have special meanings.

- The question mark (?) represents any single character.
- The asterisk (*) represents a string of zero or more characters.
- The escape character is backslash (\) and must be used when the pattern itself contains actual question marks or asterisks.

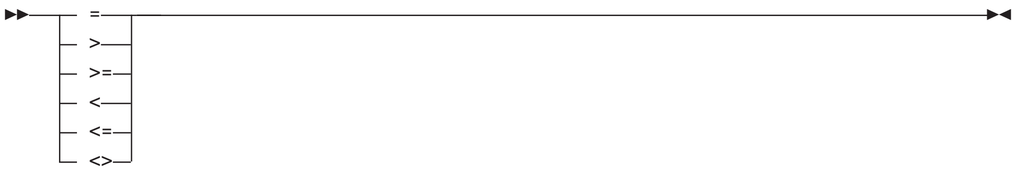
PILFilter



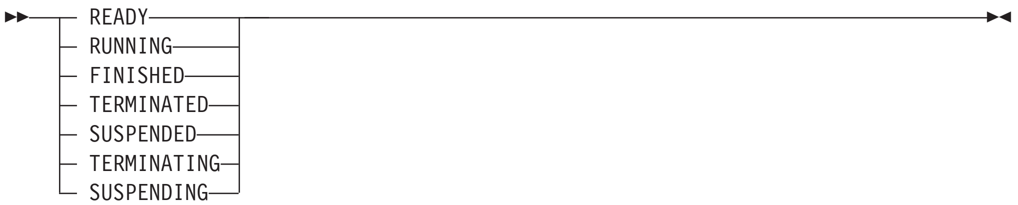
PIPredicate



BasicPredicate



PIState



PIString

```
►►──┬─ADMINISTRATOR──────────┬──────────────────────────────────────────►◄
    ├─CATEGORY───────────────┤
    ├─DESCRIPTION────────────┤
    ├─NAME───────────────────┤
    ├─PARENT_PROCESS_NAME────┤
    └─TOP_LEVEL_PROCESS_NAME─┘
```

PITimeStamp

```
►►──┬─LAST_MODIFICATION_TIME─┬──────────────────────────────────────────►◄
    └─LAST_STATE_CHANGE_TIME─┘
```

TimeStamp

```
►►──year─ -– month─ -– day──┬────────────────────────────────────────┬──►◄
                            └─ – hours─┬─────────────────────────┬───┘
                                       └─ :– minutes─┬─────────────┬─┘
                                                     └─ :– seconds─┘
```

A process instance list sort criterion is specified as a character string:

Note: The default sort order is ascending.

States are sorted according to the sequence shown in the PIState diagram.

PILOrderBy

```
                                   ┌─ , ─────────────┐
                                   ▼                 │
►►──┬─┤ PIString ├────┬────────────────┬─────────────┴──────────────────►◄
    ├─┤ PITimeStamp ├─┤                ├─ASC──┤
    └─STATE───────────┘                └─DESC─┘
```

Properties

Asynchronous function/subprogram considerations

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

None or staff definition or be the system administrator

Required connection

MQSeries Workflow execution server

API include file

fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjExecutionServiceCreateProcessInstanceList(
       FmcjExecutionServiceHandle        service,
       char const *                      name,
       enum FmcjPersistentListTypeOfList type,
       char const *                      owner,
       char const *                      description,
       char const *                      filter,
       char const *                      sortCriteria,
       unsigned long *                   threshold,
       FmcjProcessInstanceListHandle *   newList )
```

Cobol language signature

Cobol language signature

```
            PERFORM FmcjESCreateProcInstList.

         FmcjESCreateProcInstList.

            CALL    "FmcjExecutionServiceCreateProcessInstanceList"
                           USING
                           BY VALUE
                              serviceValue
                              name
                              typeValue
                              ownerValue
                              description
                              filter
                              sortCriteria
                              threshold
                           BY REFERENCE
                              newList
                           RETURNING
                              intReturnValue.
```

Parameters

service
Input. A handle to the service object representing the session with the execution server.

name Input. A user-defined name for the process instance list.

type Input. An indication whether a private or a public list is to be created.

owner Input. The owner of the list when the type is private. Ignored for public lists.

description
Input. A user-defined description of the process instance list.

filter Input. The filter criteria which characterize the process instances to be contained in the process instance list.

sortCriteria
Input. The sort criteria to be applied to the process instances in the process instance list.

threshold
Input. The threshold which defines the maximum number of process instances in the process instance list to be passed to the client.

newList
Input/Output. The newly created process instance list.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_INVALID_DESCRIPTION(810)
The specified description is invalid.

FMC_ERROR_INVALID_FILTER(125)
The specified filter is invalid.

FMC_ERROR_INVALID_LIST_TYPE(813)
The specified list type is invalid.

FMC_ERROR_INVALID_NAME(134)
The specified process instance list name does not comply with the syntax rules.

FMC_ERROR_INVALID_USER(132)
The user ID specified for the owner of the list does not conform to the syntax rules.

FMC_ERROR_INVALID_SORT(808)
The specified sort criteria are invalid.

FMC_ERROR_INVALID_THRESHOLD(807)
The specified threshold is invalid; exceeds the maximum possible value.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_NOT_AUTHORIZED(119)
Not authorized.

FMC_ERROR_OWNER_NOT_FOUND(812)
The person to become the owner of the process instance list is not found.

FMC_ERROR_NOT_UNIQUE(121)
The name of the process instance list is not unique within the specified type.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Examples

- For a C language example see “Create a process instance list (C language)” on page 443.
- For a Cobol example see “Create a process instance list (Cobol language)” on page 444.

FmcjExecutionServiceCreateProcessTemplateList()

This function/subprogram creates a process template list on the MQSeries Workflow execution server so that process templates can be grouped to one’s own taste or for a group of users (action call).

A process template list is identified by:
- Its name, which is unique per type
- Its type, that is, an indicator whether the list is for public or private usage
- Its owner, that is, the owner of the list when the type is private

When the process template list is to be created for public usage or for the private usage of another user, that is, not the logged-on user itself, then the logged-on user needs to have staff definition authorization.

A process template list groups a set of process templates which have the same characteristics. These characteristics are defined via filters. The number of process templates in the list can be restricted via a threshold which specifies the maximum number of process templates to be returned to the client. That threshold is applied after the process template list has been sorted according to sort criteria specified. Process templates are sorted on the server, that is, the code page of the server determines the sort sequence.

The following rules apply for specifying a process template list name:
- You can specify a maximum of 32 characters.
- You can use any printable characters depending on your current locale, except the following:
  `* ? " ; : .`
- You can use blanks with these restrictions: no leading blanks, no trailing blanks, and no consecutive blanks.

The following rules apply for specifying a description:
- You can specify a maximum of 254 characters.

- You can use any printable characters depending on your current locale, including the end-of-line and new-line characters.

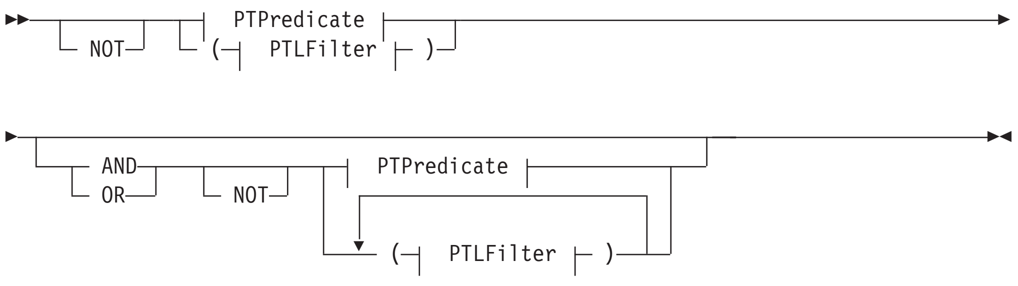
A process template list filter is specified as a character string containing a filter on process templates (refer to “How to read the syntax diagrams” on page x):

Note: A *string* constant is to be enclosed in single quotes (′).

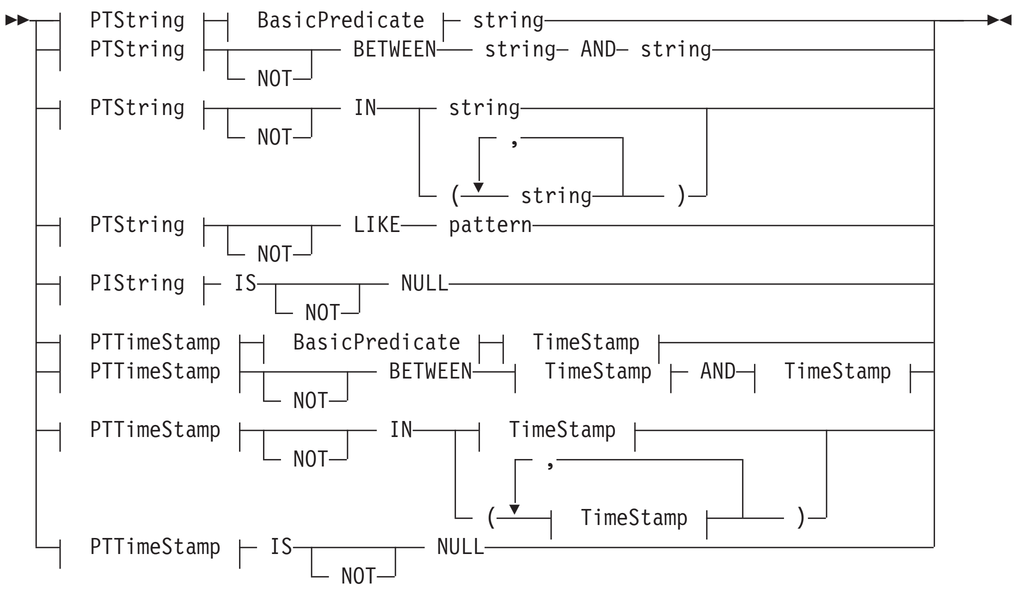
A *pattern* is a string constant in which the asterisk and the question mark have special meanings.

- The question mark (?) represents any single character.
- The asterisk (*) represents a string of zero or more characters.
- The escape character is backslash (\) and must be used when the pattern itself contains actual question marks or asterisks.

PTLFilter



PTPredicate



BasicPredicate

```
►►──┬─ =──┬──────────────────────────────────────────────────────────►◄
    ├─ >──┤
    ├─ >=─┤
    ├─ <──┤
    ├─ <=─┤
    └─ <>─┘
```

PTString

```
►►──┬─ CATEGORY────┬─────────────────────────────────────────────────►◄
    ├─ DESCRIPTION─┤
    └─ NAME────────┘
```

PTTimeStamp

```
►►──LAST_MODIFICATION_TIME───────────────────────────────────────────►◄
```

TimeStamp

```
►►──year─ -- month─ -- day──┬────────────────────────────────────────┬──►◄
                            └── - hours─┬─────────────────────────┬─┘
                                        └─ :- minutes─┬─────────────┬┘
                                                      └─ :- seconds─┘
```

A process template list sort criterion is specified as a character string:

Note: The default sort order is ascending.

PTLOrderBy

```
                                    ┌─ ,──────────┐
                                    ▼             │
►►──┬─┤ PTString ├───┬──────────┬─────────┬───────────►◄
    └─┤ PTTimeStamp ├┘          ├─ ASC──┤
                                └─ DESC─┘
```

Properties

Asynchronous function/subprogram considerations

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

None or staff definition or be the system administrator

Required connection

MQSeries Workflow execution server

API include file

fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjExecutionServiceCreateProcessTemplateList(
       FmcjExecutionServiceHandle       service,
       char const *                     name,
       enum FmcjPersistentListTypeOfList type,
       char const *                     owner,
       char const *                     description,
       char const *                     filter,
       char const *                     sortCriteria,
       unsigned long *                  threshold,
       FmcjProcessTemplateListHandle *   newList )
```

Cobol language signature

Cobol language signature

```
            PERFORM FmcjESCreateProcTemplList.

         FmcjESCreateProcTemplList.

            CALL    "FmcjExecutionServiceCreateProcessTemplateList"
                           USING
                           BY VALUE
                              serviceValue
                              name
                              typeValue
                              ownerValue
                              description
                              filter
                              sortCriteria
                              threshold
                           BY REFERENCE
                              newList
                           RETURNING
                              intReturnValue.
```

Parameters

service
Input. A handle to the service object representing the session with the execution server.

name Input. A user-defined name for the process template list.

type Input. An indication whether a private or a public list is to be created.

owner Input. The owner of the list when the type is private. Ignored for public lists.

description
Input. A user-defined description of the process template list.

filter
Input. The filter criteria which characterize the process templates in the process template list.

sortCriteria
Input. The sort criteria to be applied to the process templates in the process template list.

threshold
Input. The threshold which defines the maximum number of process templates in the process template list.

newList
Input/Output. The newly created process template list.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_INVALID_DESCRIPTION(810)
The specified description is invalid.

FMC_ERROR_INVALID_FILTER(125)
The specified filter is invalid.

FMC_ERROR_INVALID_LIST_TYPE(813)
The specified list type is invalid.

FMC_ERROR_INVALID_NAME(134)
The specified process template list name does not comply with the syntax rules.

FMC_ERROR_INVALID_USER(132)
The user ID specified for the owner of the list does not conform to the syntax rules.

FMC_ERROR_INVALID_SORT(808)
The specified sort criteria are invalid.

FMC_ERROR_INVALID_THRESHOLD(807)
The specified threshold is invalid; exceeds the maximum possible value.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_NOT_AUTHORIZED(119)
Not authorized.

FMC_ERROR_OWNER_NOT_FOUND(812)
The person to become the owner of the process template list is not found.

FMC_ERROR_NOT_UNIQUE(121)
The name of the process template list is not unique within the specified type.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Examples

- For a C language example see “Create a process instance list (C language)” on page 443.
- For a Cobol example see “Create a process instance list (Cobol language)” on page 444.

FmcjExecutionServiceCreateWorklist()

This function/subprogram creates a worklist on the MQSeries Workflow execution server so that work items or notifications can be grouped to one’s own taste or for a group of users (action call).

A worklist is identified by:
- Its name, which is unique per type
- Its type, that is, an indicator whether the list is for public or private usage
- Its owner, that is, the owner of the list when the type is private

When the worklist is to be created for public usage or for the private usage of another user, that is, not the logged-on user itself, then the logged-on user needs to have staff definition authorization.

A worklist groups a set of work items or notifications which have the same characteristics. These characteristics are defined via filters. The number of items in the worklist can be restricted via a threshold which specifies the maximum number of items to be returned to the client. That threshold is applied after the worklist has been sorted according to sort criteria specified. Items are sorted on the server, that is, the code page of the server determines the sort sequence.

The following rules apply for specifying a worklist name:
- You can specify a maximum of 32 characters.
- You can use any printable characters depending on your current locale, except the following:
  `* ? " ; : .`
- You can use blanks with these restrictions: no leading blanks, no trailing blanks, and no consecutive blanks.

The following rules apply for specifying a description:
- You can specify a maximum of 254 characters.

- You can use any printable characters depending on your current locale, including the end-of-line and new-line characters.

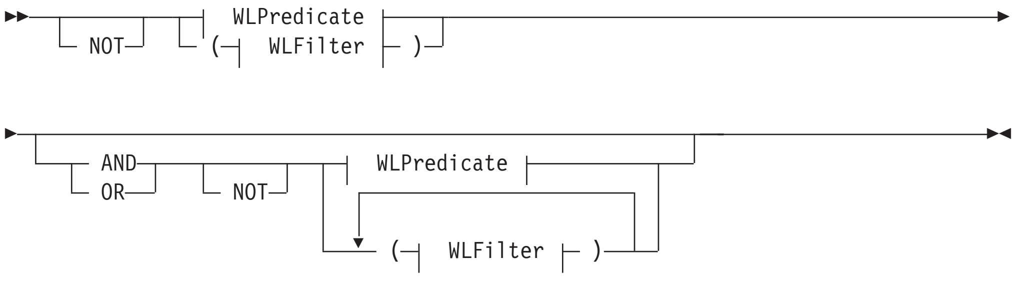
A worklist filter is specified as a character string containing a filter on the items in the worklist (refer to “How to read the syntax diagrams” on page x):

Note: A *string* constant is to be enclosed in single quotes (′).

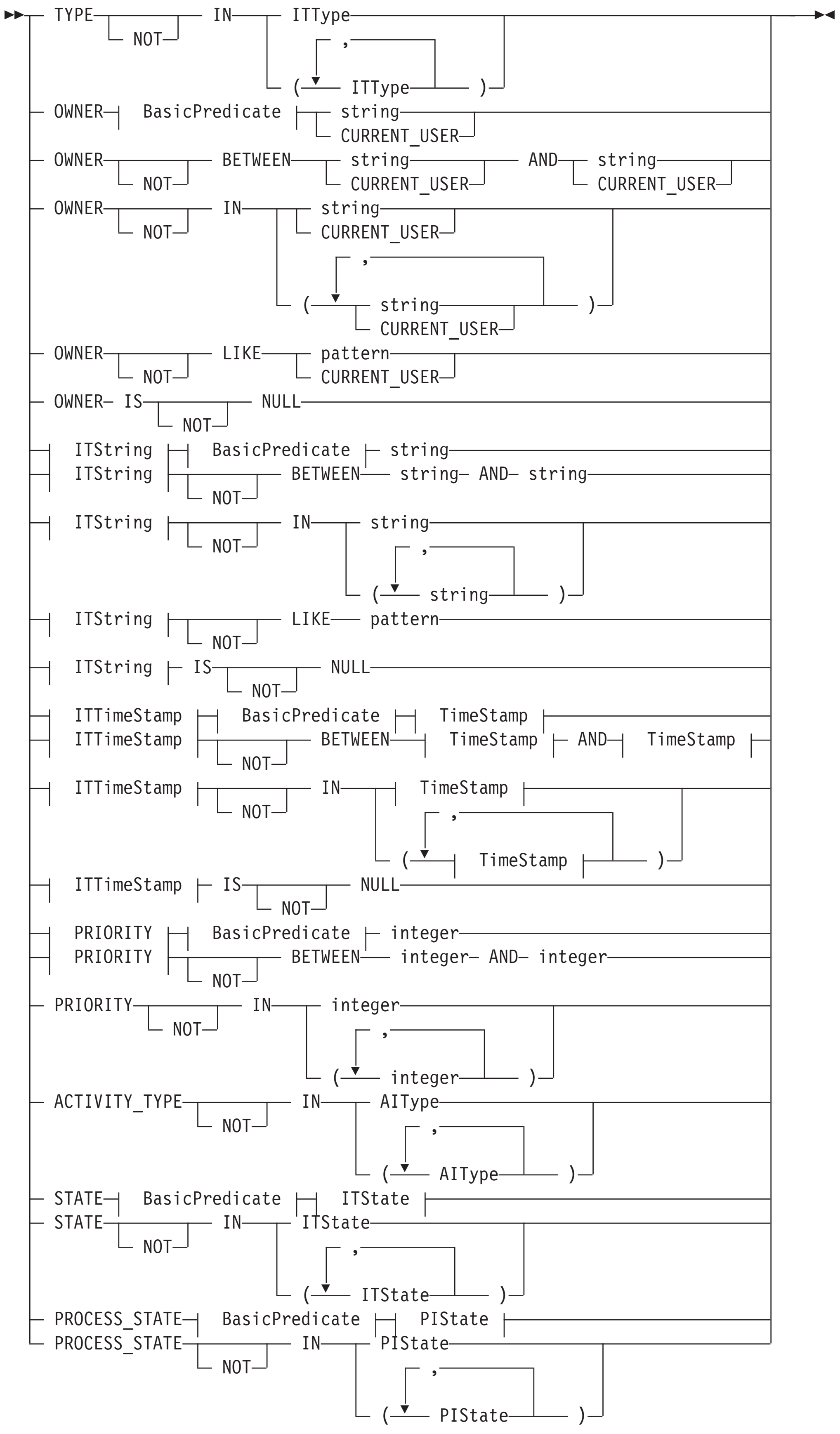
A *pattern* is a string constant in which the asterisk and the question mark have special meanings.

- The question mark (?) represents any single character.
- The asterisk (*) represents a string of zero or more characters.
- The escape character is backslash (\) and must be used when the pattern itself contains actual question marks or asterisks.

WLFilter



WLPredicate


TYPE NOT IN ITType (, ITType)
OWNER BasicPredicate string CURRENT_USER
OWNER NOT BETWEEN string CURRENT_USER AND string CURRENT_USER
OWNER NOT IN string CURRENT_USER (, string CURRENT_USER)
OWNER NOT LIKE pattern CURRENT_USER
OWNER IS NOT NULL
ITString BasicPredicate string
ITString NOT BETWEEN string AND string
ITString NOT IN string (, string)
ITString NOT LIKE pattern
ITString IS NOT NULL
ITTimeStamp BasicPredicate TimeStamp
ITTimeStamp NOT BETWEEN TimeStamp AND TimeStamp
ITTimeStamp NOT IN TimeStamp (, TimeStamp)
ITTimeStamp IS NOT NULL
PRIORITY BasicPredicate integer
PRIORITY NOT BETWEEN integer AND integer
PRIORITY NOT IN integer (, integer)
ACTIVITY_TYPE NOT IN AIType (, AIType)
STATE BasicPredicate ITState
STATE NOT IN ITState (, ITState)
PROCESS_STATE BasicPredicate PIState
PROCESS_STATE NOT IN PIState (, PIState)

AIType

```
►►──┬─ PROCESS_ACTIVITY─┬──────────────────────────────────────────►◄
    └─ PROGRAM_ACTIVITY─┘
```

BasicPredicate

```
►►──┬─ =──┬──────────────────────────────────────────────────────────►◄
    ├─ >──┤
    ├─ >=─┤
    ├─ <──┤
    ├─ <=─┤
    └─ <>─┘
```

ITState

```
►►──┬─ READY──────────┬───────────────────────────────────────────────►◄
    ├─ RUNNING────────┤
    ├─ FINISHED───────┤
    ├─ TERMINATED─────┤
    ├─ SUSPENDED──────┤
    ├─ DISABLED───────┤
    ├─ CHECKED_OUT────┤
    ├─ IN_ERROR───────┤
    ├─ EXECUTED───────┤
    ├─ PLANNING───────┤
    ├─ FORCE_FINISHED─┤
    ├─ TERMINATING────┤
    └─ SUSPENDING─────┘
```

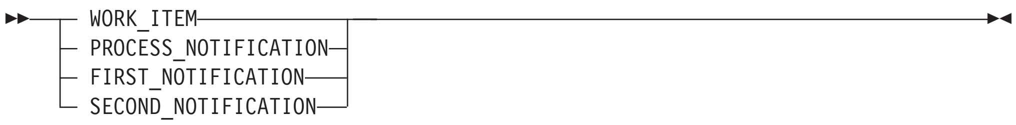
ITString

```
►►──┬─ DESCRIPTION──────┬─────────────────────────────────────────────►◄
    ├─ NAME─────────────┤
    ├─ PROCESS_CATEGORY─┤
    └─ PROCESS_NAME─────┘
```

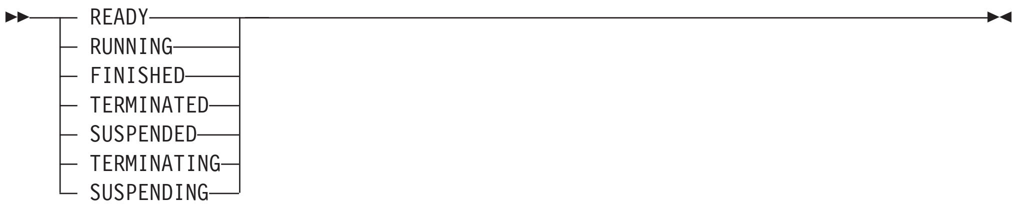
ITTimeStamp

```
►►──┬─ LAST_MODIFICATION_TIME─┬───────────────────────────────────────►◄
    └─ RECEIVED_TIME──────────┘
```

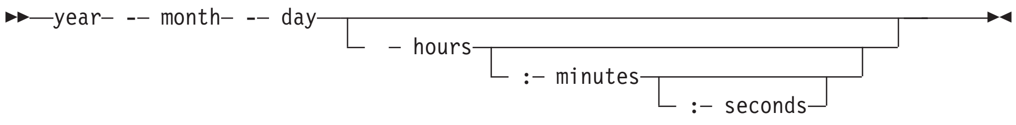
ITType



PIState



TimeStamp



A worklist sort criterion is specified as a character string.

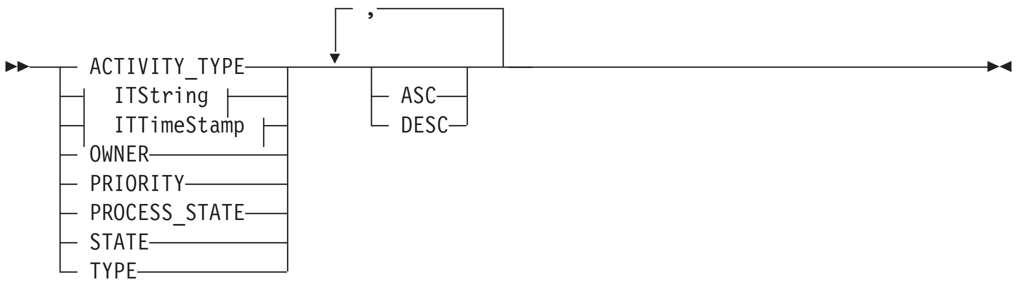
Note: The default sort order is ascending.

Activity types are sorted according to the sequence shown in the AIType diagram.

Item types are sorted according to the sequence shown in the ITType diagram.

States are sorted according to the sequence shown in the ITState respectively the PIState diagram.

WLOrderBy



Properties

Asynchronous function/subprogram considerations

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

None or staff definition or be the system administrator

Required connection

MQSeries Workflow execution server

API include file

fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjExecutionServiceCreateWorklist(
       FmcjExecutionServiceHandle       service,
       char const *                     name,
       enum FmcjPersistentListTypeOfList type,
       char const *                     owner,
       char const *                     description,
       char const *                     filter,
       char const *                     sortCriteria,
       unsigned long *                  threshold,
       FmcjWorklistHandle *             newList )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjESCreateWorklist.

        FmcjESCreateWorklist.

           CALL    "FmcjExecutionServiceCreateWorklist"
                          USING
                          BY VALUE
                             serviceValue
                             name
                             typeValue
                             ownerValue
                             description
                             filter
                             sortCriteria
                             threshold
                          BY REFERENCE
                             newList
                          RETURNING
                             intReturnValue.
```

Parameters

service
: Input. A handle to the service object representing the session with the execution server.

name Input. A user-defined name for the worklist.

type Input. An indication whether a private or a public list is to be created.

owner Input. The owner of the list when the type is private. Ignored for public lists.

description
Input. A user-defined description of the worklist.

filter Input. The filter criteria which characterize the items in the worklist.

sortCriteria
Input. The sort criteria to be applied to the items in the worklist.

threshold
Input. The threshold which defines the maximum number of items in the worklist.

newList
Input/Output. The newly created worklist.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_INVALID_DESCRIPTION(810)
The specified description is invalid.

FMC_ERROR_INVALID_FILTER(125)
The specified filter is invalid.

FMC_ERROR_INVALID_LIST_TYPE(813)
The specified list type is invalid.

FMC_ERROR_INVALID_NAME(134)
The specified worklist name does not comply with the syntax rules.

FMC_ERROR_INVALID_USER(132)
The user ID specified for the owner of the list does not conform to the syntax rules.

FMC_ERROR_INVALID_SORT(808)
The specified sort criteria are invalid.

FMC_ERROR_INVALID_THRESHOLD(807)
The specified threshold is invalid; exceeds the maximum possible value.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_NOT_AUTHORIZED(119)
Not authorized.

FMC_ERROR_OWNER_NOT_FOUND(812)
The person to become the owner of the worklist is not found.

FMC_ERROR_NOT_UNIQUE(121)
The name of the worklist is not unique within the specified type.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Examples

- For a C language example see "Create a process instance list (C language)" on page 443.
- For a Cobol example see "Create a process instance list (Cobol language)" on page 444.

FmcjExecutionServiceLogoff()

This function/subprogram allows the application to finish the specified user session with an MQSeries Workflow execution server (action call).

When logoff has been successfully executed, no further client/server calls are accepted using this execution service object.

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjExecutionServiceLogoff(
       FmcjExecutionServiceHandle  service )
```

Cobol language signature

Cobol language signature

```
            PERFORM FmcjESLogoff.

         FmcjESLogoff.

            CALL    "FmcjExecutionServiceLogoff"
                           USING
                           BY VALUE
                              serviceValue
                           RETURNING
                              intReturnValue.
```

Parameters

service
: Input. A handle to the service object representing the session with the execution server.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
: Timeout has occurred.

Examples

For examples see “Part 8. Examples” on page 441.

FmcjExecutionServiceLogon()

This function/subprogram allows an application to establish a user session with an MQSeries Workflow execution server (action call).

A successful Logon() is the prerequisite for using all other action and program execution management functions/subprograms of the MQSeries Workflow API.

The user ID to log on with must be a registered MQSeries Workflow user.

When logon has been successfully executed, the execution service object represents that single user session. A further request to log on with a different user ID will be rejected. You can, however, establish as many sessions as needed, even for the same user, using different execution service objects; one per session.

At logon time, several options can be specified which define your mode of operation. The session mode determines whether you are operating in a ″normal″ default session mode or whether the execution server should assume that you are present. When you are present, activity instances which are started automatically may be scheduled on your behalf.

There can only be a single present session per user. The *present here* option can be used, to force that other session logoff and to newly establish a present session here.

The following enumeration constants can be used to specify the session mode; it is strongly advised to use the symbolic names instead of the integer values:

C language	integer value
Fmc_SM_Default	0
Fmc_SM_Present	1
Fmc_SM_PresentHere	2

At logon time, you can also specify whether you are back in case you are set to be absent. When you are not absent you participate in work assignment; otherwise no work items are assigned to you.

The following enumeration constants can be used to deal with your absence; it is strongly advised to use the symbolic names instead of the integer values:

C language	integer value
Fmc_SA_NotSet	0
Fmc_SA_Reset	1
Fmc_SA_Leave	2

Leave means that your absence setting should stay as is. *Reset* means that your absence setting should be reset; you are back. *NotSet* means that you do not say anything about your absence, which means that your absence setting is reset or not according to the definition in your person record.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

Be a registered MQSeries Workflow user

Required connection

None

API include file

fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjExecutionServiceLogon (
       FmcjExecutionServiceHandle       service,
       char const *                     userID,
       char const *                     password,
       enum FmcjServiceSessionMode      sessionMode,
       enum FmcjServiceAbsenceIndicator absenceIndicator )
```

Cobol language signatures

Cobol language signature

```
           PERFORM FmcjESLogon.

        FmcjESLogon.

           CALL    "FmcjExecutionServiceLogon"
                          USING
                          BY VALUE
                             serviceValue
                             userID
                             passwordValue
                             sessionMode
                             absenceIndicator
                          RETURNING
                             intReturnValue.
```

Parameters

service
Input. A handle to the service object representing the session to be established with the execution server.

userID
Input. The user ID of the user on whose behalf a logon is to be made.

password
Input. The password of the user.

sessionMode
Input. The mode of the session to be established.

absenceIndicator
Input. An indicator to state how to handle any absence set.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_ALREADY_LOGGED_ON(11)
: The user is already logged on with present mode or the execution service object already represents a different user session.

FMC_ERROR_INVALID_ABSENCE_SPEC(905)
: An unknown absence setting has been specified.

FMC_ERROR_INVALID_SESSION_MODE(901)
: An unknown session mode has been specified.

FMC_ERROR_PASSWORD(12)
: Incorrect password.

FMC_ERROR_PROFILE(124)
: Required user or workstation profile entries cannot be found.

FMC_ERROR_USERID_UNKNOWN(10)
: No user ID registered with MQSeries Workflow has been provided.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
: Timeout has occurred.

Examples

For examples see “Part 8. Examples” on page 441.

FmcjExecutionServicePassthrough()

This function/subprogram can be used by an activity implementation or support tool to establish a user session with an MQSeries Workflow execution server from within this program (activity-implementation call).

When successfully executed, a session to the same execution server is set up from where the work item or support tool implemented by this program was started; the user on whose behalf the session is set up is the same one on whose behalf the work item was started.

Note that this function/subprogram call will fail after the COMMAREA/ I/O Area has been changed with FmcjContainerSetOutContainer or FmcjContainerSetRemoteOutContainer.

Properties

Usage notes

- See "Activity implementation functions/subprograms" on page 76 for general information.

Authorization

Activity implementation or support tool started by MQSeries Workflow

Required connection

None but active MQSeries Workflow program execution server

API include file

fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjExecutionServicePassthrough(
       FmcjExecutionServiceHandle   service )
```

Cobol language signature

Cobol language signature

```
            PERFORM FmcjESPassthrough.

         FmcjESPassthrough.

            CALL    "FmcjExecutionServicePassthrough"
                           USING
                           BY VALUE
                              serviceValue
                           RETURNING
                              intReturnValue.
```

Parameters

service
: Input. A handle to the service object which is to represent the session to be established with the execution server.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_PROGRAM_EXECUTION(126)
Passthrough was not called from within an activity implementation or support tool or the program execution server is not active.

FMC_ERROR_USERID_UNKNOWN(10)
The user who started the work item does no longer exist.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Examples

- For a C language example see "Programming an executable (C language)" on page 465 .
- For a Cobol example see "Programming an executable (Cobol language)" on page 466.

FmcjExecutionServiceQueryActivityInstance Notifications()

This function/subprogram retrieves the activity instance notifications the user has access to from the MQSeries Workflow execution server (action call).

Any activity instance notifications retrieved are appended to the supplied vector. If you want to read the current activity instance notifications only, you have to clear the vector before you call this function/subprogram. This means that you should set the vector handle to 0 via the FmcjXxxVectorDeallocate function.

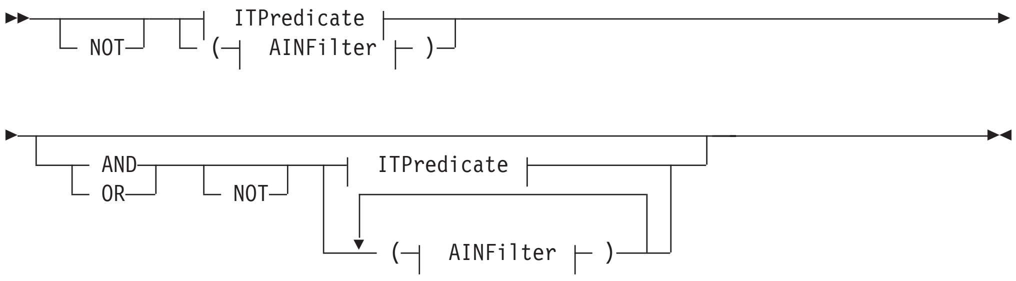
The activity instance notifications to be retrieved can be characterized by a filter. An activity instance notification filter is specified as a character string:

Note: A *string* constant is to be enclosed in single quotes (′).

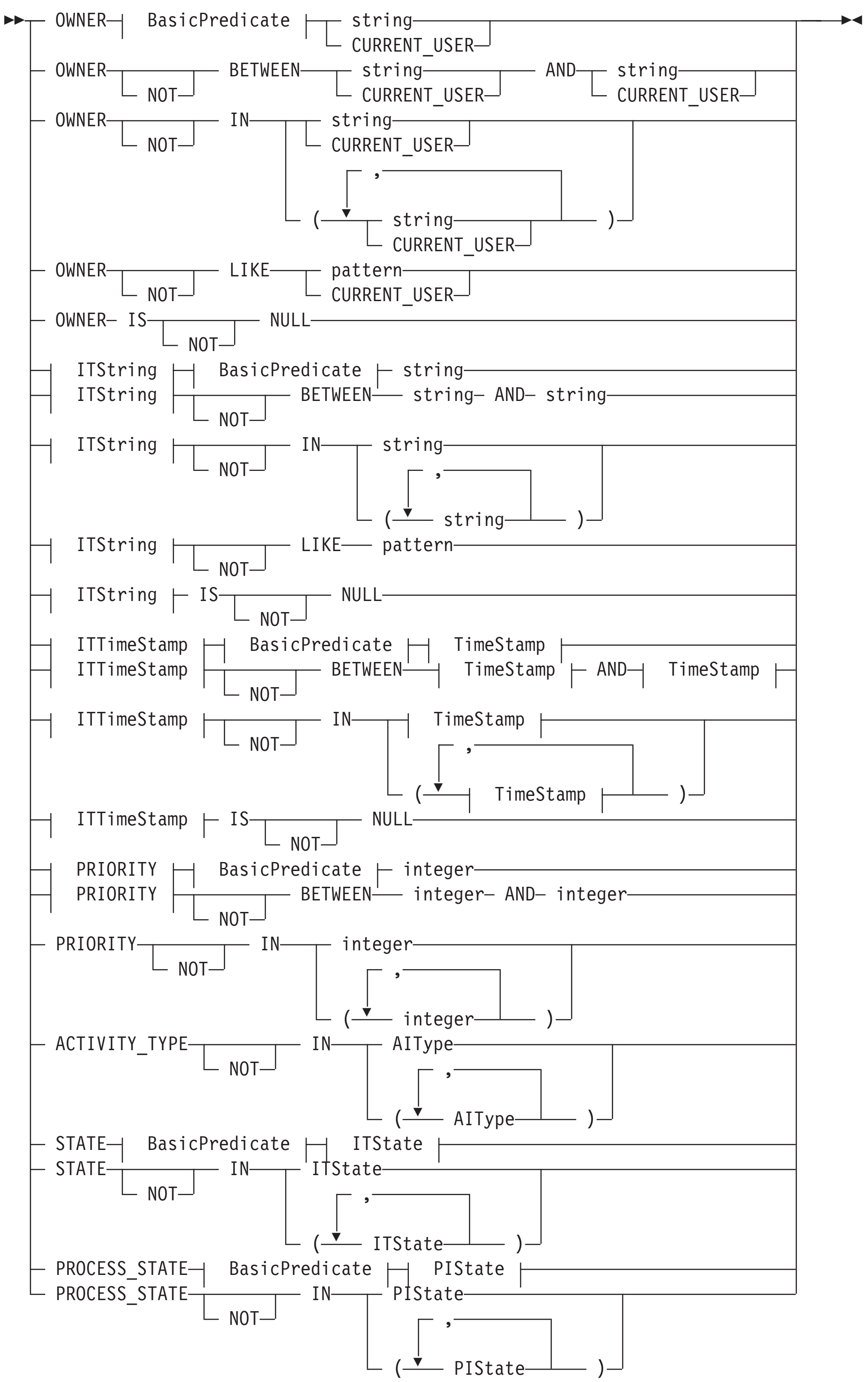
A *pattern* is a string constant in which the asterisk and the question mark have special meanings.

- The question mark (?) represents any single character.
- The asterisk (*) represents a string of zero or more characters.
- The escape character is backslash (\) and must be used when the pattern itself contains actual question marks or asterisks.

AINFilter


NOT
ITPredicate
(
AINFilter
)
AND
OR
NOT
ITPredicate
(
AINFilter
)

ITPredicate



OWNER BasicPredicate string CURRENT_USER
OWNER NOT BETWEEN string CURRENT_USER AND string CURRENT_USER
OWNER NOT IN string CURRENT_USER (, string CURRENT_USER)
OWNER NOT LIKE pattern CURRENT_USER
OWNER IS NOT NULL
ITString BasicPredicate string
ITString NOT BETWEEN string AND string
ITString NOT IN string (, string)
ITString NOT LIKE pattern
ITString IS NOT NULL
ITTimeStamp BasicPredicate TimeStamp
ITTimeStamp NOT BETWEEN TimeStamp AND TimeStamp
ITTimeStamp NOT IN TimeStamp (, TimeStamp)
ITTimeStamp IS NOT NULL
PRIORITY BasicPredicate integer
PRIORITY NOT BETWEEN integer AND integer
PRIORITY NOT IN integer (, integer)
ACTIVITY_TYPE NOT IN AIType (, AIType)
STATE BasicPredicate ITState
STATE NOT IN ITState (, ITState)
PROCESS_STATE BasicPredicate PIState
PROCESS_STATE NOT IN PIState (, PIState)

AIType

```
►►──┬─PROCESS_ACTIVITY─┬────────────────────────────────────────────────►◄
    └─PROGRAM_ACTIVITY─┘
```

BasicPredicate

```
►►──┬─ =──┬─────────────────────────────────────────────────────────────►◄
    ├─ >──┤
    ├─ >=─┤
    ├─ <──┤
    ├─ <=─┤
    └─ <>─┘
```

ITState

```
►►──┬─READY──────────┬──────────────────────────────────────────────────►◄
    ├─RUNNING────────┤
    ├─FINISHED───────┤
    ├─TERMINATED─────┤
    ├─SUSPENDED──────┤
    ├─DISABLED───────┤
    ├─CHECKED_OUT────┤
    ├─IN_ERROR───────┤
    ├─EXECUTED───────┤
    ├─PLANNING───────┤
    ├─FORCE_FINISHED─┤
    ├─TERMINATING────┤
    └─SUSPENDING─────┘
```

ITString

```
►►──┬─DESCRIPTION──────┬────────────────────────────────────────────────►◄
    ├─NAME─────────────┤
    ├─PROCESS_CATEGORY─┤
    └─PROCESS_NAME─────┘
```

ITTimeStamp

```
►►──┬─LAST_MODIFICATION_TIME─┬──────────────────────────────────────────►◄
    └─RECEIVED_TIME──────────┘
```

PIState

```
►►──┬─ READY───────┬──────────────────────────────────────────────►◄
    ├─ RUNNING─────┤
    ├─ FINISHED────┤
    ├─ TERMINATED──┤
    ├─ SUSPENDED───┤
    ├─ TERMINATING─┤
    └─ SUSPENDING──┘
```

TimeStamp

```
►►──year─ -─ month─ -─ day──┬───────────────────────────────────────┬──►◄
                            └─ ─ hours──┬────────────────────────┬──┘
                                        └─ :─ minutes──┬──────────────┬┘
                                                       └─ :─ seconds──┘
```

Activity instance notifications can be sorted. An activity instance notification sort criterion is specified as a character string:

Note: The default sort order is ascending.

Activity types are sorted according to the sequence shown in the AIType diagram.

States are sorted according to the sequence shown in the ITState respectively the PIState diagram.

AINOrderBy

```
                                ┌─ ,────────┐
                                ▼           │
►►──┬─ ACTIVITY_TYPE─────┬──────┬───────┬───┴────────────────────────►◄
    ├┤ ITString ├────────┤      ├─ ASC──┤
    ├┤ ITTimeStamp ├─────┤      └─ DESC─┘
    ├─ OWNER─────────────┤
    ├─ PRIORITY──────────┤
    ├─ PROCESS_STATE─────┤
    └─ STATE─────────────┘
```

The number of activity instance notifications to be retrieved can be restricted via a threshold which specifies the maximum number of activity instance notifications to be returned to the client. That threshold is applied after the activity instance notifications have been sorted according to the sort criteria specified. Note that the activity instance notifications are sorted on the server, that is, the code page of the server determines the sort sequence.

The primary information that is retrieved for each activity instance notification is:
- ActivityType
- Category
- CreationTime
- Description
- Icon

- Implementation
- Kind
- LastModificationTime
- Name
- Owner
- Priority
- ProcessInstanceName
- ReceivedAs
- ReceivedTime
- StartTime
- State
- SupportTools

FmcjExecutionServiceQueryItems()

This function/subprogram retrieves the work items or notifications the user has access to from the MQSeries Workflow execution server (action call).

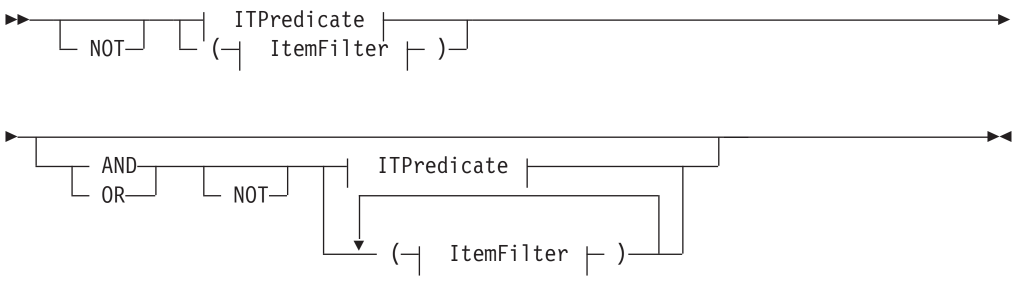
Any items retrieved are appended to the supplied vector. If you want to read the current items only, you have to clear the vector before you call this function/subprogram. This means that you should set the handle to 0 via the FmcjXxxVectorDeallocate function.

The items to be retrieved can be characterized by a filter. An item filter is specified as a character string:

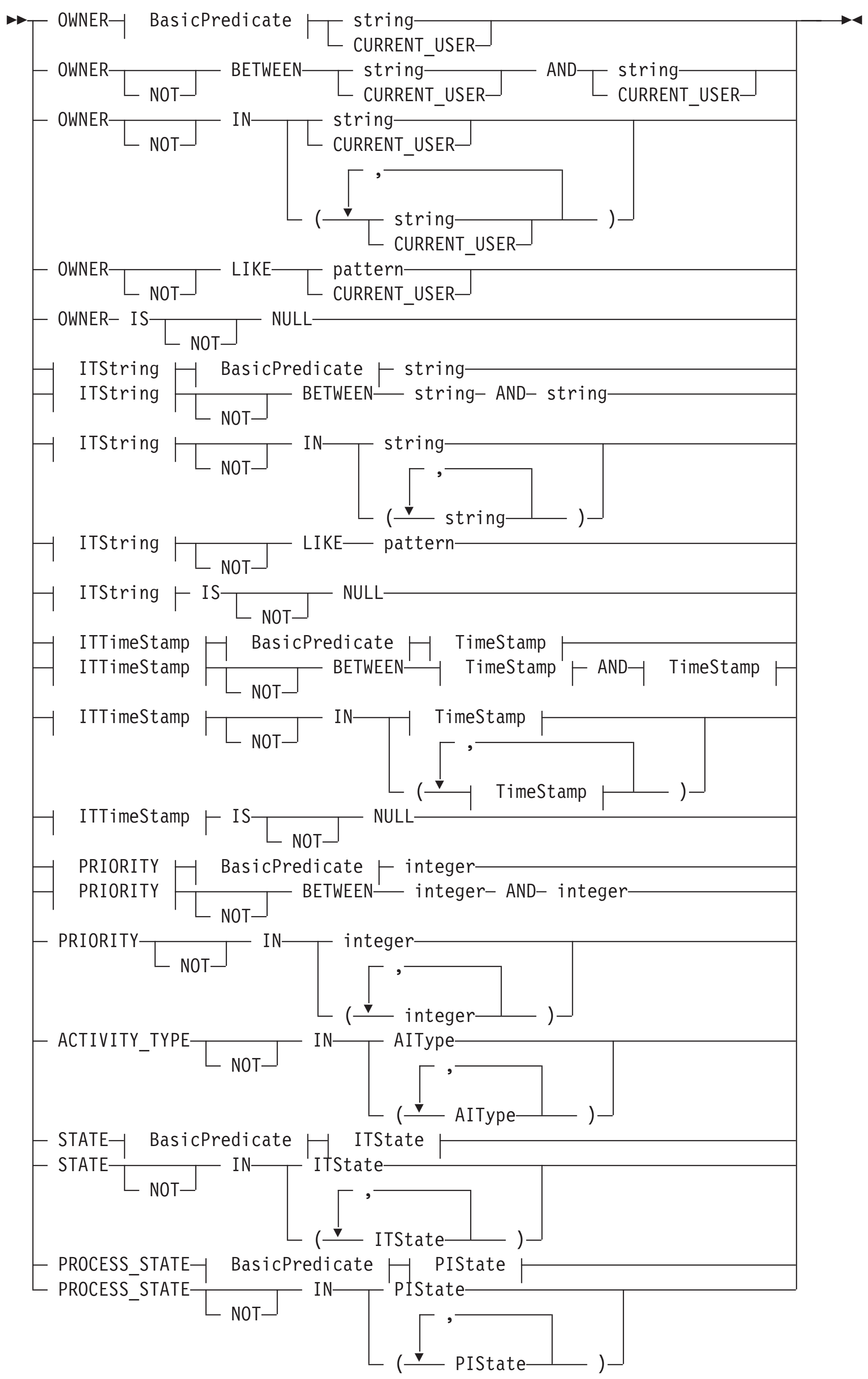
Note: A *string* constant is to be enclosed in single quotes (′).

A *pattern* is a string constant in which the asterisk and the question mark have special meanings.
- The question mark (?) represents any single character.
- The asterisk (*) represents a string of zero or more characters.
- The escape character is backslash (\) and must be used when the pattern itself contains actual question marks or asterisks.

ItemFilter



ITPredicate


OWNER BasicPredicate string CURRENT_USER
OWNER NOT BETWEEN string CURRENT_USER AND string CURRENT_USER
OWNER NOT IN string CURRENT_USER (, string CURRENT_USER)
OWNER NOT LIKE pattern CURRENT_USER
OWNER IS NOT NULL
ITString BasicPredicate string
ITString NOT BETWEEN string AND string
ITString NOT IN string (, string)
ITString NOT LIKE pattern
ITString IS NOT NULL
ITTimeStamp BasicPredicate TimeStamp
ITTimeStamp NOT BETWEEN TimeStamp AND TimeStamp
ITTimeStamp NOT IN TimeStamp (, TimeStamp)
ITTimeStamp IS NOT NULL
PRIORITY BasicPredicate integer
PRIORITY NOT BETWEEN integer AND integer
PRIORITY NOT IN integer (, integer)
ACTIVITY_TYPE NOT IN AIType (, AIType)
STATE BasicPredicate ITState
STATE NOT IN ITState (, ITState)
PROCESS_STATE BasicPredicate PIState
PROCESS_STATE NOT IN PIState (, PIState)

AIType

```
►►──┬─ PROCESS_ACTIVITY─┬──────────────────────────────────────────────►◄
    └─ PROGRAM_ACTIVITY─┘
```

BasicPredicate

```
►►──┬─ =──┬─────────────────────────────────────────────────────────────►◄
    ├─ >──┤
    ├─ >=─┤
    ├─ <──┤
    ├─ <=─┤
    └─ <>─┘
```

ITState

```
►►──┬─ READY──────────┬─────────────────────────────────────────────────►◄
    ├─ RUNNING────────┤
    ├─ FINISHED───────┤
    ├─ TERMINATED─────┤
    ├─ SUSPENDED──────┤
    ├─ DISABLED───────┤
    ├─ CHECKED_OUT────┤
    ├─ IN_ERROR───────┤
    ├─ EXECUTED───────┤
    ├─ PLANNING───────┤
    ├─ FORCE_FINISHED─┤
    ├─ TERMINATING────┤
    └─ SUSPENDING─────┘
```

ITString

```
►►──┬─ DESCRIPTION──────┬───────────────────────────────────────────────►◄
    ├─ NAME─────────────┤
    ├─ PROCESS_CATEGORY─┤
    └─ PROCESS_NAME─────┘
```

ITTimeStamp

```
►►──┬─ LAST_MODIFICATION_TIME─┬─────────────────────────────────────────►◄
    └─ RECEIVED_TIME──────────┘
```

PIState

```
►►──┬─ READY───────┬──────────────────────────────────────────────────────►◄
    ├─ RUNNING─────┤
    ├─ FINISHED────┤
    ├─ TERMINATED──┤
    ├─ SUSPENDED───┤
    ├─ TERMINATING─┤
    └─ SUSPENDING──┘
```

TimeStamp

```
►►──year─ -─ month─ -─ day──┬────────────────────────────────────────────┬──►◄
                            └─ ─ hours──┬─────────────────────────────┬──┘
                                        └─ :─ minutes──┬───────────┬──┘
                                                       └─ :─ seconds─┘
```

Items can be sorted. An item sort criterion is specified as a character string:

Note: The default sort order is ascending.

Activity types are sorted according to the sequence shown in the AIType diagram.

States are sorted according to the sequence shown in the ITState respectively the PIState diagram.

ItemOrderBy

```
                                 ┌─ , ──────────┐
                                 ▼              │
►►──┬─ ACTIVITY_TYPE──────┬──────┬──────────┬───┴───────────────────────────►◄
    ├┤ ITString ├─────────┤      ├─ ASC──┤
    ├┤ ITTimeStamp ├──────┤      └─ DESC─┘
    ├─ OWNER──────────────┤
    ├─ PRIORITY───────────┤
    ├─ PROCESS_STATE──────┤
    └─ STATE──────────────┘
```

The number of items to be retrieved can be restricted via a threshold which specifies the maximum number of items to be returned to the client. That threshold is applied after the items have been sorted according to the sort criteria specified. Note that the items are sorted on the server, that is, the code page of the server determines the sort sequence.

The primary information that is retrieved for each item is:
- ActivityType
- Category
- CreationTime
- Description
- Icon
- Implementation

- Kind
- LastModificationTime
- Name
- Owner
- Priority
- ProcessInstanceName
- ReceivedAs
- ReceivedTime
- StartTime
- State
- SupportTools

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjExecutionServiceQueryItems(
       FmcjExecutionServiceHandle service,
       char const *               filter,
       char const *               sortCriteria,
       unsigned long const *      threshold,
       FmcjItemHandle *           items )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjESQueryItems.

        FmcjESQueryItems.

           CALL    "FmcjExecutionServiceQueryItems"
                          USING
                          BY VALUE
                             serviceValue
                             filter
                             sortCriteria
                             threshold
                          BY REFERENCE
                             items
                          RETURNING
                             intReturnValue.
```

Parameters

service
Input. A handle to the service object representing the session with the execution server.

filter Input. The filter criteria which characterize the items to be retrieved.

sortCriteria
Input. The sort criteria to be applied to the items found.

threshold
Input. The threshold which defines the maximum number of items to be returned to the client.

items Input/Output. The qualifying vector of items.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_INVALID_FILTER(125)
The specified filter is invalid.

FMC_ERROR_INVALID_SORT(808)
The specified sort criteria are invalid.

FMC_ERROR_INVALID_THRESHOLD(807)
The specified threshold is invalid.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Examples

- For a C language example see "Query process instances (C language)" on page 455.
- For a Cobol example see "Query process instances (Cobol language)" on page 456.

FmcjExecutionServiceQueryProcessInstanceLists()

This function/subprogram retrieves the process instance lists the user has access to from the MQSeries Workflow execution server (action call).

Any process instance lists retrieved are appended to the supplied vector. If you want to read the current process instance lists only, you have to clear the vector before you call this function/subprogram. This means that you should set the vector handle to 0 via the FmcjXxxVectorDeallocate function.

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjExecutionServiceQueryProcessInstanceLists(
       FmcjExecutionServiceHandle            service,
       FmcjProcessInstanceListVectorHandle *  lists )
```

Cobol language signature

Cobol language signature

```
     PERFORM FmcjESQueryProcInstLists.

  FmcjESQueryProcInstLists.

     CALL    "FmcjExecutionServiceQueryProcessInstanceLists"
                  USING
                  BY VALUE
                     serviceValue
                  BY REFERENCE
                     lists
                  RETURNING
                     intReturnValue.
```

Parameters

service
: Input. A handle to the service object representing the session with the execution server.

lists
: Input/Output. The vector of process instance lists.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
: Timeout has occurred.

Examples

- For a C language example see "Query worklists (C language)" on page 447.
- For a Cobol example see "Query worklists (Cobol language)" on page 448.

FmcjExecutionServiceQueryProcessInstance Notifications()

This function/subprogram retrieves the process instance notifications the user has access to from the MQSeries Workflow execution server (action call).

Any process instance notifications retrieved are appended to the supplied vector. If you want to read the current process instance notifications only, you have to clear the vector before you call this function/subprogram. This means that you should set the vector handle to 0 via the FmcjXxxVectorDeallocate function.

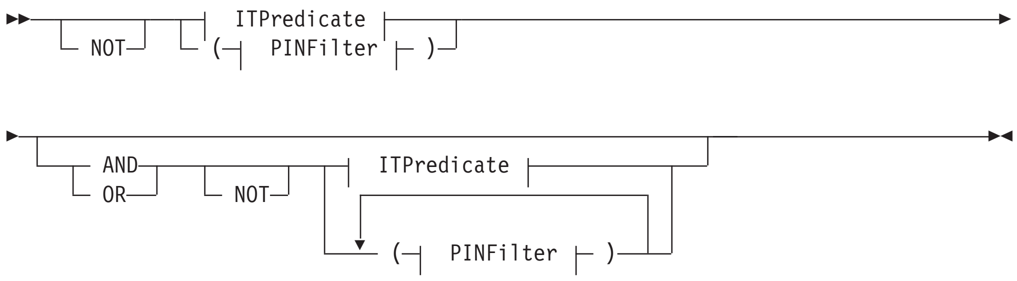
The process instance notifications to be retrieved can be characterized by a filter. A process instance notification filter is specified as a character string.

Note: A *string* constant is to be enclosed in single quotes (′).

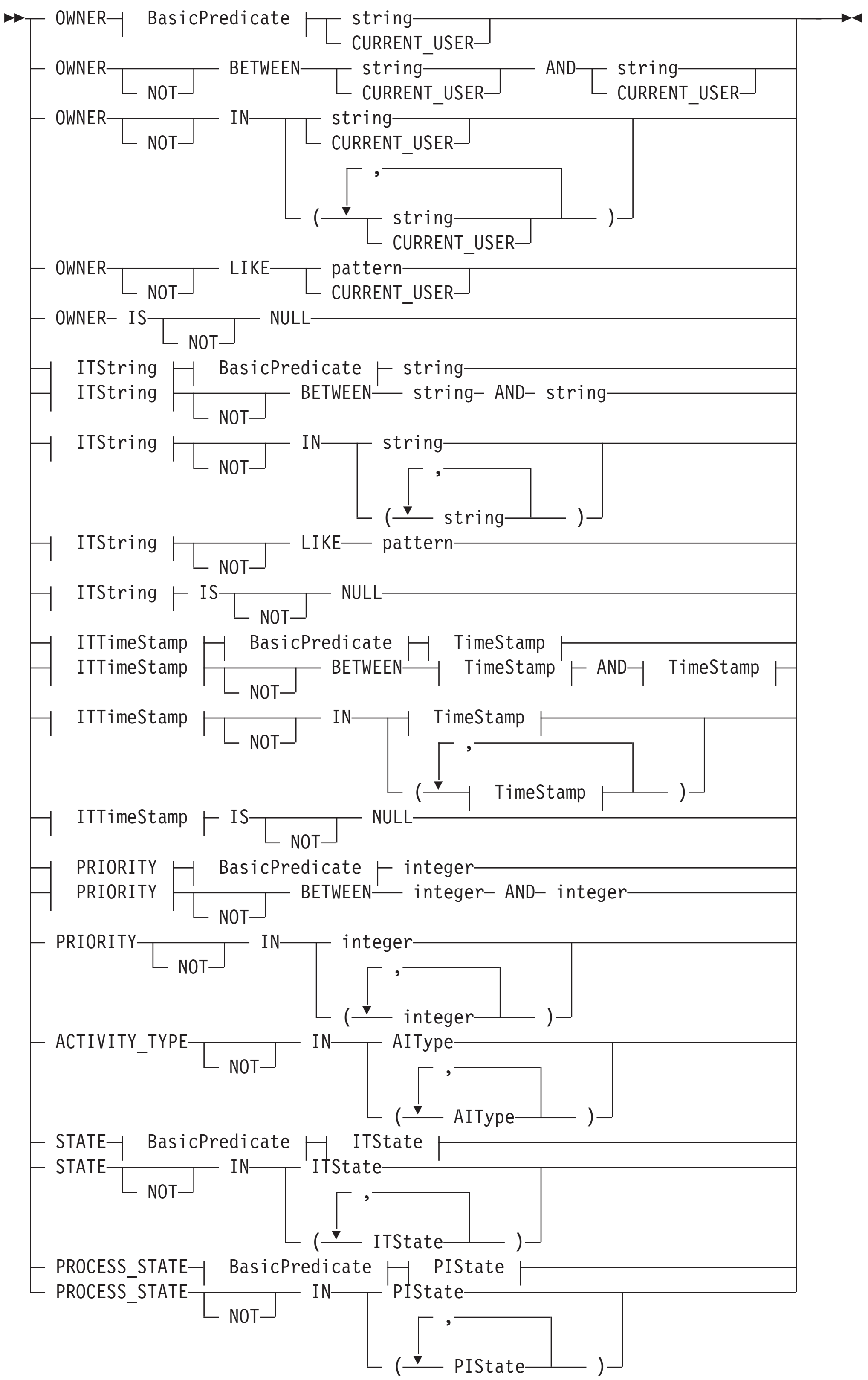
A *pattern* is a string constant in which the asterisk and the question mark have special meanings.

- The question mark (?) represents any single character.
- The asterisk (*) represents a string of zero or more characters.
- The escape character is backslash (\) and must be used when the pattern itself contains actual question marks or asterisks.

PINFilter



ITPredicate



OWNER BasicPredicate string CURRENT_USER
OWNER NOT BETWEEN string CURRENT_USER AND string CURRENT_USER
OWNER NOT IN string CURRENT_USER (, string CURRENT_USER)
OWNER NOT LIKE pattern CURRENT_USER
OWNER IS NOT NULL
ITString BasicPredicate string
ITString NOT BETWEEN string AND string
ITString NOT IN string (, string)
ITString NOT LIKE pattern
ITString IS NOT NULL
ITTimeStamp BasicPredicate TimeStamp
ITTimeStamp NOT BETWEEN TimeStamp AND TimeStamp
ITTimeStamp NOT IN TimeStamp (, TimeStamp)
ITTimeStamp IS NOT NULL
PRIORITY BasicPredicate integer
PRIORITY NOT BETWEEN integer AND integer
PRIORITY NOT IN integer (, integer)
ACTIVITY_TYPE NOT IN AIType (, AIType)
STATE BasicPredicate ITState
STATE NOT IN ITState (, ITState)
PROCESS_STATE BasicPredicate PIState
PROCESS_STATE NOT IN PIState (, PIState)

AIType

```
►►──┬─ PROCESS_ACTIVITY─┬──────────────────────────────────────────────────────►◄
    └─ PROGRAM_ACTIVITY─┘
```

BasicPredicate

```
►►──┬─ =──┬────────────────────────────────────────────────────────────────────►◄
    ├─ >──┤
    ├─ >=─┤
    ├─ <──┤
    ├─ <=─┤
    └─ <>─┘
```

ITState

```
►►──┬─ READY──────────┬────────────────────────────────────────────────────────►◄
    ├─ RUNNING────────┤
    ├─ FINISHED───────┤
    ├─ TERMINATED─────┤
    ├─ SUSPENDED──────┤
    ├─ DISABLED───────┤
    ├─ CHECKED_OUT────┤
    ├─ IN_ERROR───────┤
    ├─ EXECUTED───────┤
    ├─ PLANNING───────┤
    ├─ FORCE_FINISHED─┤
    ├─ TERMINATING────┤
    └─ SUSPENDING─────┘
```

ITString

```
►►──┬─ DESCRIPTION──────┬──────────────────────────────────────────────────────►◄
    ├─ NAME─────────────┤
    ├─ PROCESS_CATEGORY─┤
    └─ PROCESS_NAME─────┘
```

ITTimeStamp

```
►►──┬─ LAST_MODIFICATION_TIME─┬────────────────────────────────────────────────►◄
    └─ RECEIVED_TIME──────────┘
```

PIState

```
►►──┬─ READY───────┬──────────────────────────────────────────────────►◄
    ├─ RUNNING─────┤
    ├─ FINISHED────┤
    ├─ TERMINATED──┤
    ├─ SUSPENDED───┤
    ├─ TERMINATING─┤
    └─ SUSPENDING──┘
```

TimeStamp

```
►►──year─ -─ month─ -─ day──┬──────────────────────────────────────┬──►◄
                            └─ ─ hours──┬──────────────────────┬───┘
                                        └─ :─ minutes──┬────────────┬─┘
                                                       └─ :─ seconds─┘
```

Process instance notifications can be sorted. A process instance notification sort criterion is specified as a character string.

Note: The default sort order is ascending.

Activity types are sorted according to the sequence shown in the AIType diagram.

States are sorted according to the sequence shown in the ITState respectively the PIState diagram.

PINOrderBy

```
                                  ┌─ ,──────────┐
                                  ▼             │
►►──┬─ ACTIVITY_TYPE──────┬───────┬───────┬─────┴─────────────────────►◄
    ├─┤ ITString ├────────┤       ├─ ASC──┤
    ├─┤ ITTimeStamp ├─────┤       └─ DESC─┘
    ├─ OWNER──────────────┤
    ├─ PRIORITY───────────┤
    ├─ PROCESS_STATE──────┤
    └─ STATE──────────────┘
```

The number of process instance notifications to be retrieved can be restricted via a threshold which specifies the maximum number of process instance notifications to be returned to the client. That threshold is applied after the activity instance notifications have been sorted according to the sort criteria specified. Note that the process instance notifications are sorted on the server, that is, the code page of the server determines the sort sequence.

The primary information that is retrieved for each process instance notification is:
- Category
- CreationTime
- Description
- Icon
- Kind

- LastModificationTime
- Name
- Owner
- ProcessInstanceName
- ReceivedAs
- ReceivedTime
- StartTime
- State

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjExecutionServiceQueryProcessInstanceNotifications(
       FmcjExecutionServiceHandle                     service,
       char const *                                   filter,
       char const *                                   sortCriteria,
       unsigned long const *                          threshold,
       FmcjProcessInstanceNotificationVectorHandle *  notifications )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjESQueryProcInstNotifs.

        FmcjESQueryProcInstNotifs.

           CALL
               "FmcjExecutionServiceQueryProcessInstanceNotifications"
                          USING
                          BY VALUE
                             serviceValue
                             filter
                             sortCriteria
                             threshold
                          BY REFERENCE
                             notifications
                          RETURNING
                             intReturnValue.
```

Parameters

service
: Input. A handle to the service object representing the session with the execution server.

filter
: Input. The filter criteria which characterize the process instance notifications to be retrieved.

sortCriteria
: Input. The sort criteria to be applied to the process instance notifications found.

threshold
: Input. The threshold which defines the maximum number of process instance notifications to be returned to the client.

items
: Input/Output. The qualifying vector of process instance notifications.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_INVALID_FILTER(125)
: The specified filter is not applicable to process instance notifications.

FMC_ERROR_INVALID_SORT(808)
: The specified sort criteria are not applicable to process instance notifications.

FMC_ERROR_INVALID_THRESHOLD(807)
: The specified threshold is invalid.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Examples

- For a C language example see “Query process instances (C language)” on page 455.
- For a Cobol example see “Query process instances (Cobol language)” on page 456.

FmcjExecutionServiceQueryProcessInstances()

This function/subprogram retrieves the current process instances the user has access to from the MQSeries Workflow execution server (action call).

Any process instances retrieved are appended to the supplied vector. If you want to read the current process instances only, you have to clear the vector before you call this function/subprogram. This means that you should set the vector handle to 0 via the FmcjXxxVectorDeallocate function.

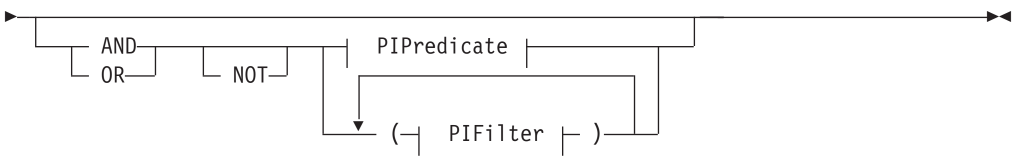
A filter on process instances is specified as a character string containing a filter predicate:

Note: A *string* constant is to be enclosed in single quotes (′).

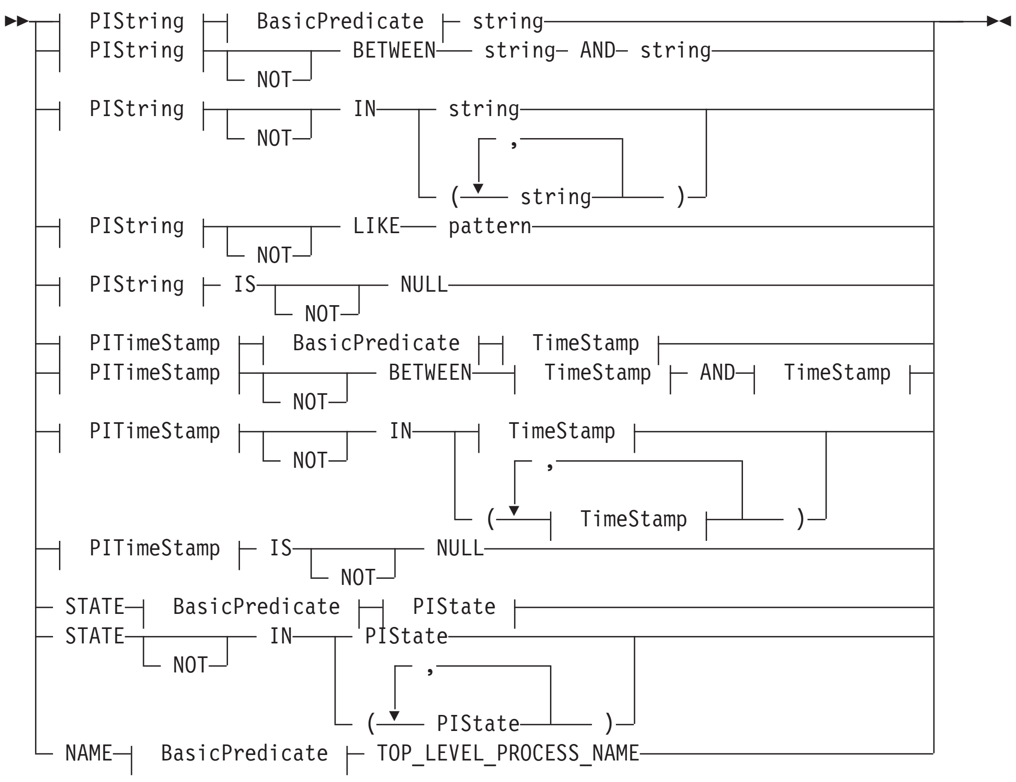
A *pattern* is a string constant in which the asterisk and the question mark have special meanings.
- The question mark (?) represents any single character.
- The asterisk (*) represents a string of zero or more characters.
- The escape character is backslash (\) and must be used when the pattern itself contains actual question marks or asterisks.

PIFilter

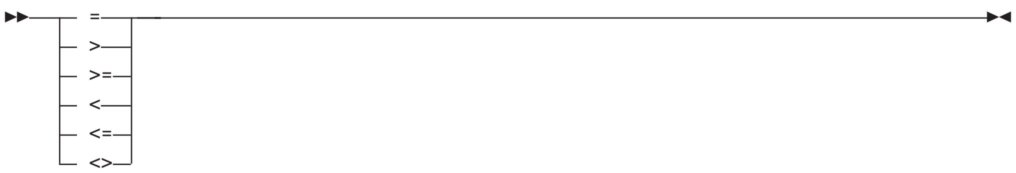
```
►►─┬─────┬─┬─┤ PIPredicate ├─────┬──────────────────────────────────►
   └─NOT─┘ └─(─┤ PIFilter ├─ )──┘
```



PIPredicate



BasicPredicate



PIState

```
►►──┬─ READY───────┬──────────────────────────────────────────────────────────►◄
    ├─ RUNNING─────┤
    ├─ FINISHED────┤
    ├─ TERMINATED──┤
    ├─ SUSPENDED───┤
    ├─ TERMINATING─┤
    └─ SUSPENDING──┘
```

PIString

```
►►──┬─ ADMINISTRATOR──────────┬───────────────────────────────────────────────►◄
    ├─ CATEGORY───────────────┤
    ├─ DESCRIPTION────────────┤
    ├─ NAME───────────────────┤
    ├─ PARENT_PROCESS_NAME────┤
    └─ TOP_LEVEL_PROCESS_NAME─┘
```

PITimeStamp

```
►►──┬─ LAST_MODIFICATION_TIME─┬───────────────────────────────────────────────►◄
    └─ LAST_STATE_CHANGE_TIME─┘
```

TimeStamp

```
►►──year─ -─ month─ -─ day──┬─────────────────────────────────────────┬───────►◄
                            └─ ─ hours──┬──────────────────────────┬──┘
                                        └─ :─ minutes──┬────────────┬──┘
                                                       └─ :─ seconds─┘
```

Process instances can be sorted. A process instance sort criterion is specified as a character string.

Note: The default sort order is ascending.

States are sorted according to the sequence shown in the PIState diagram.

PIOrderBy

```
                                ┌─ , ──────────┐
                                ▼              │
►►──┬─┤ PIString ├──────┬────────┬──────┬──────┴──────────────────────────────►◄
    ├─┤ PITimeStamp ├───┤        ├─ ASC──┤
    └─ STATE────────────┘        └─ DESC─┘
```

The number of process instances to be retrieved can be restricted via a threshold which specifies the maximum number of process instances to be returned to the client. That threshold is applied after the process instances have been sorted according to the sort criteria specified. Note that the process instances are sorted on the server, that is, the code page of the server determines the sort sequence.

The primary information that is retrieved for each process instance is:

- Category
- Description
- Icon
- InContainerNeeded
- LastModificationTime
- LastStateChangeTime
- Name
- ParentName
- ProcessTemplateName
- StartTime
- State
- SuspensionTime
- SystemName
- SystemGroupName
- TopLevelName

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjExecutionServiceQueryProcessInstances(
       FmcjExecutionServiceHandle        service,
       char const *                      filter,
       char const *                      sortCriteria,
       unsigned long const *             threshold,
       FmcjProcessInstanceVectorHandle * instances )
```

Cobol language signature

Cobol language signature

```
     PERFORM FmcjESQueryProcInsts.

   FmcjESQueryProcInsts.

     CALL    "FmcjExecutionServiceQueryProcessInstances"
                  USING
                  BY VALUE
                     serviceValue
                     filter
                     sortCriteria
                     threshold
                  BY REFERENCE
                     instances
                  RETURNING
                     intReturnValue.
```

Parameters

service
: Input. A handle to the service object representing the session with the execution server.

filter
: Input. The filter criteria which characterize the process instances to be retrieved.

sortCriteria
: Input. The sort criteria to be applied to the process instances found.

threshold
: Input. The threshold which defines the maximum number of process instances to be returned to the client.

instances
: Input/Output. The qualifying vector of process instances.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_INVALID_FILTER(125)
: The specified filter is not applicable to process instances.

FMC_ERROR_INVALID_SORT(808)
: The specified sort criteria are not applicable to process instances.

FMC_ERROR_INVALID_THRESHOLD(807)
: The specified threshold is invalid.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Examples

- For a C language example see “Query process instances (C language)” on page 455.
- For a Cobol example see “Query process instances (Cobol language)” on page 456.

FmcjExecutionServiceQueryProcessTemplateLists()

This function/subprogram retrieves the current process template lists the user has access to from the MQSeries Workflow execution server (action call).

Any process template lists retrieved are appended to the supplied vector. If you want to read the current process template lists only, you have to clear the vector before you call this function/subprogram. This means that you should set the vector handle to 0 via the FmcjXxxVectorDeallocate function.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjExecutionServiceQueryProcessTemplateLists(
       FmcjExecutionServiceHandle             service,
       FmcjProcessTemplateListVectorHandle *  lists )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjESQueryProcTemplLists.

        FmcjESQueryProcTemplLists.

           CALL    "FmcjExecutionServiceQueryProcessTemplateLists"
                          USING
                          BY VALUE
                             serviceValue
                          BY REFERENCE
                             lists
                          RETURNING
                             intReturnValue.
```

Parameters

service
Input. A handle to the service object representing the session with the execution server.

lists Input/Output. The vector of process template lists.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Examples

- For a C language example see “Query worklists (C language)” on page 447.
- For a Cobol example see “Query worklists (Cobol language)” on page 448.

FmcjExecutionServiceQueryProcessTemplates()

This function/subprogram retrieves the current process templates from the MQSeries Workflow execution server (action call).

Any process templates retrieved are appended to the supplied vector. If you want to read the current process templates only, you have to clear the vector before you call this function/subprogram. This means that you should set the vector handle to 0 via the FmcjXxxVectorDeallocate function.

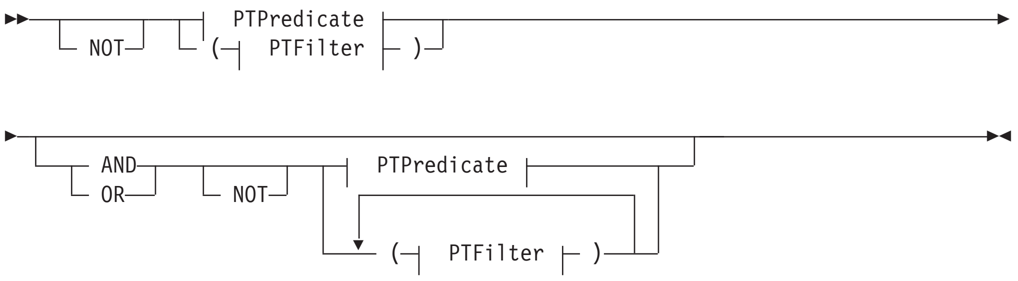
A filter on process templates is specified as a character string containing a filter predicate:

Note: A *string* constant is to be enclosed in single quotes (′).

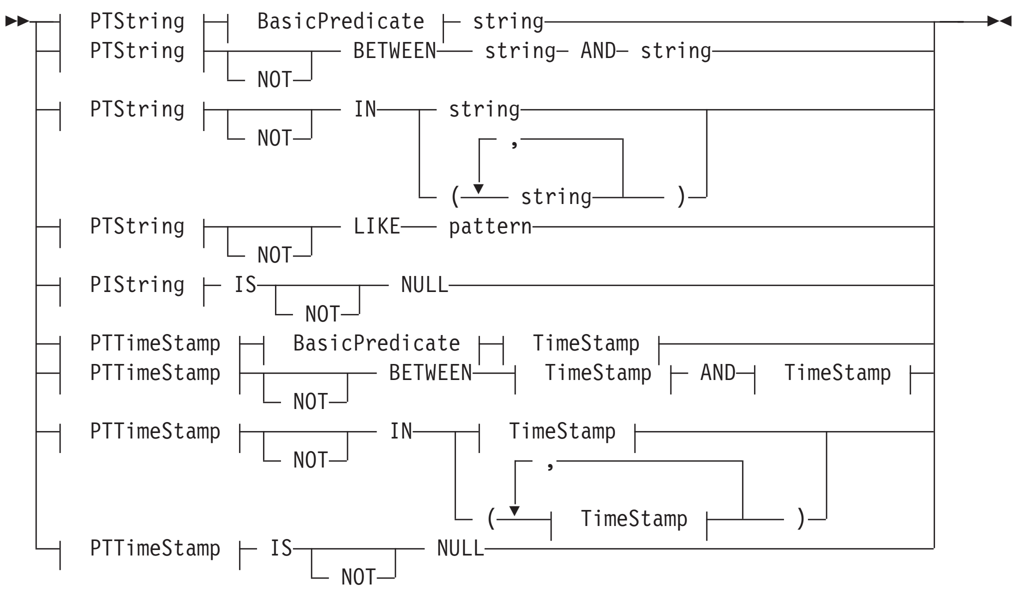
A *pattern* is a string constant in which the asterisk and the question mark have special meanings.

- The question mark (?) represents any single character.
- The asterisk (*) represents a string of zero or more characters.
- The escape character is backslash (\) and must be used when the pattern itself contains actual question marks or asterisks.

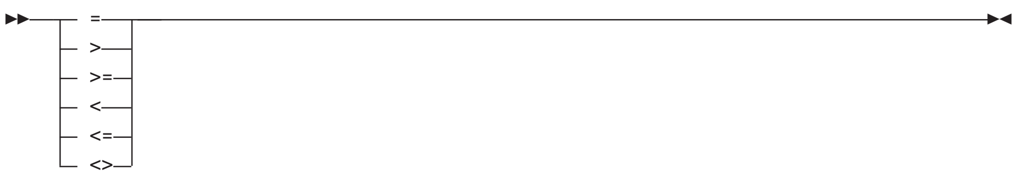
PTFilter



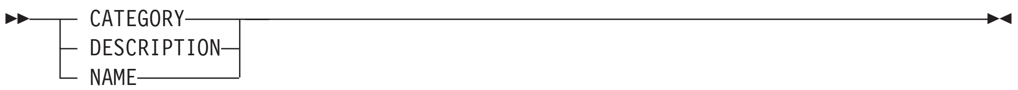
PTPredicate


PTString BasicPredicate string
PTString NOT BETWEEN string AND string
PTString NOT IN string , (string)
PTString NOT LIKE pattern
PIString IS NOT NULL
PTTimeStamp BasicPredicate TimeStamp
PTTimeStamp NOT BETWEEN TimeStamp AND TimeStamp
PTTimeStamp NOT IN TimeStamp , (TimeStamp)
PTTimeStamp IS NOT NULL

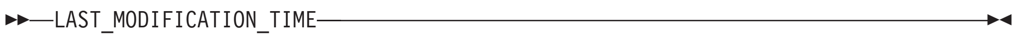
BasicPredicate



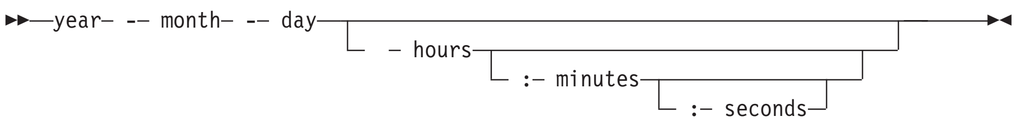
PTString



PTTimeStamp



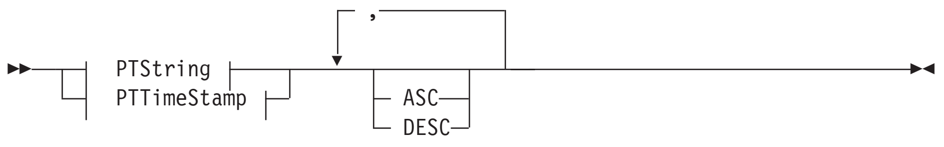
TimeStamp



Process templates can be sorted. A process template sort criterion is specified as a character string.

Note: The default sort order is ascending.

PTOrderBy



The number of process templates to be retrieved can be restricted via a threshold which specifies the maximum number of process templates to be returned to the client. That threshold is applied after the process templates have been sorted according to the sort criteria specified. Note that the process templates are sorted on the server, that is, the code page of the server determines the sort sequence.

The primary information that is retrieved for each process template is:
- Category
- CreationTime
- Description
- Icon
- InContainerNeeded
- LastModificationTime
- Name

Properties

Usage notes
- See "Action functions/subprograms" on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjExecutionServiceQueryProcessTemplates(
       FmcjExecutionServiceHandle        service,
       char const *                      filter,
       char const *                      sortCriteria,
       unsigned long const *             threshold,
       FmcjProcessTemplateVectorHandle *  templates )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjESQueryProcTempls.

        FmcjESQueryProcTempls.

           CALL    "FmcjExecutionServiceQueryProcessTemplates"
                          USING
                          BY VALUE
                             serviceValue
                             filter
                             sortCriteria
                             threshold
                          BY REFERENCE
                             templates
                          RETURNING
                             intReturnValue.
```

Parameters

service
: Input. A handle to the service object representing the session with the execution server.

filter
: Input. The filter criteria which characterize the process templates to be retrieved.

sortCriteria
: Input. The sort criteria to be applied to the process templates found.

threshold
: Input. The threshold which defines the maximum number of process templates to be returned to the client.

templates
: Input/Output. The qualifying vector of process templates.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_INVALID_FILTER(125)
The specified filter is not applicable to process templates.

FMC_ERROR_INVALID_SORT(808)
The specified sort criteria are not applicable to process templates.

FMC_ERROR_INVALID_THRESHOLD(807)
The specified threshold is invalid.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Examples

- For a C language example see "Query process instances (C language)" on page 455.
- For a Cobol example see "Query process instances (Cobol language)" on page 456.

FmcjExecutionServiceQueryWorkitems()

This function/subprogram retrieves the work items the user has access to from the MQSeries Workflow execution server (action call).

Any work items retrieved are appended to the supplied vector. If you want to read the current work items only, you have to clear the vector before you call this function/subprogram. This means that you should set the vector handle to 0 via the FmcjXxxVectorDeallocate function.

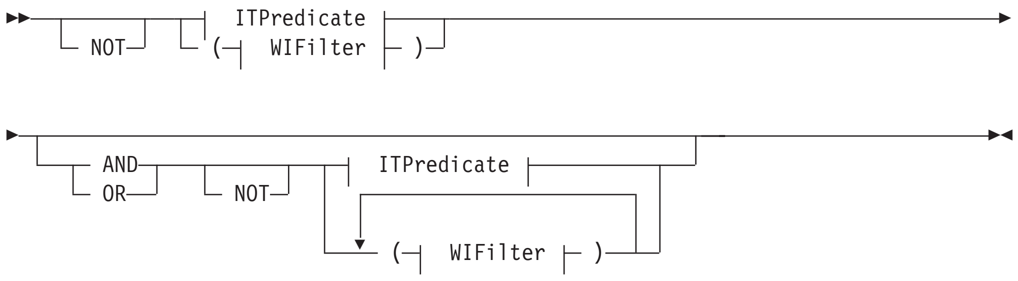
The work items to be retrieved can be characterized by a filter. A work item filter is specified as a character string:

Note: A *string* constant is to be enclosed in single quotes (′).

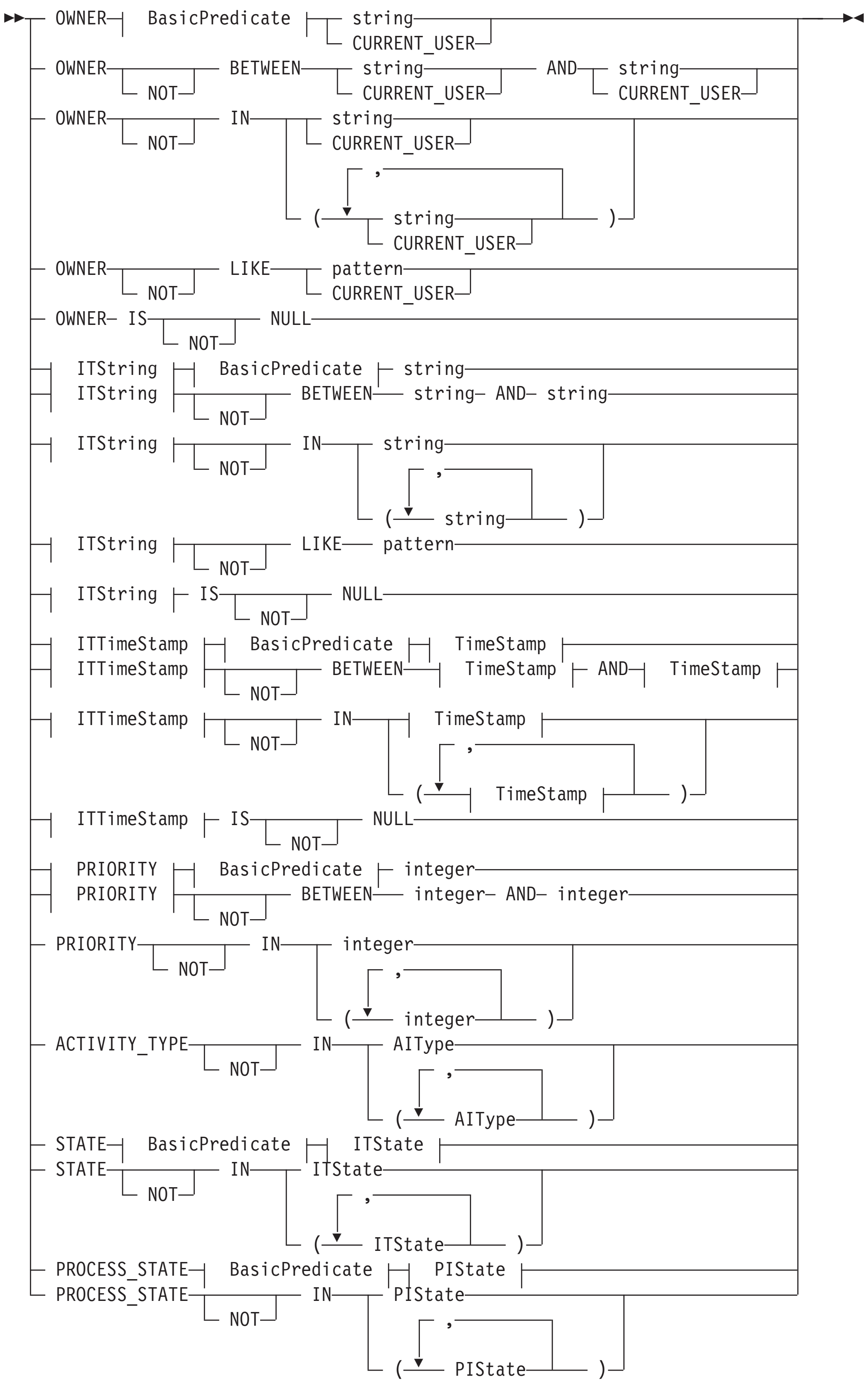
A *pattern* is a string constant in which the asterisk and the question mark have special meanings.

- The question mark (?) represents any single character.
- The asterisk (*) represents a string of zero or more characters.
- The escape character is backslash (\) and must be used when the pattern itself contains actual question marks or asterisks.

WIFilter



ITPredicate



OWNER BasicPredicate string CURRENT_USER
OWNER NOT BETWEEN string CURRENT_USER AND string CURRENT_USER
OWNER NOT IN string CURRENT_USER (, string CURRENT_USER)
OWNER NOT LIKE pattern CURRENT_USER
OWNER IS NOT NULL
ITString BasicPredicate string
ITString NOT BETWEEN string AND string
ITString NOT IN string (, string)
ITString NOT LIKE pattern
ITString IS NOT NULL
ITTimeStamp BasicPredicate TimeStamp
ITTimeStamp NOT BETWEEN TimeStamp AND TimeStamp
ITTimeStamp NOT IN TimeStamp (, TimeStamp)
ITTimeStamp IS NOT NULL
PRIORITY BasicPredicate integer
PRIORITY NOT BETWEEN integer AND integer
PRIORITY NOT IN integer (, integer)
ACTIVITY_TYPE NOT IN AIType (, AIType)
STATE BasicPredicate ITState
STATE NOT IN ITState (, ITState)
PROCESS_STATE BasicPredicate PIState
PROCESS_STATE NOT IN PIState (, PIState)

AIType

```
▶▶──┬─ PROCESS_ACTIVITY─┬─────────────────────────────────────────▶◀
    └─ PROGRAM_ACTIVITY─┘
```

BasicPredicate

```
▶▶──┬─ =──┬───────────────────────────────────────────────────────▶◀
    ├─ >──┤
    ├─ >=─┤
    ├─ <──┤
    ├─ <=─┤
    └─ <>─┘
```

ITState

```
▶▶──┬─ READY──────────┬───────────────────────────────────────────▶◀
    ├─ RUNNING────────┤
    ├─ FINISHED───────┤
    ├─ TERMINATED─────┤
    ├─ SUSPENDED──────┤
    ├─ DISABLED───────┤
    ├─ CHECKED_OUT────┤
    ├─ IN_ERROR───────┤
    ├─ EXECUTED───────┤
    ├─ PLANNING───────┤
    ├─ FORCE_FINISHED─┤
    ├─ TERMINATING────┤
    └─ SUSPENDING─────┘
```

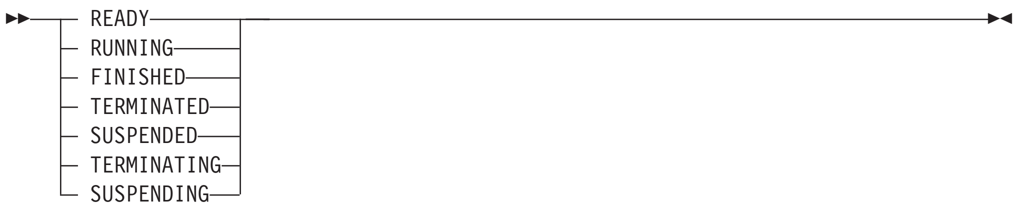
ITString

```
▶▶──┬─ DESCRIPTION──────┬─────────────────────────────────────────▶◀
    ├─ NAME─────────────┤
    ├─ PROCESS_CATEGORY─┤
    └─ PROCESS_NAME─────┘
```

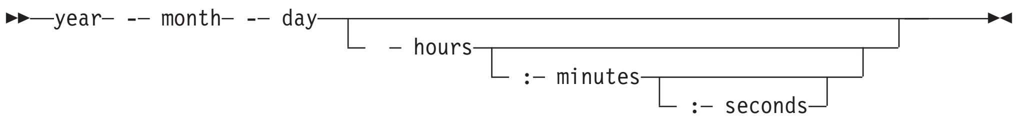
ITTimeStamp

```
▶▶──┬─ LAST_MODIFICATION_TIME─┬───────────────────────────────────▶◀
    └─ RECEIVED_TIME──────────┘
```

PIState



TimeStamp



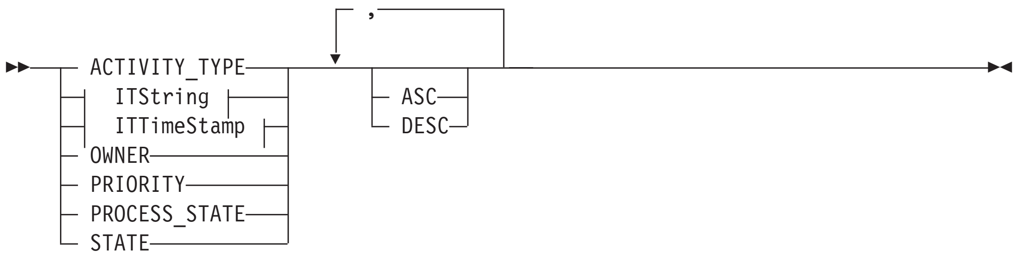
Work items can be sorted. A work item sort criterion is specified as a character string.

Note: The default sort order is ascending.

Activity types are sorted according to the sequence shown in the AIType diagram.

States are sorted according to the sequence shown in the ITState respectively the PIState diagram.

WIOrderBy



The number of work items to be retrieved can be restricted via a threshold which specifies the maximum number of work items to be returned to the client. That threshold is applied after the items have been sorted according to the sort criteria specified. Note that the items are sorted on the server, that is, the code page of the server determines the sort sequence.

The primary information that is retrieved for each work item is:
- ActivityType
- Category
- CreationTime
- Description
- Icon

- Implementation
- Kind
- LastModificationTime
- Name
- Owner
- Priority
- ProcessInstanceName
- ReceivedAs
- ReceivedTime
- StartTime
- State
- SupportTools

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjExecutionServiceQueryWorkitems(
       FmcjExecutionServiceHandle      service,
       char const *                    filter,
       char const *                    sortCriteria,
       unsigned long const *           threshold,
       FmcjWorkitemHandle *            workitems )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjESQueryWorkitems.

        FmcjESQueryWorkitems.

           CALL    "FmcjExecutionServiceQueryWorkitems"
                          USING
                          BY VALUE
                             serviceValue
                             filter
                             sortCriteria
                             threshold
                          BY REFERENCE
                             workitems
                          RETURNING
                             intReturnValue.
```

Parameters

service
Input. A handle to the service object representing the session with the execution server.

filter Input. The filter criteria which characterize the work items to be retrieved.

sortCriteria
Input. The sort criteria to be applied to the work items found.

threshold
Input. The threshold which defines the maximum number of work items to be returned to the client.

workitems
Input/Output. The qualifying vector of work items.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_INVALID_FILTER(125)
The specified filter is not applicable to work items.

FMC_ERROR_INVALID_SORT(808)
The specified sort criteria are not applicable to work items.

FMC_ERROR_INVALID_THRESHOLD(807)
The specified threshold is invalid.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Examples

- For a C language example see "Query process instances (C language)" on page 455.
- For a Cobol example see "Query process instances (Cobol language)" on page 456.

FmcjExecutionServiceQueryWorklists()

This function/subprogram retrieves the worklists the user has access to from the MQSeries Workflow execution server (action call).

Any worklists retrieved are appended to the supplied vector. If you want to read the current worklists only, you have to clear the vector before you call this function/subprogram. This means that you should set the vector handle to 0 via the FmcjXxxVectorDeallocate function.

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjExecutionServiceQueryWorklists(
       FmcjExecutionServiceHandle   service,
       FmcjWorklistVectorHandle *   lists )
```

Cobol language signature

Cobol language signature

```
          PERFORM FmcjESQueryWorklists.

       FmcjESQueryWorklists.

          CALL    "FmcjExecutionServiceQueryWorklists"
                         USING
                         BY VALUE
                            serviceValue
                         BY REFERENCE
                            lists
                         RETURNING
                            intReturnValue.
```

Parameters

service
: Input. A handle to the service object representing the session with the execution server.

lists
: Input/Output. The vector of worklists.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
: Timeout has occurred.

Examples

- For a C language example see “Query worklists (C language)” on page 447.
- For a Cobol example see “Query worklists (Cobol language)” on page 448.

FmcjExecutionServiceReceive()

This function/subprogram allows for receiving data pushed by an MQSeries Workflow execution server (action call).

The correlation ID can be used to receive specific data. Currently, it **must** specify Fmcj_No_CorrelID in order to receive any data sent. Note that this parameter is changed by a successful retrieval of data. You **must** reset it for each request.

The timeout value specifies how long the application should wait at a maximum for some data to arrive. If no data arrives, a timeout error is indicated. A timeout value of -1 indicates an indefinite wait time.

If data is successfully received, the execution data contains the data sent and can be used for updating objects or for creating new objects. If an object is to be updated, the persistent object ID of the object described by the execution data can be obtained, the object can be searched for, and its Update() function/subprogram can be applied. If an object is to be created, the type of the object described by the execution data can be obtained, and the appropriate object can be created from the execution data. See "FmcjExecutionData" on page 188 for functions/subprograms supported by the execution data.

The following enumeration constants can be used to determine the contents of the execution data received; it is strongly advised to use the symbolic names instead of the integer values:

C language	integer value
Fmc_DART_Terminate	2
Fmc_DART_ItemDeleted	1000
Fmc_DART_ItemUpdated	1001
Fmc_DART_Workitem	1002
Fmc_DART_ActivityInstanceNotification	1003
Fmc_DART_ProcessInstanceNotification	1004

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server (present session mode)

API include file

fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjExecutionServiceReceive(
       FmcjExecutionServiceHandle   service,
       FmcjCorrelID *               correlID,
       FmcjExecutionDataHandle *    data,
       signed long                  timeout )
```

Cobol language sigature

Cobol language signature

```
            PERFORM FmcjESReceive.

         FmcjESReceive.

            CALL    "FmcjExecutionServiceReceive"
                           USING
                           BY VALUE
                              serviceValue
                           BY REFERENCE
                              correlID
                              data
                           BY VALUE
                              timeoutValue
                           RETURNING
                              intReturnValue.
```

Parameters

service
: Input. A handle to the service object representing the present session with the execution server.

correlID
: Input/Output. The correlation ID by which this data can be correlated to a previous request. Must currently be Fmcj_No_CorrelID for each request.

data
: Output. The data sent by an MQSeries Workflow execution server.

timeout
: Input. The maximum time period to wait for some data to arrive.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is incorrect; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; the server to which the connection should be established is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Examples

- For a C language example see "Query worklists (C language)" on page 447.
- For a Cobol example see "Query worklists (Cobol language)" on page 448.

FmcjExecutionServiceRemotePassthrough()

This function/subprogram can be used by an application program to establish a user session with an MQSeries Workflow execution server from within this program (activity-implementation call).

When that activity implementation or support tool decides to distribute work among other programs and starts those programs as separate operating system processes, then those processes have to get passed the COMMAREA/IMS I/O Area be able to retrieve the information needed.

When successfully executed, a session to the same execution server is set up from where the original work item or support tool was started; the user on whose behalf the session is set up is the same one on whose behalf the original work item or support tool was started.

Note that this function/subprogram call will fail after the COMMAREA/ I/O Area has been changed with FmcjContainerSetOutContainer or FmcjContainerSetRemoteOutContainer.

Properties

Usage notes

- See "Activity implementation functions/subprograms" on page 76 for general information.
- ''

Authorization

Valid program identification

Required connection

None but active MQSeries Workflow program execution server

API include file

fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjExecutionServiceRemotePassthrough(
       FmcjExecutionServiceHandle   service ,
       char const *                 programID )
```

Cobol language signature

Cobol language signature

```
            PERFORM FmcjESRemotePassthrough.

         FmcjESRemotePassthrough.

            CALL    "FmcjExecutionServiceRemotePassthrough"
                           USING
                           BY VALUE
                              serviceValue
                           BY REFERENCE
                              programID
                           RETURNING
                              intReturnValue.
```

Parameters

service
: Input. A handle to the service object representing the session to be established with the execution server.

programID
: Input. The program identification by which the starting activity implementation or support tool is known to the program execution server.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_PROGRAM_EXECUTION(126)
: Passthrough was not called from within an activity implementation.

FMC_ERROR_INVALID_PROGRAMID(135)
: The program identification is invalid.

FMC_ERROR_USERID_UNKNOWN(10)
: The user who started the work item does no longer exist.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjExecutionServiceTerminateReceive()

This function/subprogram causes information to be placed into the client input queue to tell that receiving data from an MQSeries Workflow execution server can end (action call).

In this way, the receiving part of the application gets to know that receiving data can end. Any resulting actions are up to the application.

Note: The correlation ID passed must be Fmcj_No_CorrelID for each request.

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server (present session mode)

API include file

fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjExecutionServiceTerminateReceive(
       FmcjExecutionServiceHandle  service,
       FmcjCorrelID *              correlID )
```

Cobol language signature

Cobol language signature

```
          PERFORM FmcjESTerminateReceive.

       FmcjESTerminateReceive.

          CALL    "FmcjExecutionServiceTerminateReceive"
                         USING
                         BY VALUE
                            serviceValue
                         BY REFERENCE
                            correlID
                         RETURNING
                            intReturnValue.
```

Parameters

service
: Input. A handle to the service object representing the present session with the execution server.

correlID
: Input/Output. The correlation ID by which this data can be correlated to a previous request; must be Fmcj_No_CorrelID for each request.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is incorrect; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; the server to which the connection should be established is not defined in your profile.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
: Timeout has occurred.

Examples

- For a C language example see “Query worklists (C language)” on page 447.
- For a Cobol example see “Query worklists (Cobol language)” on page 448.

Chapter 36. FmcjItem functions/subprograms

An FmcjItem object represents a work item or an activity instance notification or a process instance notification.

An FmcjItem object represents the common aspects of work items and notifications. In the C++ implementation, it is thus the superclass of the FmcjWorkitem, FmcjActivityInstanceNotification, and FmcjProcessInstanceNotification classes. Similarly, in the C and Cobol language, it provides for common implementations of functions.

An item is uniquely identified by its object identifier.

FmcjItemDelete()

This function/subprogram deletes the specified item from the MQSeries Workflow execution server (client-server call).

The item must be in states *Ready*, *Finished*, *ForceFinished*, or *Disabled*. If the item is in the *Ready* state and represents the only work associated with the activity instance, then deletion is rejected.

There are no impacts on the transient representation of your item; you have to destruct or deallocate the transient object when it is no longer needed.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

One of:

- Be the item owner
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjItemDelete(
       FmcjItemHandle    hdlItem )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjItemDelete.

        FmcjItemDelete.

           CALL    "FmcjItemDelete"
                          USING
                          BY VALUE
                             hdlItem
                          RETURNING
                             intReturnValue.
```

Parameters

hdlItem
Input. The handle of the item to be deleted.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The item does no longer exist.

FMC_ERROR_NOT_AUTHORIZED(119)
Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_WRONG_STATE(120)
The item is in the wrong state.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Examples

- For a C language example see
- For a Cobol example see

FmcjItemObtainProcessInstanceMonitor()

This function/subprogram retrieves the process instance monitor for the process instance the item is part of from the MQSeries Workflow execution server (action call).

When the deep option is specified, then activity instances of type Block are resolved, that is, their block instance monitors are also fetched from the server.

Note: Deep is currently not supported.

The application is completely responsible for the ownership of objects, that is, it is not checked whether the process instance monitor handle already points to some object when a new one is assigned.

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

One of:

- Process authorization
- Process administration authorization
- Be the process administrator
- Be the process creator
- Be the system administrator

Required connection

MQSeries Workflow execution server (present session mode)

API include file

fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjItemObtainProcessInstanceMonitor(
       FmcjItemHandle                      hdlItem,
       bool                                deep,
       FmcjProcessInstanceMonitorHandle *  monitor)
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjAINObtainProcInstMon.

        FmcjAINObtainProcInstMon.

           CALL    "FmcjItemObtainProcessInstanceMonitor"
                          USING
                          BY VALUE
                             hdlItem
                             deep
                          BY REFERENCE
                             monitor
                          RETURNING
                             intReturnValue.
```

Parameters

hdlItem
: Input. The item whose process instance monitor is to be retrieved.

deep
: Input. An indicator whether activity instances of typed Block are to be resolved, that is, their monitor is also to be provided. Note, deep is currently ignored.

monitor
: Input/Output. The address of the handle to the process instance monitor respectively the process instance monitor object to be set.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_EMPTY(122)
: The object has not yet been read from the database, that is, does not yet represent a persistent one.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is incorrect; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The item does no longer exist.

FMC_ERROR_NOT_AUTHORIZED(119)
: Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; the server to which the connection should be established is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjItemProcessInstance()

This function/subprogram retrieves the process instance the item is a part of from the MQSeries Workflow execution server (action call).

All information about the process instance, primary and secondary, is retrieved.

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

One of:

- Process authorization
- Process administration authorization
- Be the process creator
- Be the process administrator
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjItemProcessInstance(
      FmcjItemHandle              hdlItem,
      FmcjProcessInstanceHandle *  instance )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjItemProcInst.

        FmcjItemProcInst.

           CALL    "FmcjItemProcessInstance"
                          USING
                          BY VALUE
                             hdlItem
                          BY REFERENCE
                             instance
                          RETURNING
                             intReturnValue.
```

Parameters

hdlItem
: Input. The handle of the item object to be queried.

instance
: Input/Output. The process instance object to be retrieved (initialized).

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The item does no longer exist.

FMC_ERROR_NOT_AUTHORIZED(119)
: Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
: Timeout has occurred.

FmcjItemRefresh()

This function/subprogram refreshes the item from the MQSeries Workflow execution server (action call).

All information about the item, primary and secondary, is retrieved.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjItemRefresh( FmcjItemHandle hdlItem )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjItemRefresh.

        FmcjItemRefresh.

           CALL    "FmcjItemRefresh"
                          USING
                          BY VALUE
                             hdlItem
                          RETURNING
                             intReturnValue.
```

Parameters

hdlItem
: Input. The handle of the item object to be refreshed.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The item does no longer exist.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjItemSetDescription()

This function/subprogram sets the description of the item to the specified value (action call).

If no description is provided (NULL pointer), the description of the item is erased.

The following rules apply for specifying an item description:
- You can specify a maximum of 254 characters.
- You can use any printable characters depending on your current locale, including the end-of-line and new-line characters.

Properties

Usage notes
- See "Action functions/subprograms" on page 76 for general information.

Authorization

One of:
- Be the item owner
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjItemSetDescription(
       FmcjItemHandle   hdlItem,
       char const *     description )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjItemSetDescription.

        FmcjItemSetDescription.

           CALL    "FmcjItemSetDescription"
                          USING
                          BY VALUE
                             hdlItem
                             description
                          RETURNING
                             intReturnValue.
```

Parameters

hdlItem
: Input. The handle of the item object whose description is to be set.

description
: Input. A pointer to the description to be set; can be a NULL pointer.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The item does no longer exist.

FMC_ERROR_INVALID_DESCRIPTION(810)
: The description does not conform to the syntax rules.

FMC_ERROR_NOT_AUTHORIZED(119)
: Not authorized.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjItemSetName()

This function/subprogram sets the name of the item (action call).

The following rules apply for specifying an item name:

- You can specify a maximum of 32 characters.
- You can use any printable characters depending on your current locale, except the following:
  `! " ' ( ) * + , - . / : : < = > [ \ ] ˜`
- You can use blanks with these restrictions: no leading blanks, no trailing blanks, and no consecutive blanks.
- You cannot use leading digits
- You cannot use keywords AND, OR, NOT, IS, NULL, DIV, MOD, MUL, LOWER, UPPER, VALUE

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

One of:

- Be the item owner
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjItemSetName(
       FmcjItemHandle   hdlItem,
       char const *     name )
```

Cobol language signature

Cobol language signature

```
            PERFORM FmcjItemSetName.

         FmcjItemSetName.

            CALL    "FmcjItemSetName"
                           USING
                           BY VALUE
                              hdlItem
                              name
                           RETURNING
                              intReturnValue.
```

Parameters

hdlItem
: Input. The handle of the item to be dealt with.

name
: Input. The new name of the item.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The work item does no longer exist.

FMC_ERROR_INVALID_NAME(134)
: The name does not conform to the syntax rules.

FMC_ERROR_NOT_AUTHORIZED(119)
: Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjItemTransfer()

This function/subprogram transfers an item to the specified user (action call).

The item must be in states *Ready*, *InError*, *Executed*, *Suspending*, or *Suspended* and the associated process instance in states *Running*, *Suspending*, or *Suspended*.

The user who transfers the item must have work item authorization for the new user.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

One of:

- Be the item owner
- Work item authorization
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjItemTransfer(
       FmcjItemHandle  hdlItem,
       char const *    userID )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjItemTransfer.

        FmcjItemTransfer.

           CALL    "FmcjItemTransfer"
                          USING
                          BY VALUE
                             hdlItem
                             userID
                          RETURNING
                             intReturnValue.
```

Parameters

hdlItem
Input. The handle of the item object to be transferred.

userID
Input. The ID of the user to whom the item is to be transferred.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The item does no longer exist.

FMC_ERROR_NEW_OWNER_ABSENT(110)
The user to whom the item is to be transferred is absent, that is, the item is not transferred.

FMC_ERROR_NEW_OWNER_NOT_FOUND(107)
The user to whom the item is to be transferred is unknown.

FMC_ERROR_NOT_AUTHORIZED(119)
Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_OWNER_ALREADY_ASSIGNED(133)
The user to whom the item is to be transferred does already have that item.

FMC_ERROR_WRONG_STATE(120)
The item is in the wrong state.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Chapter 37. FmcjPersistentList functions/subprograms

An FmcjPersistentList object represents a set of objects of the same type the user is authorized for. Moreover, all objects which are accessible through this list have the same characteristics. These characteristics are specified by a filter. Additionally, sort criteria can be applied and, after that, a threshold to restrict the number of objects to be transferred from a server to the client.

As the name indicates, the list definition is stored persistently. The objects contained in the list are, however, assembled dynamically when they are queried.

A persistent list can be a process template list, a process instance list, or a worklist.

An FmcjPersistentList object represents the common aspects of persistent lists. In the C++ implementation, it is the superclass of the FmcjProcessTemplateList, the FmcjProcessInstanceList, and the FmcjWorklist classes. Similarly, in the C and Cobol language, it provides for common implementations of functions.

A persistent list is uniquely identified by its name, type, and owner. It can be defined for general access purposes; it is then of a *public* type. Or, it can be defined for some specific user; it is then of a *private* type.

FmcjPersistentListDelete()

This function/subprogram deletes the specified persistent list from the MQSeries Workflow execution server (action call).

The transient representation of the persistent list is not impacted; you have to destruct or deallocate the transient object when it is no longer needed.

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

One of:

- Be the owner of the list
- Staff definition
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjPersistentListDelete(
       FmcjPersistentListHandle  hdlList )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjPLDelete.

        FmcjPLDelete.

           CALL    "FmcjPersistentListDelete"
                          USING
                          BY VALUE
                             hdlList
                          RETURNING
                             intReturnValue.
```

Parameters

hdlList
: Input. The handle of the persistent list to be deleted.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_NOT_AUTHORIZED(119)
: Not authorized to use the function/subprogram.

FMC_ERROR_DOES_NOT_EXIST(118)
: The persistent list does no longer exist.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
: Timeout has occurred.

FmcjPersistentListRefresh()

This function/subprogram refreshes the persistent list from the MQSeries Workflow execution server (action call).

All information about the persistent list is retrieved.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjPersistentListRefresh(
       FmcjPersistentListHandle  hdlList )
```

Cobol language signature

Cobol language signature

```
            PERFORM FmcjPLRefresh.

         FmcjPLRefresh.

            CALL    "FmcjPersistentListRefresh"
                           USING
                           BY VALUE
                              hdlList
                           RETURNING
                              intReturnValue.
```

Parameters

hdlList
Input. The handle of the persistent list to be refreshed.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The persistent list does no longer exist.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjPersistentListSetDescription()

This function/subprogram sets the description of the persistent list to the specified value (action call).

If no description is provided (NULL pointer), the description of the persistent list is erased.

The following rules apply for specifying a persistent list description:
- You can specify a maximum of 254 characters.
- You can use any printable characters depending on your current locale, including the end-of-line and new-line characters.

Properties

Usage notes
- See “Action functions/subprograms” on page 76 for general information.

Authorization

One of:
- Be the owner of the list
- Staff definition
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjPersistentListSetDescription(
       FmcjPersistentListHandle  hdlList,
       char const *              description )
```

Cobol language signature

Cobol language signature

```
            PERFORM FmcjPLSetDescription.

         FmcjPLSetDescription.

            CALL    "FmcjPersistentListSetDescription"
                           USING
                           BY VALUE
                              hdlList
                              description
                           RETURNING
                              intReturnValue.
```

Parameters

hdlList
Input. The handle of the persistent list object whose description is to be set.

description
Input. A pointer to the description to be set; can be a NULL pointer.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The persistent list does no longer exist.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjPersistentListSetFilter()

This function/subprogram sets the filter of the persistent list to the specified value (action call).

If no filter is provided (NULL pointer), the current filter of the persistent list is erased. This means that all objects authorized for will be selected via this list.

Refer to the appropriate list creation for a description of a valid filter syntax.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

One of:

- Be the owner of the list
- Staff definition
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjPersistentListSetFilter(
       FmcjPersistentListHandle  hdlList,
       char const *              filter )
```

Cobol language signature

Cobol language signature

```
          PERFORM FmcjPLSetFilter.

       FmcjPLSetFilter.

          CALL    "FmcjPersistentListSetFilter"
                         USING
                         BY VALUE
                            hdlList
                            filter
                         RETURNING
                            intReturnValue.
```

Parameters

hdlList
Input. The handle of the persistent list object whose filter is to be set.

filter Input. A pointer to the filter to be set; can be a NULL pointer.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The persistent list does no longer exist.

FMC_ERROR_INVALID_FILTER(125)
The specified filter is invalid.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjPersistentListSetSortCriteria()

This function/subprogram sets the sort criteria of the persistent list to the specified value (action call).

If no sort criteria are provided (NULL pointer), the current sort criteria of the persistent list are erased. This means that objects selected via this list will not be sorted.

Refer to the appropriate list creation for a description of a valid sort criteria syntax.

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

One of:

- Be the owner of the list
- Staff definition
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjPersistentListSetFilter(
       FmcjPersistentListHandle  hdlList,
       char const *              sortCriteria )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjPLSetSortCriteria.

        FmcjPLSetSortCriteria.

           CALL    "FmcjPersistentListSetSortCriteria"
                          USING
                          BY VALUE
                             hdlList
                             sortCriteria
                          RETURNING
                             intReturnValue.
```

Parameters

hdlList
: Input. The handle of the persistent list object whose sort criteria are to be set.

sortCriteria
: Input. A pointer to the sort criteria to be set; can be a NULL pointer.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The persistent list does no longer exist.

FMC_ERROR_INVALID_SORT(808)
: The specified sort criteria are invalid.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
: Timeout has occurred.

FmcjPersistentListSetThreshold()

This function/subprogram sets the threshold of the persistent list to the specified value (action call).

If no threshold is provided (NULL pointer), the threshold of the persistent list is erased. This means that all objects contained in the list will be provided when queried.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

One of:
- Be the owner of the list
- Staff definition
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjPersistentListSetThreshold(
       FmcjPersistentListHandle  hdlList,
       unsigned long const *     threshold )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjPLSetThreshold.

        FmcjPLSetThreshold.

           CALL    "FmcjPersistentListSetThreshold"
                          USING
                          BY VALUE
                             hdlList
                             threshold
                          RETURNING
                             intReturnValue.
```

Parameters

hdlList
Input. The handle of the persistent list object whose threshold is to be set.

threshold
Input. A pointer to the threshold to be set; can be a NULL pointer.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The persistent list does no longer exist.

FMC_ERROR_INVALID_THRESHOLD(807)
The threshold is invalid.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Chapter 38. FmcjPerson functions/subprograms

A Person object represents an MQSeries Workflow user. A person is uniquely identified by its object identifier or by its user identification.

FmcjPersonRefresh()

This function/subprogram refreshes the person from the MQSeries Workflow execution server (action call).

All information about the person is retrieved.

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjPersonRefresh( FmcjPersonHandle hdlPerson )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjPRefresh.

        FmcjPRefresh.

           CALL    "FmcjPersonRefresh"
                          USING
                          BY VALUE
                             hdlPerson
                          RETURNING
                             intReturnValue.
```

Parameters

hdlPerson
Input. The handle of the person to be refreshed.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is incorrect; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjPersonSetAbsence()

This function/subprogram sets the absence indication of the logged-on user to the specified value (action call).

When a person is absent, this person does not participate in staff resolution, that is, this person does not get assigned any work items.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjPersonSetAbsence(
       FmcjPersonHandle hdlPerson
       bool             new value )
```

Cobol language signature

Cobol language signature

```
            PERFORM FmcjPSetAbsence.

         FmcjPSetAbsence.

            CALL    "FmcjPersonSetAbsence"
                           USING
                           BY VALUE
                              hdlPerson
                              newValue
                           RETURNING
                              intReturnValue.
```

Parameters

hdlPerson
: Input. The handle of the person object whose absence is to be set.

newValue
: Input. True, if the person is denoted as absent, else false.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is incorrect; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
: Timeout has occurred.

FmcjPersonSetSubstitute()

This function/subprogram sets the substitute of the logged-on user (action call).

The substitute must be a registered MQSeries Workflow user ID other than the logged-on user. If no substitute is provided (NULL pointer), the substitute of the logged-on user is erased.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjPersistentListSetSubstitute(
       FmcjPersonHandle hdlPerson
       char const *     substitute )
```

Cobol language signature

Cobol language signature

```
            PERFORM FmcjPSetSubstitute.

         FmcjPSetSubstitute.

            CALL    "FmcjPersonSetSubstitute"
                           USING
                           BY VALUE
                              hdlPerson
                              substitute
                           RETURNING
                              intReturnValue.
```

Parameters

hdlPerson
: Input. The handle of the person object whose substitute is to be set.

substitute
: Input. A pointer to the substitute to be set; can be a null pointer.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is incorrect; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_INVALID_USER(132)
The user ID specified for the owner of the list does not conform to the syntax rules.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_USERID_UNKNOWN(10)
No user ID registered with MQSeries Workflow has been provided.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Chapter 39. FmcjProcessInstance functions/subprograms

An FmcjProcessInstance object represents an instance of a process template. A process instance is uniquely identified by its object identifier or by its name. Depending on the keep option when the process instance was created, the unique process instance name has been supplied by the user or has been generated by MQSeries Workflow.

The following diagram provides an overview on the possible process instance states and the actions which are allowed in those states, provided that the appropriate authority has been granted:

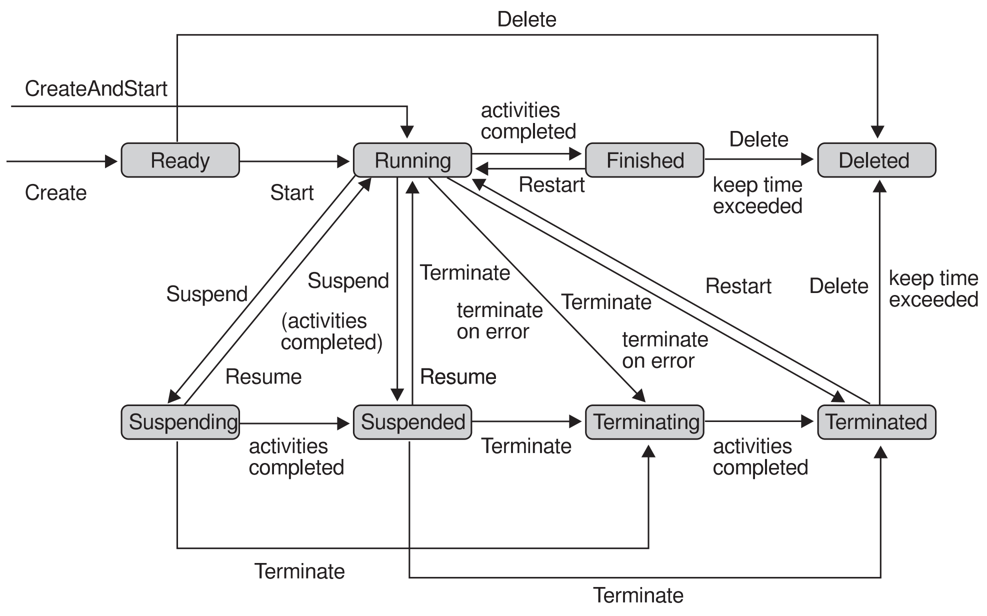


Figure 21. Process instance states

FmcjProcessInstanceDelete()

This function/subprogram deletes the specified process instance from the MQSeries Workflow execution server (action call).

The process instance must be a top level process and in states *Ready*, *Finished*, or *Terminated*. The creator can delete the process instance as long as it has not been started.

There are no impacts on your transient representation of the process instance; you have to destruct or deallocate the transient object when it is no longer needed.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

One of:
- Process administration authorization
- Be the process instance creator
- Be the process administrator
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjProcessInstanceDelete(
       FmcjProcessInstanceHandle  hdlInstance )
```

Cobol language signature

Cobol language signature

```
            PERFORM FmcjPIDelete.

         FmcjPIDelete.

            CALL    "FmcjProcessInstanceDelete"
                           USING
                           BY VALUE
                              hdlInstance
                           RETURNING
                              intReturnValue.
```

Parameters

hdlInstance
Input. The handle of the process instance to be deleted.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The process instance does no longer exist.

FMC_ERROR_NOT_AUTHORIZED(119)
Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_WRONG_STATE(120)
The process instance is in the wrong state.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjProcessInstanceInContainer()

This function/subprogram retrieves the input container associated with the process instance from the MQSeries Workflow execution server (action call).

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

One of:

- Process authorization
- Process administration authorization
- Be the process instance creator
- Be the process administrator
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjProcessInstanceInContainer(
       FmcjProcessInstanceHandle       hdlInstance,
       FmcjReadWriteContainerHandle *  input )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjPIInCtnr.

        FmcjPIInCtnr.

           CALL    "FmcjProcessInstanceInContainer"
                          USING
                          BY VALUE
                             hdlInstance
                          BY REFERENCE
                             inputValue
                          RETURNING
                             intReturnValue.
```

Parameters

hdlInstance
: Input. The handle of the process instance object whose input container is to be retrieved.

input
: Input/Output. The address of the input container handle respectively the input container of the process instance to be set.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The process instance does no longer exist.

FMC_ERROR_NOT_AUTHORIZED(119)
: Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjProcessInstanceObtainMonitor()

This function/subprogram obtains a monitor for the process instance from the MQSeries Workflow execution server (action call).

When the deep option is specified, then activity instances of type Block are resolved, that is, their block instance monitors are also fetched from the server.

Note: Deep is currently not supported.

The application is completely responsible for the ownership of objects, that is, it is not checked whether the process instance monitor handle already points to some object when a new one is assigned.

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

One of:
- Process authorization
- Process administration authorization
- Be the process instance creator
- Be the process administrator
- Be the system administrator

Required connection

MQSeries Workflow execution server (present session mode)

API include file

fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjProcessInstanceObtainMonitor(
       FmcjProcessInstanceHandle          hdlInstance,
       bool                               deep,
       FmcjProcessInstanceMonitorHandle * monitor )
```

Cobol language signature

Cobol language signature

```
PERFORM FmcjPIObtainMon.

FmcjPIObtainMon.

   CALL    "FmcjProcessInstanceObtainMonitor"
                USING
                BY VALUE
                   hdlInstance
                   deep
                BY REFERENCE
                   monitor
                RETURNING
                   intReturnValue.
```

Parameters

hdlInstance
: Input. The handle of the process instance object whose monitor is to be retrieved.

deep
: Input. An indicator whether activity instances of type Block are to be resolved, that is, their monitor is also to be provided. Note, deep is currently ignored.

monitor
: Input/Output. The address of the monitor handle respectively the monitor of the process instance to be set.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_EMPTY(122)
: The object has not yet been read from the database, that is, does not yet represent a persistent one.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is incorrect; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The process instance does no longer exist.

FMC_ERROR_NOT_AUTHORIZED(119)
: Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; the server to which the connection should be established is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjProcessInstancePersistentObject()

This function/subprogram retrieves the process instance identified by the passed object identifier from the MQSeries Workflow execution server (action call).

The MQSeries Workflow execution server from which the process instance is to be retrieved is identified by the service object or handle. The process instance handle to be initialized must be a null pointer respectively the process instance object to be initialized must be empty. The transient object is then updated with all information, primary and secondary, of the process instance.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signatures

C language signature

```
APIRET FMC_APIENTRY FmcjProcessInstancePersistentObject(
       FmcjExecutionServiceHandle  service,
       char const *                oid,
       FmcjProcessInstanceHandle * hdlInstance )
```

Cobol language signatures

Cobol language signature

```
        PERFORM FmcjPIPersistentObj.

     FmcjPIPersistentObj.

        CALL    "FmcjProcessInstancePersistentObject"
                       USING
                       BY VALUE
                          serviceValue
                          oid
                       BY REFERENCE
                          hdlInstance
                       RETURNING
                          intReturnValue.
```

Parameters

service
: Input. The service object representing the session with the execution server.

oid
: Input. The object identifier of the process instance to be retrieved.

hdlInstance
: Input/Output. The address of the handle to the process instance object to be set.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The process instance does no longer exist.

FMC_ERROR_INVALID_OID(805)
: The provided oid is invalid.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
: Timeout has occurred.

FmcjProcessInstanceRefresh()

This function/subprogram refreshes the process instance from the MQSeries Workflow execution server (action call).

All information about the process instance, primary and secondary, is retrieved.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjProcessInstanceRefresh(
       FmcjProcessInstanceHandle  hdlInstance )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjPIRefresh.

        FmcjPIRefresh.

           CALL    "FmcjProcessInstanceRefresh"
                          USING
                          BY VALUE
                             hdlInstance
                          RETURNING
                             intReturnValue.
```

Parameters

hdlInstance
Input. The handle of the process instance object to be refreshed.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The process instance does no longer exist.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjProcessInstanceResume()

This function/subprogram resumes processing of a suspended or suspending process instance (client-server call).

All non-autonomous subprocesses are also resumed, if the deep option equals true.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

One of:

- Process administration authorization
- Be the process administrator
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjProcessInstanceResume(
       FmcjProcessInstanceHandle  hdlInstance,
       bool                       deep )
```

Cobol language signature

Cobol language signature

```
            PERFORM FmcjPIResume.

         FmcjPIResume.

            CALL    "FmcjProcessInstanceResume"
                           USING
                           BY VALUE
                              hdlInstance
                              deep
                           RETURNING
                              intReturnValue.
```

Parameters

hdlInstance
Input. The handle of the process instance to be restarted.

deep
Input. If deep is true, processing of all non-autonomous subprocesses is also resumed.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The process instance does no longer exist.

FMC_ERROR_NOT_AUTHORIZED(119)
Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_WRONG_STATE(120)
The process instance is in the wrong state.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjProcessInstanceSetDescription()

This function/subprogram sets the description of the process instance to the specified value (action call).

If no description is provided (NULL pointer), the description of the process instance is erased.

The following rules apply for specifying a process instance description:
- You can specify a maximum of 254 characters.
- You can use any printable characters depending on your current locale, including the end-of-line and new-line characters.

Properties

Usage notes
- See “Action functions/subprograms” on page 76 for general information.

Authorization

One of:
- Process authorization
- Process administration authorization
- Be the process instance creator
- Be the process administrator
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjProcessInstanceSetDescription(
       FmcjProcessInstanceHandle  hdlInstance,
       char const *               description )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjPISetDescription.

        FmcjPISetDescription.

           CALL    "FmcjProcessInstanceSetDescription"
                          USING
                          BY VALUE
                             hdlInstance
                             description
                          RETURNING
                             intReturnValue.
```

Parameters

hdlInstance
: Input. The handle of the process instance object whose description is to be set.

description
: Input. A pointer to the description to be set; can be a NULL pointer.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The process instance does no longer exist.

FMC_ERROR_INVALID_DESCRIPTION(810)
: The description does not conform to the syntax rules.

FMC_ERROR_NOT_AUTHORIZED(119)
: Not authorized.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
: Timeout has occurred.

FmcjProcessInstanceSetName()

This function/subprogram sets the name of the process instance to the specified value (client-server call).

The process instance must still be in the *Ready* state.

The following rules apply for specifying a process instance name:
- You can specify a maximum of 63 characters.
- You can use any printable characters depending on your current locale, except the following:
  `* ? " ; : .`
- You can use blanks with these restrictions: no leading blanks, no trailing blanks, and no consecutive blanks.

Properties

Usage notes
- See "Action functions/subprograms" on page 76 for general information.

Authorization

One of:
- Process authorization
- Process administration authorization
- Be the process instance creator
- Be the process administrator
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjProcessInstanceSetName(
       FmcjProcessInstanceHandle  hdlInstance,
       char const *               name )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjPISetName.

        FmcjPISetName.

           CALL    "FmcjProcessInstanceSetName"
                          USING
                          BY VALUE
                             hdlInstance
                             name
                          RETURNING
                             intReturnValue.
```

Parameters

hdlInstance
Input. The handle of the process instance object whose name is to be set.

name Input. The name to be set.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The process instance does no longer exist.

FMC_ERROR_INVALID_NAME(134)
The name does not conform to the syntax rules.

FMC_ERROR_NOT_AUTHORIZED(119)
Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_NOT_UNIQUE(121)
The process instance name is not unique.

FMC_ERROR_WRONG_STATE(120)
The process instance is in the wrong state.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjProcessInstanceStart()

This function/subprogram starts a ready process instance (action call).

When successfully executed, the starter is set to the requestor of this action and the process administrator is determined.

When initial values are to be passed to the process instance to be started, an input container can be provided - see also FmcjProcessInstance InContainer(). When the process instance requires input and is started without specifying an input container, the input-container values are not set. So, when, for example, input-container values are queried from within an activity implementation, FMC_ERROR_MEMBER_NOT_SET is returned.

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

One of:

- Process authorization
- Process administration authorization
- Be the process instance creator
- Be the process administrator
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjProcessInstanceStart(
       FmcjProcessInstanceHandle     hdlInstance,
       FmcjReadWriteContainerHandle  input )
```

Cobol language signature

Cobol language signatures

```
           PERFORM FmcjPIStart.

        FmcjPIStart.

           CALL    "FmcjProcessInstanceStart"
                          USING
                          BY VALUE
                             hdlInstance
                             inputValue
                          RETURNING
                             intReturnValue.
```

Parameters

hdlInstance
Input. The handle of the process instance object to be started.

input Input. The input container of the process instance.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The process instance does no longer exist.

FMC_ERROR_NOT_AUTHORIZED(119)
Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_WRONG_STATE(120)
The process instance is in the wrong state.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjProcessInstanceSuspend()

This function/subprogram suspends (temporarily stops) the process instance (action call).

The process instance must be in state *Running*. All non-autonomous subprocesses are also suspended if the deep option equals true. Autonomous subprocesses are not considered.

The process instance remains in state *Suspending* as long as there are running program activity implementations or suspending non-autonomous subprocesses. When the activity implementations completed their executions and when the non-autonomous subprocesses reached the *Suspended* state, the process instance is put into the *Suspended* state.

Optionally, a date may be specified up to when the process instance is suspended; it is then automatically resumed, together with the non-autonomous subprocesses, if the deep option had been specified.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

One of:

- Process administration authorization
- Be the process administrator
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signatures

C language signatures

```
APIRET FMC_APIENTRY FmcjProcessInstanceSuspend(
       FmcjProcessInstanceHandle  hdlInstance,
       bool                       deep )

APIRET FMC_APIENTRY FmcjProcessInstanceSuspendUntil(
       FmcjProcessInstanceHandle  hdlInstance,
       FmcjCDateTime const *      time,
       bool                       deep )
```

Cobol language signatures

Cobol language signatures

```
           PERFORM FmcjPISuspend.

        FmcjPISuspend.

           CALL    "FmcjProcessInstanceSuspend"
                          USING
                          BY VALUE
                             hdlInstance
                             deep
                          RETURNING
                             intReturnValue.


           PERFORM FmcjPISuspendUntil.

        FmcjPISuspendUntil.

           CALL    "FmcjProcessInstanceSuspendUntil"
                          USING
                          BY VALUE
                             hdlInstance
                          BY REFERENCE
                             timeValue
                          BY VALUE
                             deep
                          RETURNING
                             intReturnValue.
```

Parameters

hdlInstance
Input. The handle of the process instance object to be started.

time Input. The date/time respectively a pointer to the date/time up to when the process instance is to be suspended.

deep Input. An indicator whether also non-autonomous subprocesses are to be suspended.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The process instance does no longer exist.

FMC_ERROR_NOT_AUTHORIZED(119)
Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_WRONG_STATE(120)
The process instance is in the wrong state.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjProcessInstanceTerminate()

This function/subprogram terminates a top level process instance or a remote subprocess and all of its non-autonomous subprocesses (action call).

The process instance must be in states *Running*, *Suspended*, or *Suspending*.

The process instance is put into state terminating as long as there are running activity implementations or terminating non-autonomous subprocesses. When the activity implementations completed their execution or when the non-autonomous subprocesses terminated, the process instance is put into the *Terminated* state. When the process instance has reached the *Terminated* state, it is deleted depending on the "delete finished items" option.

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

One of:

- Process administration authorization
- Be the process administrator
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjProcessInstanceTerminate(
       FmcjProcessInstanceHandle  hdlInstance )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjPITerminate.

        FmcjPITerminate.

           CALL    "FmcjProcessInstanceTerminate"
                          USING
                          BY VALUE
                             hdlInstance
                          RETURNING
                             intReturnValue.
```

Parameters

hdlInstance
: Input. The handle of the process instance object to be terminated.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The process instance does no longer exist.

FMC_ERROR_NOT_AUTHORIZED(119)
: Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_WRONG_STATE(120)
: The process instance is in the wrong state.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Chapter 40. FmcjProcessInstanceList functions/subprograms

A process instance list represents a set of process instances. All process instances which are accessible through this list have the same characteristics. These characteristics are specified by a filter. Additionally, sort criteria can be applied and, after that, a threshold to restrict the number of process instances to be transferred from the execution server to the client.

The process instance list definition is stored persistently.

In the C++ implementation, FmcjProcessInstanceList is a subclass of FmcjPersistentList and thus inherits all its methods. Similarly, in the C and Cobol language API, all functions of FmcjPersistentList are also applicable to FmcjProcessInstanceList.

FmcjProcessInstanceListQueryProcessInstances()

This function/subprogram retrieves the primary information for all process instances characterized by the specified process instance list from the MQSeries Workflow execution server (action call).

From the set of qualifying process instances, only those are retrieved the user is authorized for. The user is authorized for a process instance if the process instance:
- Does not belong to any category
- Does belong to a category and the user has global process authorization or global process administration authorization or selected process authorization or selected process administration authorization for that category

The primary information that is retrieved for each process instance is:
- Category
- Description
- Icon
- InContainerNeeded
- LastModificationTime
- LastStateChangeTime
- Name
- ParentName
- ProcessTemplateName
- StartTime
- State
- SuspensionExpirationTime
- SuspensionTime
- SystemName
- SystemGroupName
- TopLevelName

Any process instances retrieved are appended to the supplied vector of process instances. If you want to read those process instances only which are currently included in the process instance list, you have to clear the vector before you call this function/subprogram.

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjProcessInstanceListQueryProcessInstances(
        FmcjProcessInstanceListHandle     hdlList,
        FmcjProcessInstanceVectorHandle * instances )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjPILQueryProcInsts.

        FmcjPILQueryProcInsts.

           CALL    "FmcjProcessInstanceListQueryProcessInstances"
                          USING
                          BY VALUE
                             hdlList
                          BY REFERENCE
                             instances
                          RETURNING
                             intReturnValue.
```

Parameters

hdlList
: Input. The handle of the process instance list to be queried.

instances
: Input/Output. The vector of qualifying process instances.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The process instance list does no longer exist.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
: Timeout has occurred.

Examples

- For a C language example see "Query worklists (C language)" on page 447
- For a Cobol example see "Query worklists (Cobol language)" on page 448

Chapter 41. FmcjProcessInstanceNotification functions/subprograms

An FmcjProcessInstanceNotification object represents a notification on a process instance assigned to a user.

Other items assigned to users are activity instance notifications and work items. FmcjItem represents the common properties of such items. In the C++ implementation, FmcjProcessInstanceNotification is thus a subclass of the FmcjItem class. Similarly, in the C and Cobol language, it takes common implementations of functions from FmcjItem.

A process instance notification is uniquely identified by its object identifier.

FmcjProcessInstanceNotificationPersistentObject()

This function/subprogram retrieves the process instance notification identified by the passed object identifier from the MQSeries Workflow execution server (action call).

The MQSeries Workflow execution server from which the process instance notification is to be retrieved is identified by the service object. The process instance notification handle to be initialized must be a null pointer respectively the process instance notification object to be initialized must be empty. The transient object is then updated with all information - primary and secondary - of the activity instance notification.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signatures

C language signature

```
APIRET FMC_APIENTRY FmcjProcessInstanceNotificationPersistentObject(
       FmcjExecutionServiceHandle               service,
       char const *                             oid,
       FmcjProcessInstanceNotificationHandle *  hdlItem )
```

Cobol language signatures

Cobol language signature

```
            PERFORM FmcjPINPersistentObj.

         FmcjPINPersistentObj.

            CALL    "FmcjProcessInstanceNotificationPersistentObject"
                           USING
                           BY VALUE
                              serviceValue
                              oid
                           BY REFERENCE
                              hdlItem
                           RETURNING
                              intReturnValue.
```

Parameters

service
: Input. The service object representing the session with the execution server.

oid
: Input. The object identifier of the process instance notification to be retrieved.

hdlItem
: Input/Output. The address of the handle to the process instance notification object to be set.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The process instance notification does no longer exist.

FMC_ERROR_INVALID_OID(805)
: The provided oid is invalid.

FMC_ERROR_NOT_AUTHORIZED(119)
: Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Chapter 42. FmcjProcessTemplate functions/subprograms

An FmcjProcessTemplate object is the frozen state of a process model from which it is created via translation. All program definitions and data structures referenced by the process model are copied into the process template (early binding). Subprocesses are lately bound. Their definitions are only located during execution.

A process template is uniquely identified by its object identifier or by its name and a valid-from date. This *valid-from date* determines since when the process template can be used to create process instances.

When process templates are queried from the execution server, then only valid process templates are returned.

FmcjProcessTemplateCreateAndStartInstance()

This function/subprogram creates a process instance from the specified process template and starts the resulting process instance (action call).

Depending on the keepName option, a process instance name must be provided. If the process instance name is to be kept *as is*, you cannot provide an empty string.

The following rules apply for specifying a process instance name:
- You can specify a maximum of 63 characters.
- You can use any printable characters depending on your current locale, except the following:
  `* ? " ; : .`
- You can use blanks with these restrictions: no leading blanks, no trailing blanks, and no consecutive blanks.

If a unique name may be generated by MQSeries Workflow, the following applies:
- If no or an empty process instance name is provided, an instance is created with a default name *ProcessTemplateNameOid*, where *Oid* is a 32 byte printable version of the process instance object identifier. Since the process instance name cannot become longer than 63 characters, any process template name longer than 31 characters is shortened.
- If a process instance name is provided, that name is kept as long as it is unique. If the provided process instance name is already used for another instance, an instance is created with the name *nameOid*, where *Oid* is a 32 byte printable version of the process instance object identifier. Since the process instance name cannot become longer than 63 characters, any name longer than 31 characters is shortened.

The passed name parameter value remains unchanged; FmcjProcessInstanceName() returns the actual name of the process instance created.

When initial values are to be passed to the process instance to be created and started, an input container can be provided - see also FmcjProcessTemplateInContainer(). When a process instance that requires input is started without specifying an input container, the input-container values are not set. When, for example, input-container values are queried from within an activity implementation, FMC_ERROR_MEMBER_NOT_SET is returned.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

One of:

- Process authorization
- Process administration authorization
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjProcessTemplateCreateAndStartInstance(
       FmcjProcessTemplateHandle     hdlTemplate,
       char const *                  name,
       char const *                  reserved1,
       char const *                  reserved2,
       FmcjReadWriteContainerHandle  input,
       bool                          keepName,
       FmcjProcessInstanceHandle *   newInstance )
```

Cobol language signature

Cobol language signatures

```
           PERFORM FmcjPTCreateAndStartInst.

        FmcjPTCreateAndStartInst.

           CALL    "FmcjProcessTemplateCreateAndStartInstance"
                          USING
                          BY VALUE
                             hdlTemplate
                             name
                             reserved1
                             reserved2
                             inputValue
                             keepName
                          BY REFERENCE
                             newInstance
                          RETURNING
                             intReturnValue.
```

Parameters

hdlTemplate
Input. The handle of the process template object to be used.

name Input. The name of the process instance to be created and started.

input Input. The input container of the process instance.

keepName
Input. True, if only the specified name can be used for the process instance. False, if a unique name can be generated.

newInstance
Input/Output. The newly created and started process instance.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The process template does no longer exist or is no longer valid.

FMC_ERROR_INVALID_NAME(134)
The specified process instance name does not comply with the syntax rules.

FMC_ERROR_NOT_AUTHORIZED(119)
Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_NOT_UNIQUE(121)
The name of the process instance is not unique; can also happen when a process instance with the same name is not yet physically deleted.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjProcessTemplateCreateAndSuspendInstance()

This function/subprogram creates a process instance from the specified process template and suspends the resulting process instance (action call). Depending on the keepName option, a process instance name should be provided. If the provided process instance name is to be kept *as is*, you cannot provide an empty string. Optionally, a system group where to create and suspend the process instance may be provided. If a system group is specified, then also a system at that system group must be specified to designate the system - actually the execution server - which should perform the action. If no system and system group name is provided, then the user's session determines the location of the process instance; it is created on the logged on user's home system group by the connected execution server.

The following rules apply for specifying a process instance name:

- You can specify a maximum of 63 characters.
- You can use any printable characters depending on your current locale, except the following:
  `* ? " ; : .`
- You can use blanks with these restrictions: no leading blanks, no trailing blanks, and no consecutive blanks.

If a unique name may be generated by MQSeries Workflow, the following applies:

- If no or an empty process instance name is provided, an instance is created with a default name *ProcessTemplateNameOid*, where *Oid* is a 32 byte printable version of the process instance object identifier. Since the process instance name cannot become longer than 63 characters, any process template name longer than 31 characters is shortened.
- If a process instance name is provided, that name is kept as long as it is unique. If the provided process instance name is already used for another instance, an instance is created with the name *nameOid*, where *Oid* is a 32 byte printable version of the process instance object identifier. Since the process instance name cannot become longer than 63 characters, any name longer than 31 characters is shortened.

The passed name remains unchanged; FmcjProcessInstanceName() returns the actual name of the process instance created.

When initial values are to be passed to the process instance to be created and suspended, an input container may be provided - see also FmcjProcessTemplateInContainer(). When a process instance that requires input is started/ suspended without specifying an input container, the input-container values are not set. So, when, for example, input-container values are queried from within an activity implementation, FMC_ERROR_MEMBER_NOT_SET is returned.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

One of:

- Process administration authorization
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjProcessTemplateCreateAndSuspendInstance(
       FmcjProcessTemplateHandle     hdlTemplate,
       char const *                  name,
       char const *                  systemGroup,
       char const *                  system,
       FmcjReadWriteContainerHandle  input,
       bool                          keepName,
       FmcjProcessInstanceHandle *   newInstance )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjPTCreateAndSuspendInst.

        FmcjPTCreateAndSuspendInst.

           CALL    "FmcjProcessTemplateCreateAndSuspendInstance"
                          USING
                          BY VALUE
                             hdlTemplate
                          BY REFERENCE
                             name
                             systemGroup
                             system
                          BY VALUE
                             inputValue
                             keepName
                          BY REFERENCE
                             newInstance
                          RETURNING
                             intReturnValue.
```

Parameters

hdlTemplate
Input. The handle of the process template object to be used.

name
Input. The name of the process instance to be created and suspended.

systemGroup
Input. The name of the system group where the process instance is to be created and suspended.

system
Input. The name of the system at the system group which should create and suspend the process instance.

input
Input. The input container of the process instance.

keepName
Input. True if only the specified name may be used for the process instance. False if a unique name may be generated.

newInstance
Input/Output. The newly created and suspended process instance.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_NOT_AUTHORIZED(119)
Not authorized to use the function/subprogram.

FMC_ERROR_DOES_NOT_EXIST(118)
The process template does no longer exist or is no longer valid.

FMC_ERROR_INVALID_NAME(134)
The specified process instance name does not comply with the syntax rules.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_NOT_UNIQUE(121)
The name of the process instance is not unique.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Examples

- For a C language example see
- For a Cobol example see

FmcjProcessTemplateCreateInstance()

This function/subprogram creates a process instance from the specified process template (action call).

Depending on the keepName option, a process instance name must be provided. If the process instance name is to be kept *as is*, you cannot provide an empty string.

The following rules apply for specifying a process instance name:

- You can specify a maximum of 63 characters.
- You can use any printable characters depending on your current locale, except the following:
  `* ? " ; : .`
- You can use blanks with these restrictions: no leading blanks, no trailing blanks, and no consecutive blanks.

If a unique name may be generated by MQSeries Workflow, the following applies:

- If no name or an empty process instance name is provided, an instance is created with a default name *ProcessTemplateNameOid*, where *Oid* is a 32 byte printable version of the process instance object identifier. Since the process instance name cannot become longer than 63 characters, any process template name longer than 31 characters is shortened.
- If a process instance name is provided, that name is kept as long as it is unique. If the provided process instance name is already used for another instance, an instance is created with the name *nameOid*, where *Oid* is a 32 byte printable

version of the process instance object identifier. Since the process instance name cannot become longer than 63 characters, any name longer than 31 characters is shortened.

The passed name parameter value remains unchanged; FmcjProcessInstanceName() returns the actual name of the process instance created.

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

One of:

- Process authorization
- Process administration authorization
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjProcessTemplateCreateInstance(
       FmcjProcessTemplateHandle  hdlTemplate,
       char const *                name,
       char const *                reserved1,
       char const *                reserved2,
       bool                        keepName,
       FmcjProcessInstanceHandle *  newInstance )
```

Cobol language signature

Cobol language signature

```
       PERFORM FmcjPTCreateInst.

    FmcjPTCreateInst.

       CALL    "FmcjProcessTemplateCreateInstance"
                      USING
                      BY VALUE
                         hdlTemplate
                         name
                         reserved1
                         reserved2
                         keepName
                      BY REFERENCE
                         newInstance
                      RETURNING
                         intReturnValue.
```

Parameters

hdlTemplate
: Input. The handle of the process template object to be used.

name
: Input. The name of the process instance to be created.

keepName
: Input. True, if only the specified name can be used for the process instance. False, if a unique name can be generated.

newInstance
: Input/Output. The newly created process instance.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The process template does no longer exist or is no longer valid.

FMC_ERROR_INVALID_NAME(134)
: The specified process instance name does not comply with the syntax rules.

FMC_ERROR_NOT_AUTHORIZED(119)
: Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_NOT_UNIQUE(121)
: The name of the process instance is not unique; can also happen when a process instance with the same name is not yet physically deleted.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjProcessTemplateDelete()

This function/subprogram deletes the specified process template from the MQSeries Workflow modeling server (client-server call). Note that the transient representation is not impacted; you have to destruct/ deallocate the transient object when it is no longer needed.

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

process definition authorization

Required connection

MQSeries Workflow modeling server

API include file

BuildTime: fmcjcbld.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjProcessTemplateDelete(
      FmcjProcessTemplateHandle hdlTemplate )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjPTDelete.

        FmcjPTDelete.

           CALL    "FmcjProcessTemplateDelete"
                          USING
                          BY VALUE
                             hdlTemplate
                          RETURNING
                             intReturnValue.
```

Parameters

hdlTemplate
Input. The handle of the process template to be deleted.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_NOT_AUTHORIZED(119)
Not authorized to use the function/subprogram.

FMC_ERROR_DOES_NOT_EXIST(118)
The process template does no longer exist.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Examples

- For a C language example see
- For a Cobol example see

FmcjProcessTemplateInContainer()

This function/subprogram retrieves the input container associated with the process template from the MQSeries Workflow execution server (action call).

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

One of:

- Process authorization
- Process administration authorization
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjProcessTemplateInContainer(
       FmcjProcessTemplateHandle      hdlTemplate,
       FmcjReadWriteContainerHandle * input )
```

Cobol language signature

Cobol language signature

```
            PERFORM FmcjPTInCtnr.

         FmcjPTInCtnr.

            CALL    "FmcjProcessTemplateInContainer"
                           USING
                           BY VALUE
                              hdlTemplate
                           BY REFERENCE
                              inputValue
                           RETURNING
                              intReturnValue.
```

Parameters

hdlTemplate
Input. The handle of the process template object whose input container is to be retrieved.

input Input/Output. The address of the input container handle respectively the input container of the process template to be set.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The process template does no longer exist or is no longer valid.

FMC_ERROR_NOT_AUTHORIZED(119)
: Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
: Timeout has occurred.

FmcjProcessTemplatePersistentObject()

This function/subprogram retrieves the process template identified by the passed object identifier from the MQSeries Workflow execution server (action call).

The MQSeries Workflow execution server from which the process template is to be retrieved is identified by the service object or handle. The process template handle to be initialized must be a null pointer respectively the process template object to be initialized must be empty. The transient object is then updated with all information - primary and secondary - of the process template.

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signatures

C language signature

```
APIRET FMC_APIENTRY FmcjProcessTemplatePersistentObject(
       FmcjExecutionServiceHandle   service,
       char const *                 oid,
       FmcjProcessTemplateHandle *  hdlTemplate )
```

Cobol language signatures

Cobol language signature

```
            PERFORM FmcjPTPersistentObj.

         FmcjPTPersistentObj.

            CALL    "FmcjProcessTemplatePersistentObject"
                           USING
                           BY VALUE
                              serviceValue
                              oid
                           BY REFERENCE
                              hdlTemplate
                           RETURNING
                              intReturnValue.
```

Parameters

service
Input. The service object representing the session with the execution server.

oid Input. The object identifier of the process template to be retrieved.

hdlTemplate
Input/Output. The address of the handle to the process template object to be set.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The process template does no longer exist or is no longer valid.

FMC_ERROR_INVALID_OID(805)
The provided oid is invalid.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjProcessTemplateRefresh()

This function/subprogram refreshes the process template from the MQSeries Workflow execution server (action call).

All information about the process template - primary and secondary - is retrieved.

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution or modeling server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjProcessTemplateRefresh(
      FmcjProcessTemplateHandle  hdlTemplate )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjPTRefresh.

        FmcjPTRefresh.

           CALL    "FmcjProcessTemplateRefresh"
                          USING
                          BY VALUE
                             hdlTemplate
                          RETURNING
                             intReturnValue.
```

Parameters

hdlTemplate
Input. The handle of the process template object to be refreshed.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The process template does no longer exist or is no longer valid.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Chapter 43. FmcjProcessTemplateList functions/subprograms

A process template list represents a set of process templates. All process templates which are accessible through this list have the same characteristics. These characteristics are specified by a filter. Additionally, sort criteria can be applied and, after that, a threshold to restrict the number of process templates to be transferred from the execution server to the client.

The process template list definition is stored persistently.

In the C++ implementation, FmcjProcessTemplateList is a subclass of FmcjPersistentList and thus inherits all its methods. Similarly, in the C and Cobol language API, all functions of FmcjPersistentList are also applicable to FmcjProcessTemplateList.

FmcjProcessTemplateListQueryProcessTemplates()

This function/subprogram retrieves the primary information for all process templates characterized by the specified process template list from the MQSeries Workflow execution server (action call).

From the set of qualifying process templates, only those are retrieved, the user is authorized for. The user is authorized for a process template if the process template:

- Does not belong to any category
- Does belong to a category and the user has global process authorization or global process administration authorization or selected process authorization or selected process administration authorization for that category

The primary information that is retrieved for each process template is:

- Category
- CreationTime
- Description
- Icon
- InContainerNeeded
- LastModificationTime
- Name
- ValidFromTime

Any process templates retrieved are appended to the supplied vector of process templates. If you want to read those process templates only which are currently included in the process template list, you have to clear the vector before you call this function/subprogram.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjProcessTemplateListQueryProcessTemplates(
       FmcjProcessTemplateListHandle      hdlList,
       FmcjProcessTemplateVectorHandle *  templates )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjPTLQueryProcTempls.

        FmcjPTLQueryProcTempls.

           CALL    "FmcjProcessTemplateListQueryProcessTemplates"
                          USING
                          BY VALUE
                             hdlList
                          BY REFERENCE
                             templates
                          RETURNING
                             intReturnValue.
```

Parameters

hdlList
: Input. The handle of the process template list to be queried.

templates
: Input/Output. The vector of qualifying process templates.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The process template list does no longer exist.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Examples

- For a C language example see "Query worklists (C language)" on page 447
- For a Cobol example see "Query worklists (Cobol language)" on page 448

Chapter 44. FmcjProcessTemplateVector functions/subprograms

A process template vector represents the result of a query for process templates in the C and Cobol language API. It provides a subset of the C++ vector functionality.

FmcjProcessTemplateVectorDeallocate()

This function deallocates the storage reserved for the specified transient vector of process templates. The set of transient process template representations are considered elements of the vector so that they are also deallocated. The vector handle is set to 0 so that it can no longer be used.

Properties

Usage notes

- See "Basic functions/subprograms" on page 53 for general information.

Authorization

None

Required connection

None

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signatures

C language signature

```
APIRET FMC_APIENTRY FmcjProcessTemplateVectorDeallocate(
       FmcjProcessTemplateVectorHandle *hdlVector )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjPTLVDeallocate.

       FmcjPTLVDeallocate.

           CALL    "FmcjProcessTemplateListVectorDeallocate"
                          USING
                          BY REFERENCE
                             hdlVector
                          RETURNING
                             intReturnValue.
```

Parameters

hdlVector
Input/Output. The address of the handle to the process template vector object to be deallocated.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A handle or pointer references an undefined location.

FMC_ERROR_INVALID_HANDLE(130)
The handle of the object to be deallocated is invalid; either 0 or no process template vector handle.

FmcjProcessTemplateVectorFirstElement()

This function returns the first element of the process template vector. That element has to be considered as an object on its own, that means you must deallocate the object if it is no longer needed.

The vector position is advanced to the next element in the process template vector.

A null handle is returned if an error occurred, that means, there is no element in the vector but you asked for it.

Properties

Usage notes

- See “Accessor function/subprograms” on page 62 for general information.

Authorization

None

Required connection

None

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
FmcjProcessTemplateHandle
FMC_APIENTRY FmcjProcessTemplateVectorFirstElement(
             FmcjProcessTemplateVectorHandle  hdlVector )
```

Cobol language signature

Cobol language signature

```
          PERFORM FmcjPTLVFirstElement.

       FmcjPTLVFirstElement.

          CALL    "FmcjProcessTemplateListVectorFirstElement"
                         USING
                         BY VALUE
                            hdlVector
                         RETURNING
                            FmcjPTLHandleReturnValue.
```

Parameters

hdlVector
Input. The handle of the process template vector object to be queried.

Return type

FmcjProcessTemplateHandle
The handle of the first process template in the vector.

FmcjProcessTemplateVectorNextElement()

This function returns the next element of the process template vector from the current position on. That element has to be considered as an object on its own, that means you must deallocate the object if it is no longer needed. The vector position is advanced to the next element.

A null handle is returned if an error occurred, that means, there is no more element in the vector but you asked for it.

Properties

Usage notes

- See "Accessor function/subprograms" on page 62 for general information.

Authorization

None

Required connection

None

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
FmcjProcessTemplateHandle
FMC_APIENTRY FmcjProcessTemplateVectorNextElement(
             FmcjProcessTemplateVectorHandle hdlVector )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjPTLVNextElement.

        FmcjPTLVNextElement.

           CALL    "FmcjProcessTemplateListVectorNextElement"
                          USING
                          BY VALUE
                             hdlVector
                          RETURNING
                             FmcjPTLHandleReturnValue.
```

Parameters

hdlVector
: Input. The handle of the process template vector object to be queried.

Return type

FmcjProcessTemplateHandle
: The handle of the next process template in the vector.

FmcjProcessTemplateVectorSize()

This function returns the number of elements in the process template vector.

Properties

Usage notes

- See "Accessor function/subprograms" on page 62 for general information.

Authorization

None

Required connection

None

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
unsigned long FMC_APIENTRY FmcjProcessTemplateVectorSize(
              FmcjProcessTemplateVectorHandle hdlVector )
```

Cobol language signature

Cobol language signature

```
            PERFORM FmcjPTLVSize.

         FmcjPTLVSize.

            CALL    "FmcjProcessTemplateListVectorSize"
                           USING
                           BY VALUE
                              hdlVector
                           RETURNING
                              ulongReturnValue.
```

Parameters

hdlVector
Input. The handle of the process template vector object to be queried.

Return type

unsigned long
The number of elements in the process template vector.

Chapter 45. FmcjService related functions/subprograms

An FmcjService object represents the common aspects of MQSeries Workflow service objects. In the C++ implementation, it is thus the superclass of the FmcjExecutionService Similarly, in the C and Cobol language, it provides for common implementations of functions.

FmcjServiceSetPassword()

This function/subprogram allows a user's password to be changed (action call).

Note: The password is case-sensitive.

The following rules apply for specifying a password:
- You can specify a maximum of 32 characters.
- You can use any printable characters depending on your current locale.

None

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

Logon required

Required connection

MQSeries Workflow execution server

API include file

fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_FMC_APIENTRY FmcjServiceSetPassword(
       FmcjServiceHandle      service,
       char const *           newPassword )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjSrvSetPassword.

        FmcjSrvSetPassword.

           CALL    "FmcjServiceSetPassword"
                          USING
                          BY VALUE
                             serviceValue
                             newPassword
                          RETURNING
                             intReturnValue.
```

Parameters

service
: Input. A handle to the service object representing the session with an MQSeries Workflow server.

newPassword
: Input. The new password to be used.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_USERID_UNKNOWN(10)
: The user does no longer exist.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_PASSWORD(12)
: The password does not comply with the MQSeries Workflow syntax rules.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
: Timeout has occurred.

FmcjServiceUserSettings()

This function/subprogram returns the settings of the logged on user (action call).

An empty object respectivly a null pointer is returned if no user has logged on yet via this service object.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signatures

C language signature

```
APIRET FMC_FMC_APIENTRY FmcjServiceUserSettings(
       FmcjServiceHandle   service,
       FmcjPersonHandle *  user )
```

Cobol language signatures

Cobol language signature

```
            PERFORM FmcjSrvUserSettings.

         FmcjSrvUserSettings.

            CALL    "FmcjServiceUserSettings"
                           USING
                           BY VALUE
                              serviceValue
                           BY REFERENCE
                              user
                           RETURNING
                              intReturnValue.
```

Parameters

service
: Input. A handle to the service object representing the session with an MQSeries Workflow server.

user Input/Output. The person object to contain respectively the address of the person handle to point to the settings of the logged on user.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Chapter 46. FmcjWorkitem functions/subprograms

An FmcjWorkitem object represents an activity instance assigned to a user in order to be worked on.

Other items assigned to users are notifications. FmcjItem represents the common properties of such items. In the C++ implementation, FmcjWorkitem is thus a subclass of the FmcjItem class. Similarly, in the C and Cobol language, it takes common implementations of functions from FmcjItem.

A work item is uniquely identified by its object identifier.

The following diagrams provide an overview on the possible work item states and the actions which are allowed in those states, provided that the appropriate authority has been granted. Note that the actions and possible states are dependent on the process instance state, the work item is a part of.

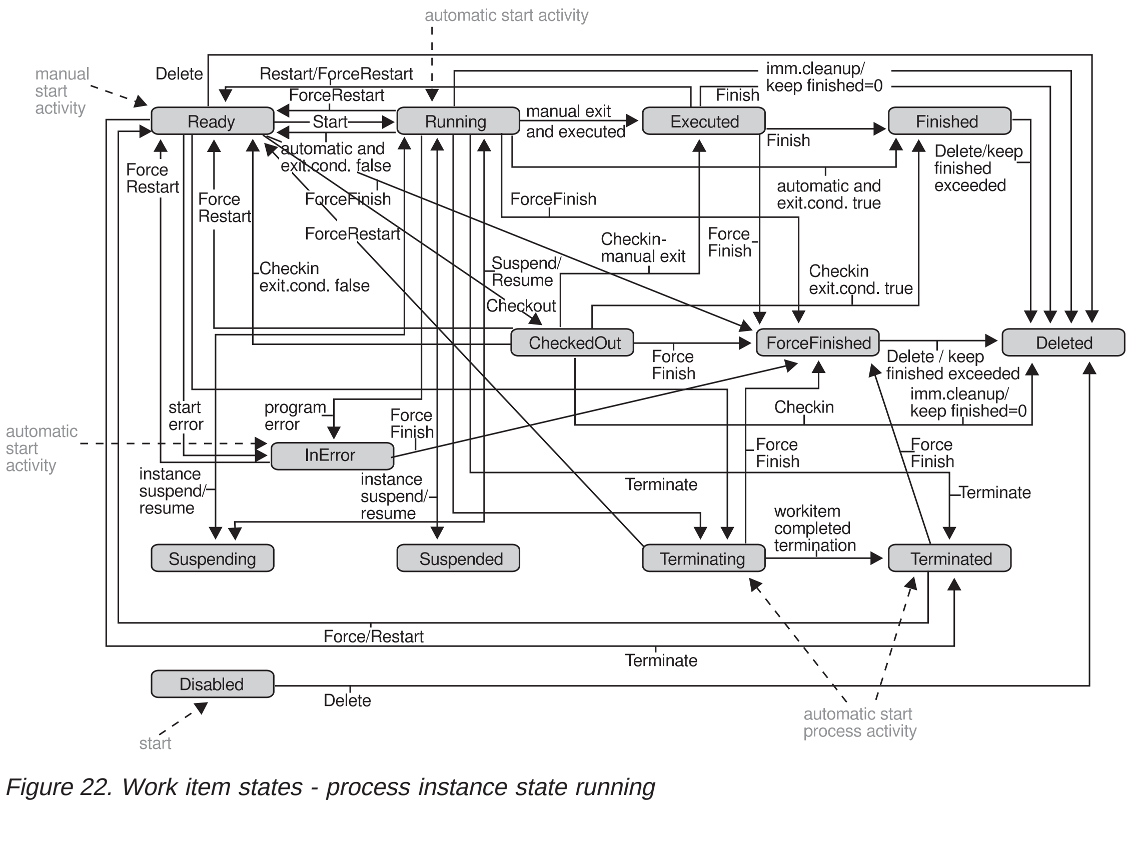


Figure 22. Work item states - process instance state running

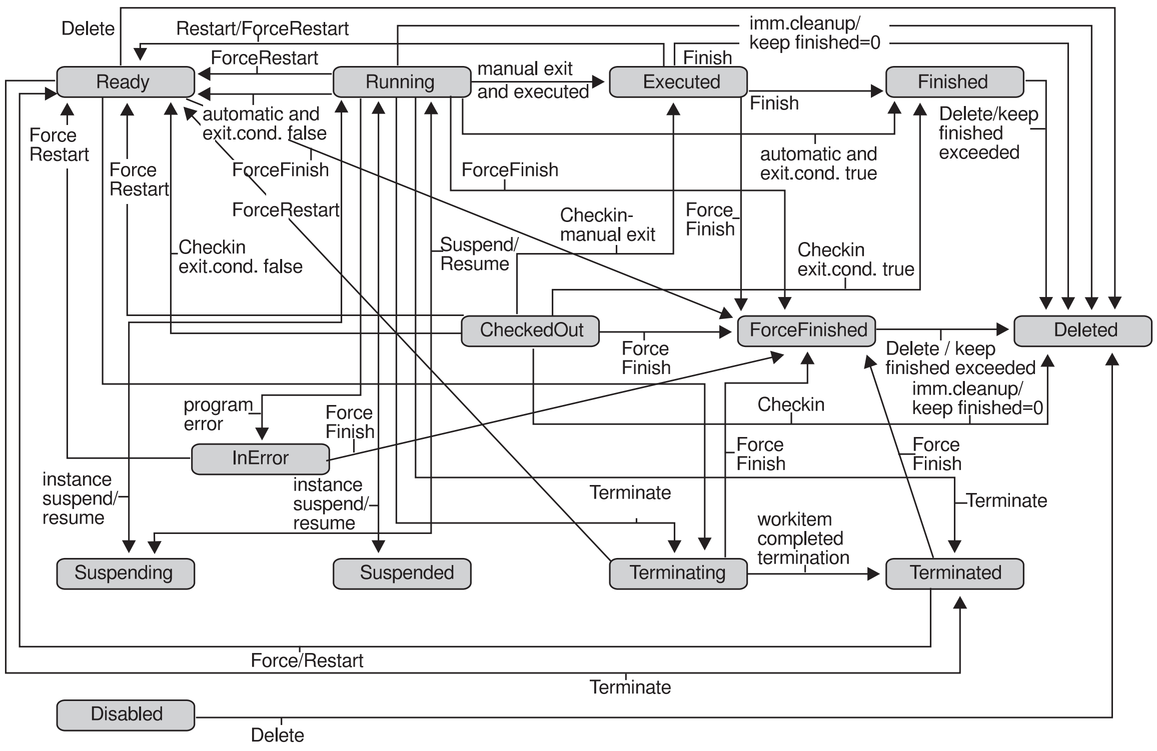


Figure 23. Work item states - process instance state suspending or suspended

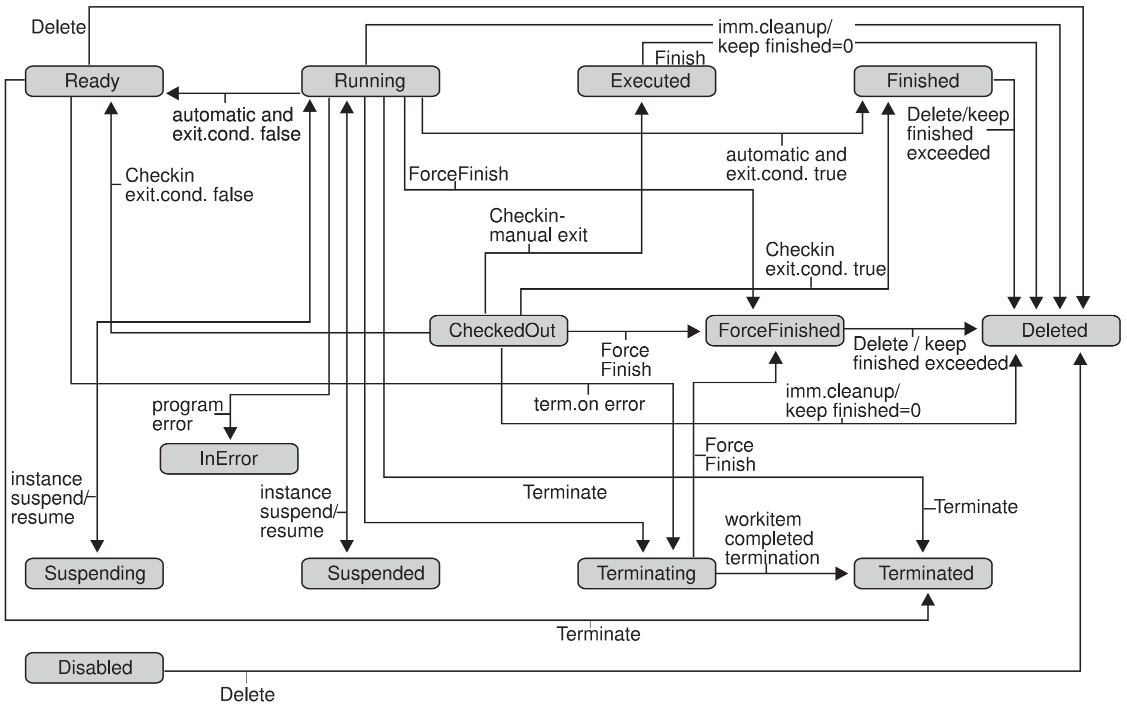


Figure 24. Work item states - process instance state terminating or terminated

FmcjWorkitemCheckIn()

This function/subprogram allows for the check in of a work item that was previously checked out for user processing (action call).

Checking in a work item tells MQSeries Workflow that user processing has finished and workflow processing under the control of MQSeries Workflow can continue. The return code of the user processing and, optionally, the output container values are passed back to MQSeries Workflow. As usual, these container values and the return code can be used in exit conditions to let navigation continue depending on the success of the processing and in transition conditions to indicate how to proceed.

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

One of:

- Be the work item owner
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjWorkitemCheckIn(
       FmcjWorkitemHandle           hdlWorkitem,
       FmcjReadWriteContainerHandle  output,
       long                         returnCode )
```

Cobol language signature

Cobol language signature

```
          PERFORM FmcjWICheckIn.

       FmcjWICheckIn.

          CALL    "FmcjWorkitemCheckIn"
                         USING
                         BY VALUE
                            hdlWorkitem
                            outputValue
                            returnCode
                         RETURNING
                            intReturnValue.
```

Parameters

hdlWorkitem
Input. The handle of the work item to be dealt with.

output
Input. A handle or pointer to the output container; can be a NULL pointer.

returnCode
Input. The return code of user processing.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The work item does no longer exist.

FMC_ERROR_NOT_AUTHORIZED(119)
Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_WRONG_STATE(120)
The work item is not checked out.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjWorkitemCheckOut()

This function/subprogram checks out a ready work item for user processing (action call).

This means that processing is not done by MQSeries Workflow's inherent program-invocation mechanism. MQSeries Workflow assumes that processing is done by user-specific means and changes the state of the work item to *CheckedOut*.

The following enumeration constants can be used to specify the requested program data; it is strongly advised to use the symbolic names instead of the integer values:

C language	integer value
Fmc_WS_NotSet	0
Fmc_WS_CommonDataOnly	1
Fmc_WS_SpecifiedDefinitions	2
Fmc_WS_AllDefinitions	4

The following enumeration constants can be used to specify the platform; it is strongly advised to use the symbolic names instead of the integer values:

C language	integer value
Fmc_DP_NotSet	0
Fmc_DP_OS2	1
Fmc_DP_AIX	2
Fmc_DP_Windows95	4
Fmc_DP_WindowsNT	5
Fmc_DP_OS390	6

The requested program definition is then returned.

CommonDataOnly
returns only data common to all platforms, the description, the icon, the unattended indicator, and the input and output containers. Any platform specification is ignored.

SpecifiedDefinitions
returns the program definition for the specified platform. A platform must be specified.

AllDefinitions
returns all available program definitions. Any platform specification is ignored.

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

One of:

- Be the work item owner

- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjWorkitemCheckOut(
       FmcjWorkitemHandle                   hdlWorkitem,
       enum FmcjWorkitemProgramRetrieval  requestedData,
       enum FmcjImplementationDataBasis   platform,
       FmcjProgramDataHandle *            programData  )
```

Cobol language signature

Cobol language signature

```
            PERFORM FmcjWICheckOut.

         FmcjWICheckOut.

            CALL    "FmcjWorkitemCheckOut"
                           USING
                           BY VALUE
                              hdlWorkitem
                              requestedData
                              platform
                           BY REFERENCE
                              programData
                           RETURNING
                              intReturnValue.
```

Parameters

hdlWorkitem
Input. The handle of the work item to be dealt with.

requestedData
Input. An indicator which program definitions are to be returned.

platform
Input. The platform for which the program definition is to be returned.

programData
Input/Output. The address of a handle to the program definition respectively the program definition object to be set.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The work item does no longer exist.

FMC_ERROR_NOT_AUTHORIZED(119)
Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_WRONG_STATE(120)
The work item is in the wrong state.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjWorkitemFinish()

This function/subprogram ends the execution of a manual-exit work item (action call).

The work item must be in state *Executed*, that is, must run at least once. The work item is then put into the *Finished* state. Depending on the “delete finished items” option, it is deleted.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

One of:

- Be the work item owner
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjWorkitemFinish(
       FmcjWorkitemHandle hdlWorkitem )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjWIFinish.

        FmcjWIFinish.

           CALL    "FmcjWorkitemFinish"
                          USING
                          BY VALUE
                             hdlWorkitem
                          RETURNING
                             intReturnValue.
```

Parameters

hdlWorkitem
: Input. The handle of the work item to be dealt with.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The work item does no longer exist.

FMC_ERROR_NOT_AUTHORIZED(119)
: Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_WRONG_STATE(120)
: The work item is in the wrong state.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjWorkitemForceFinish()

This function/subprogram ends the execution of a work item which is known to have completed in cases where MQSeries Workflow did not recognize this event (action call).

This situation can occur when the execution server aborted before it received the activity implementation completion message.

The work item must be in states *Ready*, *Running*, *Executed*, *InError*, *Terminating*, or *Terminated*. The process instance in the states *Running*, *Suspending*, *Suspended*, or *Terminating*.

The work item is then put into the *ForceFinished* state. The exit condition is considered to be true and navigation proceeds.

Depending on the "delete finished items" option, work item is deleted.

Note: Currently, only a work item implemented by a program (activity kind) can be force finished.

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

One of:

- Process administration authorization
- Be the process administrator
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjWorkitemForceFinish(
       FmcjWorkitemHandle hdlWorkitem )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjWIForceFinish.

        FmcjWIForceFinish.

           CALL    "FmcjWorkitemForceFinish"
                          USING
                          BY VALUE
                             hdlWorkitem
                          RETURNING
                             intReturnValue.
```

Parameters

hdlWorkitem
: Input. The handle of the work item to be dealt with.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The work item does no longer exist.

FMC_ERROR_NOT_AUTHORIZED(119)
: Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_WRONG_STATE(120)
: The work item is in the wrong state.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjWorkitemForceRestart()

This function/subprogram forces MQSeries Workflow to enable the restart of a work item (action call).

The work item must be in state *Running*, *Executed*, *CheckedOut*, *InError*, *Terminating*, or *Terminated*. The process instance must be in states *Running*, *Suspending*, or *Suspended*.

It is then reset into the *Ready* state. Note that automatic activity instances must now be started manually.

Note: Currently, only a work item implemented by a program (activity kind) can be force restarted.

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

One of:

- Process administration authorization
- Be the process administrator
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjWorkitemForceRestart(
      FmcjWorkitemHandle  hdlWorkitem )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjWIForceRestart.

        FmcjWIForceRestart.

           CALL    "FmcjWorkitemForceRestart"
                          USING
                          BY VALUE
                             hdlWorkitem
                          RETURNING
                             intReturnValue.
```

Parameters

hdlWorkitem
: Input. The handle of the work item to be dealt with.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The work item does no longer exist.

FMC_ERROR_NOT_AUTHORIZED(119)
: Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_WRONG_STATE(120)
: The work item is in the wrong state.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
: Timeout has occurred.

FmcjWorkitemInContainer()

This function/subprogram retrieves the input container associated with the work item from the MQSeries Workflow execution server (action call).

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

One of:

- Be the work item owner
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjWorkitemInContainer(
       FmcjWorkitemHandle             hdlWorkitem,
       FmcjReadOnlyContainerHandle *  input )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjWIInCtnr.

        FmcjWIInCtnr.

           CALL    "FmcjWorkitemInContainer"
                          USING
                          BY VALUE
                             hdlWorkitem
                          BY REFERENCE
                             inputValue
                          RETURNING
                             intReturnValue.
```

Parameters

hdlWorkitem
Input. The handle of the work item to be dealt with.

input Input/Output. The input container.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The work item does no longer exist.

FMC_ERROR_NOT_AUTHORIZED(119)
Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjWorkitemOutContainer()

This function/subprogram retrieves the output container associated with the work item from the MQSeries Workflow execution server (action call).

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

One of:

- Be the work item owner
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjWorkitemOutContainer(
       FmcjWorkitemHandle              hdlWorkitem,
       FmcjReadWriteContainerHandle *  output )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjWIOutCtnr.

        FmcjWIOutCtnr.

           CALL    "FmcjWorkitemOutContainer"
                          USING
                          BY VALUE
                             hdlWorkitem
                          BY REFERENCE
                             outputValue
                          RETURNING
                             intReturnValue.
```

Parameters

hdlWorkitem
: Input. The handle of the work item to be dealt with.

output
: Input/Output. The output container.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The work item does no longer exist.

FMC_ERROR_NOT_AUTHORIZED(119)
: Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjWorkitemPersistentObject()

This function/subprogram retrieves the work item identified by the passed object identifier from the MQSeries Workflow execution server (action call).

The MQSeries Workflow execution server from which the work item is to be retrieved is identified by the service object. The work item handle to be initialized must be a null pointer respectively the work item object to be initialized must be empty. The transient object is then updated with all information - primary and secondary - of the work item.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signatures

C language signature

```
APIRET FMC_APIENTRY FmcjWorkitemPersistentObject(
       FmcjExecutionServiceHandle service,
       char const *               oid,
       FmcjWorkitemHandle *       hdlWorkitem )
```

Cobol language signatures

Cobol language signature

```
     PERFORM FmcjWIPersistentObj.

   FmcjWIPersistentObj.

     CALL    "FmcjWorkitemPersistentObject"
                  USING
                  BY VALUE
                     serviceValue
                     oid
                  BY REFERENCE
                     hdlWorkitem
                  RETURNING
                     intReturnValue.
```

Parameters

service
: Input. The service object representing the session with the execution server.

oid
: Input. The object identifier of the work item to be retrieved.

hdlWorkitem
: Input/Output. The address of the handle to the work item object to be set.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The work item does no longer exist.

FMC_ERROR_INVALID_OID(805)
: The provided oid is invalid.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
: Timeout has occurred.

FmcjWorkitemRestart()

This function/subprogram asks MQSeries Workflow to enable the restart of a work item (action call).

The work item must be in state *Executed*. It is then reset into the *Ready* state.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

One of:

- Be the work item owner
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjWorkitemRestart(
       FmcjWorkitemHandle   hdlWorkitem )
```

Cobol language signature

Cobol language signature

```
            PERFORM FmcjWIRestart.

         FmcjWIRestart.

            CALL    "FmcjWorkitemRestart"
                           USING
                           BY VALUE
                              hdlWorkitem
                           RETURNING
                              intReturnValue.
```

Parameters

hdlWorkitem
Input. The handle of the work item to be dealt with.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The work item does no longer exist.

FMC_ERROR_NOT_AUTHORIZED(119)
: Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_WRONG_STATE(120)
: The work item is in the wrong state.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
: Timeout has occurred.

FmcjWorkitemStart()

This function/subprogram starts a ready work item (action call).

If the associated activity instance is implemented by a program, the program is started on the program execution server associated to the logged-on user. The work item is put into the *Running* state. If the activity implementation or an associated process activity cannot be started, the work item is put into the *InError* state.

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

One of:

- Be the work item owner
- Be the system administrator

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjWorkitemStart(
       FmcjWorkitemHandle  hdlWorkitem )
```

Cobol language signature

Cobol language signature

```
            PERFORM FmcjWIStart.

         FmcjWIStart.

            CALL    "FmcjWorkitemStart"
                           USING
                           BY VALUE
                              hdlWorkitem
                           RETURNING
                              intReturnValue.
```

Parameters

hdlWorkitem
Input. The handle of the work item to be dealt with.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The work item does no longer exist.

FMC_ERROR_NOT_AUTHORIZED(119)
Not authorized to use the function/subprogram.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_WRONG_STATE(120)
The work item is in the wrong state.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

FmcjWorkitemTerminate()

This function/subprogram terminates a work item (action call).

The work item may be in states *Ready*, *Running*, *InError*, or *Planning*. The work item is then put into state *terminated* or *terminating*.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

One of

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjWorkitemTerminate(
      FmcjWorkitemHandle  hdlWorkitem )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjWITerminate.

        FmcjWITerminate.

           CALL    "FmcjWorkitemTerminate"
                          USING
                          BY VALUE
                             hdlWorkitem
                          RETURNING
                             intReturnValue.
```

Parameters

hdlWorkitem
Input. The handle of the work item to be dealt with.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_NOT_AUTHORIZED(119)
Not authorized to use the function/subprogram.

FMC_ERROR_DOES_NOT_EXIST(118)
The work item does no longer exist.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

Chapter 47. FmcjWorklist functions/subprograms

An FmcjWorklist object represents a set of items. All items which are accessible through this list have the same characteristics. These characteristics are specified by a filter. Additionally, sort criteria can be applied and, after that, a threshold to restrict the number of items to be transferred from the execution server to the client.

The worklist definition is stored persistently. The items contained in the worklist are, however, assembled dynamically when they are queried.

In the C++ implementation, FmcjWorklist is a subclass of FmcjPersistentList and thus inherits all its methods. Similarly, in the C and Cobol language API, all functions of FmcjPersistentList are also applicable to FmcjWorklist.

FmcjWorklistQueryActivityInstanceNotifications()

This function/subprogram retrieves the primary information for all activity instance notifications characterized by the specified worklist from the MQSeries Workflow execution server (action call).

From the set of qualifying activity instance notifications, only those are retrieved, the user is authorized for. The user is authorized for an activity instance notification if

- He is the owner of the activity instance notification
- He has workitem authority
- He is the system administrator

The primary information that is retrieved for each activity instance notification is:

- ActivityType
- Category
- CreationTime
- Description
- Icon
- Implementation
- Kind
- LastModificationTime
- Name
- Owner
- Priority
- ProcessInstanceName
- ReceivedAs
- ReceivedTime
- StartTime
- State
- SupportTools

Any activity instance notifications retrieved are appended to the supplied vector of activity instance notifications. If you want to read those activity instance notifications only which are currently included in the worklist, you have to clear the vector before you call this function/subprogram. This means that you should set the handle to 0 via Deallocate.

Properties

Usage notes

- See "Action functions/subprograms" on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjWorklistQueryActivityInstanceNotifications(
       FmcjWorklistHandle                            hdlList,
       FmcjActivityInstanceNotificationVectorHandle *  notifications )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjWLQueryActInstNotifs.

        FmcjWLQueryActInstNotifs.

           CALL    "FmcjWorklistQueryActivityInstanceNotifications"
                          USING
                          BY VALUE
                             hdlList
                          BY REFERENCE
                             notifications
                          RETURNING
                             intReturnValue.
```

Parameters

hdlList
Input. The handle of the worklist to be queried.

notifications
Input/Output. The vector of qualifying activity instance notifications.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The worklist does no longer exist.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
: Timeout has occurred.

Examples

- For a C language example see "Query worklists (C language)" on page 447
- For a Cobol example see "Query worklists (Cobol language)" on page 448

FmcjWorklistQueryItems()

This function/subprogram retrieves the primary information for all items characterized by the specified worklist from the MQSeries Workflow execution server (action call).

From the set of qualifying items, only those are retrieved, the user is authorized for. The user is authorized for an item if

- He is the owner of the item
- He has workitem authority
- He is the system administrator

The primary information that is retrieved for each item is:

- Category
- CreationTime
- Description
- Icon
- Kind
- LastModificationTime

- Name
- Owner
- ProcessInstanceName
- ReceivedAs
- ReceivedTime
- StartTime
- State

If the item is an actual work item or an activity instance notification, then additional primary information is retrieved:
- ActivityType
- Implementation
- Priority
- SupportTools

Any items retrieved are appended to the supplied vector of items. If you want to read those items only which are currently included in the worklist, you have to clear the vector before you call this function/subprogram. This means that you should set the handle to 0 via Deallocate.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjWorklistQueryItems(
       FmcjWorklistHandle      hdlList,
       FmcjItemVectorHandle *  items )
```

Cobol language signature

Cobol language signature

```
           PERFORM FmcjWLQueryItems.

        FmcjWLQueryItems.

           CALL    "FmcjWorklistQueryItems"
                          USING
                          BY VALUE
                             hdlList
                          BY REFERENCE
                             items
                          RETURNING
                             intReturnValue.
```

Parameters

hdlList
Input. The handle of the worklist to be queried.

items
Input/Output. The vector of qualifying items.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The worklist does no longer exist.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Examples

- For a C language example see “Query worklists (C language)” on page 447
- For a Cobol example see “Query worklists (Cobol language)” on page 448

FmcjWorklistQueryProcessInstanceNotifications()

This function/subprogram retrieves the primary information for all process instance notifications characterized by the specified worklist from the MQSeries Workflow execution server (action call).

From the set of qualifying process instance notifications, only those are retrieved, the user is authorized for. The user is authorized for a process instance notification if

- He is the owner of the process instance notification
- He has workitem authority
- He is the system administrator

The primary information that is retrieved for each process instance notification is:

- Category
- CreationTime
- Description
- Icon
- Kind
- LastModificationTime
- Name
- Owner
- ProcessInstanceName
- ReceivedAs
- ReceivedTime
- StartTime
- State

Any process instance notifications retrieved are appended to the supplied vector of process instance notifications. If you want to read those process instance notifications only which are currently included in the worklist, you have to clear the vector before you call this function/subprogram. This means that you should set the handle to 0 via Deallocate.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjWorklistQueryProcessInstanceNotifications(
       FmcjWorklistHandle                             hdlList,
       FmcjProcessInstanceNotificationVectorHandle *  notifications )
```

Cobol language signature

Cobol language signature

```
            PERFORM FmcjWLQueryProcInstNotifs.

        FmcjWLQueryProcInstNotifs.

            CALL    "FmcjWorklistQueryProcessInstanceNotifications"
                           USING
                           BY VALUE
                              hdlList
                           BY REFERENCE
                              notifications
                           RETURNING
                              intReturnValue.
```

Parameters

hdlList
: Input. The handle of the worklist to be queried.

notifications
: Input/Output. The vector of qualifying process instance notifications.

Return codes

FMC_OK(0)
: The function/subprogram completed successfully.

FMC_ERROR(1)
: A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
: The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
: The worklist does no longer exist.

FMC_ERROR_NOT_LOGGED_ON(106)
: Not logged on.

FMC_ERROR_COMMUNICATION(13)
: The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
: An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
: An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Examples

- For a C language example see “Query worklists (C language)” on page 447
- For a Cobol example see “Query worklists (Cobol language)” on page 448

FmcjWorklistQueryWorkitems()

This function/subprogram retrieves the primary information for all work items characterized by the specified worklist from the MQSeries Workflow execution server (action call).

From the set of qualifying work items, only those are retrieved, the user is authorized for. The user is authorized for a work item if

- He is the owner of the work item
- He has workitem authority
- He is the system administrator

The primary information that is retrieved for each work item is:

- ActivityType
- Category
- CreationTime
- Description
- Icon
- Implementation
- Kind
- LastModificationTime
- Name
- Owner
- Priority
- ProcessInstanceName
- ReceivedAs
- ReceivedTime
- StartTime
- State
- SupportTools

Any work items retrieved are appended to the supplied vector of work items. If you want to read those work items only which are currently included in the worklist, you have to clear the vector before you call this function/subprogram. This means that you should set the handle to 0 via Deallocate.

Properties

Usage notes

- See “Action functions/subprograms” on page 76 for general information.

Authorization

None

Required connection

MQSeries Workflow execution server

API include file

Runtime: fmcjcrun.h (C language) respectively fmcvars.cpy and fmcperf.cpy (Cobol language)

C language signature

C language signature

```
APIRET FMC_APIENTRY FmcjWorklistQueryWorkitems(
       FmcjWorklistHandle          hdlList,
       FmcjWorkitemVectorHandle *  workitems )
```

Cobol language signature

Cobol language signature

```
            PERFORM FmcjWLQueryWorkitems.

         FmcjWLQueryWorkitems.

            CALL    "FmcjWorklistQueryWorkitems"
                           USING
                           BY VALUE
                              hdlList
                           BY REFERENCE
                              workitems
                           RETURNING
                              intReturnValue.
```

Parameters

hdlList
Input. The handle of the worklist to be queried.

workitems
Input/Output. The vector of qualifying work items.

Return codes

FMC_OK(0)
The function/subprogram completed successfully.

FMC_ERROR(1)
A parameter references an undefined location. For example, the address of a handle is 0.

FMC_ERROR_INVALID_HANDLE(130)
The handle provided is invalid; it is 0 or it is not pointing to an object of the requested type.

FMC_ERROR_DOES_NOT_EXIST(118)
The worklist does no longer exist.

FMC_ERROR_NOT_LOGGED_ON(106)
Not logged on.

FMC_ERROR_COMMUNICATION(13)
The specified server cannot be reached; maybe the server to connect to is not defined in your profile.

FMC_ERROR_INTERNAL(100)
An MQSeries Workflow internal error has occurred. Contact your IBM representative.

FMC_ERROR_MESSAGE_FORMAT(103)
An internal message format error. Contact your IBM representative.

FMC_ERROR_TIMEOUT(14)
Timeout has occurred.

Examples

- For a C language example see "Query worklists (C language)" on page 447
- For a Cobol example see "Query worklists (Cobol language)" on page 448

The following chapter shows some examples. They are intended to present the special concept stated in the heading. This means that they are not always complete and may not destruct all objects created or retrieved.

Part 8. Examples

Chapter 48. How to create persistent lists

The following examples show how to create a persistent list, that is, a persistent view on a set of objects. They define a view on process instances. Other possible lists to define are process template lists or worklists.

Create a process instance list (C language)

```
#include <stdio.h>
#include <fmcjcrun.h>                  /* MQ Workflow Runtime API */
int main()
{
  APIRET                            rc           = FMC_OK;
  FmcjExecutionServiceHandle        service      = 0;
  FmcjProcessInstanceListHandle     instanceList = 0;
  unsigned long                     threshold    = 10;
  int                               enumValue    = 0;

  char name[50]  = "MyTenInstances";
  char desc[50]   = "This list contains no more than 10 instances";

  FmcjGlobalConnect();

  /* logon */
  rc= FmcjExecutionServiceAllocate( &service );
  if (rc != FMC_OK)
  {
    printf("Service object could not be allocated - rc: %u%\n",rc);
    return -1;
  }

  rc= FmcjExecutionServiceLogon( service,
                                 "USERID", "password",
                                 Fmc_SM_Default, Fmc_SA_NotSet
                               );
  if (rc != FMC_OK)
  {
    printf("Logon failed - rc: u\n",rc);
    FmcjExecutionServiceDeallocate( &service );
    return -1;
  }

  /* create a process instance list */
  rc = FmcjExecutionServiceCreateProcessInstanceList(
                         service,
                         name,
                         Fmc_LT_Private,
                         "USERID",
                         desc,
                         FmcjNoFilter,
                         FmcjNoSortCriteria,
                         &threshold,
                         &instanceList );
  if ( rc != FMC_OK)
     printf( "CreateProcessInstanceList returns: %u%\n",rc );
  else
     printf( "CreateProcessInstanceList okay\n" );

  FmcjExecutionServiceLogoff( service );
  FmcjExecutionServiceDeallocate( &service );
  FmcjGlobalDisconnect();
  return 0;
}
```

Create a process instance list (Cobol language)

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. "VECTOR".

 DATA DIVISION.

   WORKING-STORAGE SECTION.

   COPY fmcvars.
   COPY fmcconst.
   COPY fmcrcs.

   01 localUserID   PIC X(30) VALUE z"USERID".
   01 localPassword PIC X(30) VALUE z"PASSWORD".
   01 listName      PIC X(50) VALUE z"MyTenInstances".
   01 desc          PIC X(50)
         VALUE z"This list contains no more than 10 instances".

   LINKAGE SECTION.

   01 retCode            PIC S9(9) BINARY.

 PROCEDURE DIVISION USING retCode.

      PERFORM FmcjGlobalConnect.

* logon
      PERFORM FmcjESAllocate.
      MOVE intReturnValue TO retCode
      IF retCode NOT = FMC-OK
         DISPLAY "Service object could not be allocated"
         DISPLAY "rc: " retCode
         MOVE -1 TO retCode
         GOBACK
      END-IF

      CALL "SETADDR" USING localUserId userId.
      CALL "SETADDR" USING localPassword passwordValue.
      MOVE Fmc-SM-Default TO sessionMode.
      MOVE Fmc-SA-Reset TO absenceIndicator.
      PERFORM FmcjESLogon.

      MOVE intReturnValue TO retCode
      IF retCode NOT = FMC-OK
         DISPLAY "Logon failed - rc: " retCode
         PERFORM FmcjESDeallocate
         MOVE -1 TO retCode
         GOBACK
      END-IF

* create a process instance list
      CALL "SETADDR" USING listName name.
      CALL "SETADDR" USING localUserID ownerValue.
      CALL "SETADDR" USING desc description.
      CALL "SETADDR" USING FmcjNoFilter filter.
      CALL "SETADDR" USING FmcjNoSortCriteria sortCriteria.
      MOVE FmcjNoThreshold TO threshold.
      MOVE Fmc-LT-Private TO typeValue.
      PERFORM FmcjESCreateProcInstList.

      MOVE intReturnValue TO retCode
      IF retCode NOT = FMC-OK
         DISPLAY "CreateProcessInstanceList returns - rc: "
         DISPLAY retCode
      ELSE
         DISPLAY "CreateProcessInstanceList okay"
      END-IF

      PERFORM FmcjESLogoff.
      PERFORM FmcjESDeallocate.
      PERFORM FmcjGlobalDisconnect.
      MOVE FMC-OK TO retCode.
      GOBACK.

      COPY fmcperf.
 IDENTIFICATION DIVISION.
 PROGRAM-ID. "VECTOR".

 DATA DIVISION.

   WORKING-STORAGE SECTION.

   COPY fmcvars.
   COPY fmcconst.
   COPY fmcrcs.

   01 localUserID   PIC X(30) VALUE z"USERID".
   01 localPassword PIC X(30) VALUE z"PASSWORD".
   01 listName      PIC X(50) VALUE z"MyTenInstances".
   01 desc          PIC X(50)
         VALUE z"This list contains no more than 10 instances".

   LINKAGE SECTION.

   01 retCode            PIC S9(9) BINARY.

 PROCEDURE DIVISION USING retCode.

      CALL "FmcjGlobalConnect".
* logon
      CALL "FmcjExecutionServiceAllocate"
         USING BY REFERENCE serviceValue
         RETURNING intReturnValue.
      MOVE intReturnValue TO retCode
      IF retCode NOT = FMC-OK
         DISPLAY "Service object could not be allocated"
         DISPLAY "rc: " retCode
         MOVE -1 TO retCode
         GOBACK
      END-IF

      CALL "SETADDR" USING localUserId userId.
      CALL "SETADDR" USING localPassword passwordValue.
      CALL "FmcjExecutionServiceLogon"
         USING BY VALUE serviceValue
                        userID
                        passwordValue
                        Fmc-SM-Default
                        Fmc-SA-Reset
         RETURNING intReturnValue.

      MOVE intReturnValue TO retCode
      IF retCode NOT = FMC-OK
         DISPLAY "Logon failed - rc: " retCode
         CALL "FmcjExecutionServiceDeallocate"
            USING BY REFERENCE serviceValue
            RETURNING intReturnValue
         MOVE -1 TO retCode
```

```
            GOBACK
         END-IF

   * create a process instance list
         CALL "SETADDR" USING listName name.
         CALL "SETADDR" USING localUserID ownerValue.
         CALL "SETADDR" USING desc description.
         CALL "SETADDR" USING FmcjNoFilter filter.
         CALL "SETADDR" USING FmcjNoSortCriteria sortCriteria.
         CALL "FmcjExecutionServiceCreateProcessInstanceList"
            USING BY VALUE serviceValue
                           name
                           Fmc-LT-Private
                           ownerValue
                           description
                           filter
                           sortCriteria
                           FmcjNoThreshold
                        BY REFERENCE
                           newList
                        RETURNING
                           intReturnValue.

         MOVE intReturnValue TO retCode
         IF retCode NOT = FMC-OK
            DISPLAY "CreateProcessInstanceList returns - rc: "
            DISPLAY retCode
         ELSE
            DISPLAY "CreateProcessInstanceList okay"
         END-IF

         CALL "FmcjExecutionServiceLogoff"
            USING BY VALUE serviceValue
            RETURNING intReturnValue.
         CALL "FmcjExecutionServiceDeallocate"
            USING BY REFERENCE serviceValue
            RETURNING intReturnValue.
         CALL "FmcjGlobalDisconnect".
         MOVE FMC-OK TO retCode.
         GOBACK.
```

Chapter 49. How to query persistent lists

The following examples show how to retrieve persistent lists from the MQSeries Workflow execution server and how to query the characteristics of a list. They use worklists as example. Other possible lists to query are process template lists or process instance lists.

Query worklists (C language)

```
#include <stdio.h>
#include <memory.h>
#include <fmcjcrun.h>                  /* MQ Workflow Runtime API */
int main()
{
  APIRET                       rc           = FMC_OK;
  FmcjExecutionServiceHandle   service      = 0;
  FmcjWorklistHandle           worklist     = 0;
  FmcjWorklistVectorHandle     lists        = 0;
  unsigned long                numWList     = 0;
  unsigned long                i            = 0;
  unsigned long                enumValue    = 0;
  char                         tInfo[4096+1]= "";

  FmcjGlobalConnect();

  /* logon */
  rc= FmcjExecutionServiceAllocate( &service );
  if (rc != FMC_OK)
  {
    printf("Service object could not be allocated - rc: %u%\n",rc);
    return -1;
  }

  rc= FmcjExecutionServiceLogon( service,
                                 "USERID", "password",
                                 Fmc_SM_Default, Fmc_SA_NotSet
                               );
  if (rc != FMC_OK)
  {
    printf("Logon failed - rc: %u%\n",rc);
    FmcjExecutionServiceDeallocate( &service );
    return -1;
  }

  /* query worklists  */
  rc = FmcjExecutionServiceQueryWorklists( service, &lists );
  if ( rc != FMC_OK)
     printf( "QueryWorklists() returns: %u%\n",rc );
  else
     printf( "QueryWorklists() returns okay\n" );

  if (rc == FMC_OK)
  {
    numWList= FmcjWorklistVectorSize(lists);
    printf ("Number of worklists returned : %u\n", numWList);

    for( i=1; i<= numWList; i++ )
    {
      worklist= FmcjWorklistVectorNextElement(lists);
      FmcjWorklistName( worklist, tInfo, 4097 );
      printf("- Name                              : %s\n",tInfo);
      enumValue= FmcjWorklistType(worklist);
      if ( enumValue == Fmc_LT_Private )
        printf("- Type                           : %s\n","private");
      if ( enumValue == Fmc_LT_Public )
        printf("- Type                           : %s\n","public");
      FmcjWorklistOwnerOfList( worklist, tInfo, 4097 );
      printf("- OwnerOfList                    : %s\n",tInfo);
      printf("- OwnerOfList is null ?          : %u\n",
              FmcjWorklistOwnerOfListIsNull(worklist) );
      FmcjWorklistDescription( worklist, tInfo, 4097 );
      printf("- Description                    : %s\n",tInfo);
      printf("- Description is null ?          : %u\n",
              FmcjWorklistDescriptionIsNull(worklist) );
      FmcjWorklistFilter( worklist, tInfo, 4097 );
      printf("- Filter                         : %s\n",tInfo);
      printf("- Filter is null ?               : %u\n",
              FmcjWorklistFilterIsNull(worklist) );
      FmcjWorklistSortCriteria( worklist, tInfo, 4097 );
      printf("- SortCriteria                   : %s\n",tInfo);
      printf("- SortCriteria is null ?         : %u\n",
              FmcjWorklistSortCriteriaIsNull(worklist) );
      printf("- Threshold                      : %u\n",
              FmcjWorklistThreshold(worklist) );
      printf("- Threshold is null ?            : %u\n",
              FmcjWorklistThresholdIsNull(worklist) );

                                 /* deallocate just read object  */
      FmcjWorklistDeallocate(&worklists );
    }

    FmcjWorklistVectorDeallocate(&lists );
  }

  FmcjExecutionServiceLogoff( service );
  FmcjExecutionServiceDeallocate( &service );

  FmcjGlobalDisconnect();
  return 0;
}
```

Query worklists (Cobol language)

```
        IDENTIFICATION DIVISION.
        PROGRAM-ID. "QUERYWL".

        DATA DIVISION.

          WORKING-STORAGE SECTION.

          COPY fmcvars.
          COPY fmcconst.
          COPY fmcrcs.

          01 localUserID   PIC X(30) VALUE z"USERID".
          01 localPassword PIC X(30) VALUE z"PASSWORD".
          01 numWList      PIC 9(9) BINARY VALUE 0.
          01 tInfo         PIC X(4097).
          01 i             PIC 9(9) BINARY VALUE 0.

          LINKAGE SECTION.

          01 retCode            PIC S9(9) BINARY.

        PROCEDURE DIVISION USING retCode.

             PERFORM FmcjGlobalConnect.

       * logon
```

```
           PERFORM FmcjESAllocate.
           MOVE intReturnValue TO retCode
           IF retCode NOT = FMC-OK
              DISPLAY "Service object could not be allocated"
              DISPLAY "rc: " retCode
              MOVE -1 TO retCode
              GOBACK
           END-IF

           CALL "SETADDR" USING localUserId userId.
           CALL "SETADDR" USING localPassword passwordValue.
           MOVE Fmc-SM-Default TO sessionMode.
           MOVE Fmc-SA-Reset TO absenceIndicator.
           PERFORM FmcjESLogon.

           MOVE intReturnValue TO retCode
           IF retCode NOT = FMC-OK
              DISPLAY "Logon failed - rc: " retCode
              PERFORM FmcjESDeallocate
              MOVE -1 TO retCode
              GOBACK
           END-IF

      * query worklists
           PERFORM FmcjESQueryWorklists.
           MOVE intReturnValue TO retCode
           IF retCode NOT = FMC-OK
              DISPLAY "QueryWorklists returns - rc: " retCode
           ELSE
              DISPLAY "QueryWorklists returns okay"
           END-IF

           IF retCode = FMC-OK
              SET hdlVector TO lists
              PERFORM FmcjWLVectorSize
              MOVE ulongReturnValue TO numWList
              DISPLAY "Number of worklists returned : " numWList
              PERFORM VARYING i FROM 1 BY 1 UNTIL i >= numWList
                 PERFORM FmcjWLVectorNextElement
                 SET hdlList TO FmcjWLHandleReturnValue
                 MOVE 4097 TO bufferLength
                 CALL "SETADDR" USING tInfo listNameBuffer
                 PERFORM FmcjWLName
                 DISPLAY "- Name               : " tInfo
                 PERFORM FmcjWLType
                 IF intReturnValue = Fmc-LT-Private
                    DISPLAY "- Type               : private"
                 END-IF
                 IF intReturnValue = Fmc-LT-Public
                    DISPLAY "- Type               : public"
                 END-IF
                 CALL "SETADDR" USING tInfo userIdBuffer
                 PERFORM FmcjWLOwnerOfList
                 DISPLAY "- OwnerOfList        : " tInfo
                 PERFORM FmcjWLOwnerOfListIsNull
                 IF boolReturnValue = 0
                    DISPLAY "- OwnerOfList is null ? : false"
                 ELSE
                    DISPLAY "- OwnerOfList is null ? : true"
                 END-IF
                 CALL "SETADDR" USING tInfo descriptionBuffer
                 PERFORM FmcjWLDescription
                 DISPLAY "- Description        : " tInfo
                 PERFORM FmcjWLDescriptionIsNull
                 IF boolReturnValue = 0
                    DISPLAY "- Description is null ? : false"
                 ELSE
```

```
                DISPLAY "- Description is null ? : true"
             END-IF
             CALL "SETADDR" USING tInfo filterBuffer
             PERFORM FmcjWLFilter
             DISPLAY "- Filter             : " tInfo
             PERFORM FmcjWLFilterIsNull
             IF boolReturnValue = 0
                DISPLAY "- Filter is null ?      : false"
             ELSE
                DISPLAY "- Filter is null ?      : true"
             END-IF
             CALL "SETADDR" USING tInfo sortCriteriaBuffer
             PERFORM FmcjWLSortCriteria
             DISPLAY "- SortCriteria       : " tInfo
             PERFORM FmcjWLSortCriteriaIsNull
             IF boolReturnValue = 0
                DISPLAY "- SortCriteria is null ?: false"
             ELSE
                DISPLAY "- SortCriteria is null ?: true"
             END-IF
             PERFORM FmcjWLThreshold
             DISPLAY "- Threshold          : " ulongReturnValue
             PERFORM FmcjWLThresholdIsNull
             IF boolReturnValue = 0
                DISPLAY "- Threshold is null ?   : false"
             ELSE
                DISPLAY "- Threshold is null ?   : true"
             END-IF
             PERFORM FmcjWLDeallocate
          END-PERFORM
          PERFORM FmcjWLVectorDeallocate
       END-IF

       PERFORM FmcjESLogoff.
       PERFORM FmcjESDeallocate.
       PERFORM FmcjGlobalDisconnect.
       MOVE FMC-OK TO retCode.
       GOBACK.

       COPY fmcperf.
 IDENTIFICATION DIVISION.
 PROGRAM-ID. "QUERYWL".

 DATA DIVISION.

   WORKING-STORAGE SECTION.

   COPY fmcvars.
   COPY fmcconst.
   COPY fmcrcs.

   01 localUserID   PIC X(30) VALUE z"USERID".
   01 localPassword PIC X(30) VALUE z"PASSWORD".
   01 numWList      PIC 9(9) BINARY VALUE 0.
   01 tInfo         PIC X(4097).
   01 i             PIC 9(9) BINARY VALUE 0.
   01 bufferPtr     USAGE IS POINTER VALUE NULL.

   LINKAGE SECTION.

   01 retCode            PIC S9(9) BINARY.

 PROCEDURE DIVISION USING retCode.

       CALL "FmcjGlobalConnect".
* logon
       CALL "FmcjExecutionServiceAllocate"
```

```
              USING BY REFERENCE serviceValue
              RETURNING intReturnValue.
           MOVE intReturnValue TO retCode
           IF retCode NOT = FMC-OK
              DISPLAY "Service object could not be allocated"
              DISPLAY "rc: " retCode
              MOVE -1 TO retCode
              GOBACK
           END-IF

           CALL "SETADDR" USING localUserId userId.
           CALL "SETADDR" USING localPassword passwordValue.
           CALL "FmcjExecutionServiceLogon"
              USING BY VALUE serviceValue
                             userID
                             passwordValue
                             Fmc-SM-Default
                             Fmc-SA-Reset
              RETURNING intReturnValue.

           MOVE intReturnValue TO retCode
           IF retCode NOT = FMC-OK
              DISPLAY "Logon failed - rc: " retCode
              CALL "FmcjExecutionServiceDeallocate"
                 USING BY REFERENCE serviceValue
                 RETURNING intReturnValue
              MOVE -1 TO retCode
              GOBACK
           END-IF

      * query worklists
           CALL "FmcjExecutionServiceQueryWorklists"
              USING BY VALUE serviceValue
                    BY REFERENCE lists
              RETURNING intReturnValue.
           MOVE intReturnValue TO retCode
           IF retCode NOT = FMC-OK
              DISPLAY "QueryWorklists returns - rc: " retCode
           ELSE
              DISPLAY "QueryWorklists returns okay"
           END-IF

           IF retCode = FMC-OK
              SET hdlVector TO lists
              CALL "FmcjWorklistVectorSize"
                 USING BY VALUE hdlVector
                 RETURNING ulongReturnValue
              MOVE ulongReturnValue TO numWList
              DISPLAY "Number of worklists returned : " numWList
              PERFORM VARYING i FROM 1 BY 1 UNTIL i >= numWList
                 CALL "FmcjWorklistVectorNextElement"
                    USING BY VALUE hdlVector
                    RETURNING FmcjWLHandleReturnValue
                 SET hdlList TO FmcjWLHandleReturnValue
                 MOVE 4097 TO bufferLength
                 CALL "SETADDR" USING tInfo bufferPtr
                 CALL "FmcjPersistentListName"
                    USING BY VALUE hdlList
                                   bufferPtr
                                   bufferLength
                    RETURNING pointerReturnValue
                 DISPLAY "- Name               : " tInfo
                 CALL "FmcjPersistentListType"
                    USING BY VALUE hdlList
                    RETURNING intReturnValue
                 IF intReturnValue = Fmc-LT-Private
                    DISPLAY "- Type               : private"
```

```
END-IF
IF intReturnValue = Fmc-LT-Public
   DISPLAY "- Type               : public"
END-IF
CALL "FmcjPersistentListOwnerOfList"
   USING BY VALUE hdlList
                  bufferPtr
                  bufferLength
   RETURNING pointerReturnValue
DISPLAY "- OwnerOfList        : " tInfo
CALL "FmcjPersistentListOwnerOfListIsNull"
   USING BY VALUE hdlList
   RETURNING boolReturnValue
IF boolReturnValue = 0
   DISPLAY "- OwnerOfList is null ? : false"
ELSE
   DISPLAY "- OwnerOfList is null ? : true"
END-IF
CALL "FmcjPersistentListDescription"
   USING BY VALUE hdlList
                  bufferPtr
                  bufferLength
   RETURNING pointerReturnValue
DISPLAY "- Description        : " tInfo
CALL "FmcjPersistentListDescriptionIsNull"
   USING BY VALUE hdlList
   RETURNING boolReturnValue
IF boolReturnValue = 0
   DISPLAY "- Description is null ? : false"
ELSE
   DISPLAY "- Description is null ? : true"
END-IF
CALL "FmcjPersistentListFilter"
   USING BY VALUE hdlList
                  bufferPtr
                  bufferLength
   RETURNING pointerReturnValue
DISPLAY "- Filter             : " tInfo
CALL "FmcjPersistentListFilterIsNull"
   USING BY VALUE hdlList
   RETURNING boolReturnValue
IF boolReturnValue = 0
   DISPLAY "- Filter is null ?      : false"
ELSE
   DISPLAY "- Filter is null ?      : true"
END-IF
CALL "FmcjPersistentListSortCriteria"
   USING BY VALUE hdlList
                  bufferPtr
                  bufferLength
   RETURNING pointerReturnValue
DISPLAY "- SortCriteria       : " tInfo
CALL "FmcjPersistentListSortCriteriaIsNull"
   USING BY VALUE hdlList
   RETURNING boolReturnValue
IF boolReturnValue = 0
   DISPLAY "- SortCriteria is null ?: false"
ELSE
   DISPLAY "- SortCriteria is null ?: true"
END-IF
CALL "FmcjPersistentListThreshold"
   USING BY VALUE hdlList
   RETURNING ulongReturnValue
DISPLAY "- Threshold          : " ulongReturnValue
CALL "FmcjPersistentListThresholdIsNull"
   USING BY VALUE hdlList
   RETURNING boolReturnValue
```

```
         IF boolReturnValue = 0
            DISPLAY "- Threshold is null ?   : false"
         ELSE
            DISPLAY "- Threshold is null ?   : true"
         END-IF
         CALL "FmcjWorklistDeallocate"
            USING BY REFERENCE hdlList
            RETURNING intReturnValue
      END-PERFORM
   CALL "FmcjWorklistVectorDeallocate"
      USING BY REFERENCE hdlVector
      RETURNING intReturnValue
   END-IF

   CALL "FmcjExecutionServiceLogoff"
      USING BY VALUE serviceValue
      RETURNING intReturnValue.
   CALL "FmcjExecutionServiceDeallocate"
      USING BY REFERENCE serviceValue
      RETURNING intReturnValue.
   CALL "FmcjGlobalDisconnect".
   MOVE FMC-OK TO retCode.
   GOBACK.
```

Chapter 50. How to query a set of objects

The following examples show how to query objects for which you are authorized. They use a query for process instances in order to demonstrate an ad-hoc query. They use work items in order to demonstrate how to query the contents of a predefined list, a worklist.

Query process instances (C language)

```
#include <stdio.h>
#include <memory.h>
#include <fmcjcrun.h>                  /* MQ Workflow Runtime API */
int main()
{
  APIRET                          rc           = FMC_OK;
  FmcjExecutionServiceHandle      service      = 0;
  FmcjProcessInstanceHandle       instance     = 0;
  FmcjProcessInstanceVectorHandle iList        = 0;
  unsigned long                   numIList     = 0;
  unsigned long                   i            = 0;
  char                            tInfo[4096+1]= "";

  FmcjGlobalConnect();

  /* logon */
  rc= FmcjExecutionServiceAllocate( &service );
  if (rc != FMC_OK)
  {
    printf("Service object could not be allocated - rc: %u%\n",rc);
    return -1;
  }
  rc= FmcjExecutionServiceLogon( service,
                                 "USERID", "password",
                                 Fmc_SM_Default, Fmc_SA_NotSet
                               );
  if (rc != FMC_OK)
  {
    printf("Logon failed - rc: %u%\n",rc);
    FmcjExecutionServiceDeallocate( &service );
    return -1;
  }
  /* query process instances  */
  rc= FmcjExecutionServiceQueryProcessInstances(
                        service,
                        FmcjNoFilter, FmcjNoSortCriteria, FmcjNoThreshold,
                        &iList               );
  if ( rc != FMC_OK)
     printf( "QueryProcessInstances() returns: %u%\n",rc );
  else
     printf( "QueryProcessInstances() returns okay\n" );
  if (rc == FMC_OK)
  {
    numIList= FmcjProcessInstanceVectorSize(iList);
    printf ("Number of instances returned : %u\n", numIList);

    for( i=1; i<= numIList; i++ )
    {
      instance= FmcjProcessInstanceVectorNextElement(iList);
      FmcjProcessInstanceName( instance, tInfo, 4097 );
      printf("- Name                      : %s\n",tInfo);
      FmcjProcessInstanceDeallocate(&instance );
    }
    FmcjProcessInstanceVectorDeallocate(&iList );
  }

  FmcjExecutionServiceLogoff( service );
  FmcjExecutionServiceDeallocate( &service );

  FmcjGlobalDisconnect();
  return 0;
}
```

Query process instances (Cobol language)

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. "QUERYPI".

 DATA DIVISION.

   WORKING-STORAGE SECTION.

   COPY fmcvars.
   COPY fmcconst.
   COPY fmcrcs.

   01 localUserID   PIC X(30) VALUE z"USERID".
   01 localPassword PIC X(30) VALUE z"PASSWORD".
   01 numIList      PIC 9(9) BINARY VALUE 0.
   01 tInfo         PIC X(4097).
   01 i             PIC 9(9) BINARY VALUE 0.

   LINKAGE SECTION.

   01 retCode            PIC S9(9) BINARY.

 PROCEDURE DIVISION USING retCode.

      PERFORM FmcjGlobalConnect.

* logon
      PERFORM FmcjESAllocate.
      MOVE intReturnValue TO retCode
      IF retCode NOT = FMC-OK
         DISPLAY "Service object could not be allocated"
         DISPLAY "rc: " retCode
         MOVE -1 TO retCode
         GOBACK
      END-IF

      CALL "SETADDR" USING localUserId userId.
      CALL "SETADDR" USING localPassword passwordValue.
      MOVE Fmc-SM-Default TO sessionMode.
      MOVE Fmc-SA-Reset TO absenceIndicator.
      PERFORM FmcjESLogon.

      MOVE intReturnValue TO retCode
      IF retCode NOT = FMC-OK
         DISPLAY "Logon failed - rc: " retCode
         PERFORM FmcjESDeallocate
         MOVE -1 TO retCode
         GOBACK
      END-IF

* query process instances
      CALL "SETADDR" USING FmcjNoFilter filter.
      CALL "SETADDR" USING FmcjNoSortCriteria sortCriteria.
      MOVE FmcjNoThreshold TO threshold.
      PERFORM FmcjESQueryProcInsts.
```

```
           SET hdlVector TO instances.
           MOVE intReturnValue TO retCode
           IF retCode NOT = FMC-OK
              DISPLAY "QueryProcessInstances returns: " retCode
           ELSE
              DISPLAY "QueryProcessInstances returns okay"
           END-IF

           IF retCode = FMC-OK
              PERFORM FmcjPIVSize
              MOVE ulongReturnValue TO numIList
              DISPLAY "Number of instances returned: " numIList
              MOVE 4097 TO bufferLength
              CALL "SETADDR" USING tInfo instanceNameBuffer
              PERFORM VARYING i FROM 1 BY 1 UNTIL i > numIList
                 PERFORM FmcjPIVNextElement
                 SET hdlInstance TO FmcjPIHandleReturnValue
                 PERFORM FmcjPIName
                 DISPLAY "- name: " tInfo
                 PERFORM FmcjPIDeallocate
              END-PERFORM
              PERFORM FmcjPIVDeallocate
           END-IF

           PERFORM FmcjESLogoff.
           PERFORM FmcjESDeallocate.
           PERFORM FmcjGlobalDisconnect.
           MOVE FMC-OK TO retCode.
           GOBACK.

           COPY fmcperf.
       IDENTIFICATION DIVISION.
       PROGRAM-ID. "QUERYPI".

       DATA DIVISION.

         WORKING-STORAGE SECTION.

         COPY fmcvars.
         COPY fmcconst.
         COPY fmcrcs.

         01 localUserID   PIC X(30) VALUE z"USERID".
         01 localPassword PIC X(30) VALUE z"PASSWORD".
         01 numIList      PIC 9(9) BINARY VALUE 0.
         01 tInfo         PIC X(4097).
         01 i             PIC 9(9) BINARY VALUE 0.

         LINKAGE SECTION.

         01 retCode            PIC S9(9) BINARY.

       PROCEDURE DIVISION USING retCode.

          CALL "FmcjGlobalConnect".
      * logon
           CALL "FmcjExecutionServiceAllocate"
              USING BY REFERENCE serviceValue
              RETURNING intReturnValue.
           MOVE intReturnValue TO retCode
           IF retCode NOT = FMC-OK
              DISPLAY "Service object could not be allocated"
              DISPLAY "rc: " retCode
              MOVE -1 TO retCode
              GOBACK
           END-IF
           CALL "SETADDR" USING localUserId userId.
           CALL "SETADDR" USING localPassword passwordValue.
           CALL "FmcjExecutionServiceLogon"
              USING BY VALUE serviceValue
                             userID
                             passwordValue
                             Fmc-SM-Default
                             Fmc-SA-Reset
              RETURNING intReturnValue.

           MOVE intReturnValue TO retCode
           IF retCode NOT = FMC-OK
              DISPLAY "Logon failed - rc: " retCode
              CALL "FmcjExecutionServiceDeallocate"
                 USING BY REFERENCE serviceValue
                 RETURNING intReturnValue
              MOVE -1 TO retCode
              GOBACK
           END-IF

      * query process instances
           CALL "SETADDR" USING FmcjNoFilter filter.
           CALL "SETADDR" USING FmcjNoSortCriteria sortCriteria.
           CALL "FmcjExecutionServiceQueryProcessInstances"
              USING BY VALUE     serviceValue
                                 filter
                                 sortCriteria
                                 FmcjNoThreshold
                    BY REFERENCE instances
              RETURNING intReturnValue.

           SET hdlVector TO instances.
           MOVE intReturnValue TO retCode
           IF retCode NOT = FMC-OK
              DISPLAY "QueryProcessInstances returns: " retCode
           ELSE
              DISPLAY "QueryProcessInstances returns okay"
           END-IF

           IF retCode = FMC-OK
              CALL "FmcjProcessInstanceVectorSize"
                 USING BY VALUE hdlVector
                 RETURNING ulongReturnValue
              MOVE ulongReturnValue TO numIList
              DISPLAY "Number of instances returned: " numIList
              MOVE 4097 TO bufferLength
              CALL "SETADDR" USING tInfo instanceNameBuffer
              PERFORM VARYING i FROM 1 BY 1 UNTIL i > numIList
                 CALL "FmcjProcessInstanceVectorNextElement"
                    USING BY VALUE hdlVector
                    RETURNING FmcjPIHandleReturnValue
                 SET hdlInstance TO FmcjPIHandleReturnValue
                 CALL "FmcjProcessInstanceName"
                    USING BY VALUE hdlInstance
                                   instanceNameBuffer
                                   FMC-PROC-INST-NAME-LENGTH
                    RETURNING pointerReturnValue
                 DISPLAY "- name: " tInfo
                 CALL "FmcjProcessInstanceDeallocate"
                    USING BY REFERENCE hdlInstance
                    RETURNING intReturnValue
              END-PERFORM
              CALL "FmcjProcessInstanceVectorDeallocate"
                 USING BY REFERENCE hdlVector
                 RETURNING intReturnValue
           END-IF
```

```
CALL "FmcjExecutionServiceLogoff"
   USING BY VALUE serviceValue
   RETURNING intReturnValue.
CALL "FmcjExecutionServiceDeallocate"
   USING BY REFERENCE serviceValue
   RETURNING intReturnValue.
CALL "FmcjGlobalDisconnect".
MOVE FMC-OK TO retCode.
GOBACK.
```

Query work items from a worklist (C language)

```
#include <stdio.h>
#include <string.h>
#include <fmcjcrun.h>                 /* MQ Workflow Runtime API */

int main (int argc, char ** argv)
{
  APIRET                        rc            = FMC_OK;
  FmcjExecutionServiceHandle   service       = 0;
  FmcjWorklistVectorHandle     wLists        = 0;
  FmcjWorklistHandle           worklist      = 0;
  FmcjWorkitemVectorHandle     wVector       = 0;
  FmcjWorkitemHandle           workitem      = 0;
  unsigned long                numWList      = 0;
  char                         tInfo[4096+1] = "";

  FmcjGlobalConnect();

  /*  Logon */
  rc= FmcjExecutionServiceAllocate( &service );
  if (rc != FMC_OK)
  {
    printf("Service object could not be allocated: %u%\n",rc);
    return -1;
  }

  rc= FmcjExecutionServiceLogon( service,
                                 "USERID", "password",
                                 Fmc_SM_Default, Fmc_SA_NotSet );
  if ( rc != FMC_OK )
  {
    printf("Logon failed - rc    : %u%\n",rc);
    rc= FmcjExecutionServiceDeallocate( &service );
    return -1;
  }

  /* query worklists  */
  rc = FmcjExecutionServiceQueryWorklists( service, &wLists );
  if ( rc != FMC_OK)
     printf( "QueryWorklists() returns: %u%\n",rc );
  else
     printf( "QueryWorklists() returns okay\n" );
  if (rc == FMC_OK)
  {
    numWList= FmcjWorklistVectorSize(wLists);
    printf ("Number of worklists returned : %u\n", numWList);
    if ( numWList == 0 )
    {
      printf("No worklist found \n");
      FmcjWorklistVectorDeallocate(&wLists);
      rc= FmcjExecutionServiceDeallocate( &service );
      return -1;
    }

    worklist= FmcjWorklistVectorFirstElement(wLists);
```

```
      FmcjWorklistName( worklist, tInfo, 4097 );
      printf("Name                         : %s\n",tInfo);

    /* query workitems  */
      rc= FmcjWorklistQueryWorkitems( worklist, &wVector );
      printf("\nQuery workitems of list returns rc: %u\n",rc);

      if (rc == FMC_OK)
      {
        while ( 0 != (workitem= FmcjWorkitemVectorNextElement(wVector)) )
        {
          FmcjWorkitemName( workitem, tInfo, 4097 );
          printf("- Name                       : %s\n",tInfo);

          FmcjWorkitemDeallocate(&workitem);
        }
      }

      FmcjWorklistDeallocate(&worklist);
      FmcjWorklistVectorDeallocate(&wLists);
    }

    /*  Logoff */
    rc= FmcjExecutionServiceLogoff(service);
    rc= FmcjExecutionServiceDeallocate( &service );

    FmcjGlobalDisconnect();
    return 0;
  }
```

Query work items from a worklist (Cobol language)

```
        IDENTIFICATION DIVISION.
        PROGRAM-ID. "QUERYWI".

        DATA DIVISION.

          WORKING-STORAGE SECTION.

          COPY fmcvars.
          COPY fmcconst.
          COPY fmcrcs.

          01 localUserID   PIC X(30) VALUE z"USERID".
          01 localPassword PIC X(30) VALUE z"PASSWORD".
          01 numWList      PIC 9(9) BINARY VALUE 0.
          01 tInfo         PIC X(4097).

          LINKAGE SECTION.

          01 retCode            PIC S9(9) BINARY.

        PROCEDURE DIVISION USING retCode.

             PERFORM FmcjGlobalConnect.

       * logon
             PERFORM FmcjESAllocate.
             MOVE intReturnValue TO retCode
             IF retCode NOT = FMC-OK
                DISPLAY "Service object could not be allocated"
                DISPLAY "rc: " retCode
                MOVE -1 TO retCode
                GOBACK
             END-IF

             CALL "SETADDR" USING localUserId userId.
```

```
           CALL "SETADDR" USING localPassword passwordValue.
           MOVE Fmc-SM-Default TO sessionMode.
           MOVE Fmc-SA-Reset TO absenceIndicator.
           PERFORM FmcjESLogon.

           MOVE intReturnValue TO retCode
           IF retCode NOT = FMC-OK
              DISPLAY "Logon failed - rc: " retCode
              PERFORM FmcjESDeallocate
              MOVE -1 TO retCode
              GOBACK
           END-IF

      * query worklists
           PERFORM FmcjESQueryWorklists.
           MOVE intReturnValue TO retCode
           IF retCode NOT = FMC-OK
              DISPLAY "QueryWorklists returns - rc: " retCode
           ELSE
              DISPLAY "QueryWorklists returns okay"
           END-IF

           IF retCode = FMC-OK
              SET hdlVector TO lists
              PERFORM FmcjWLVectorSize
              MOVE ulongReturnValue TO numWList
              DISPLAY "Number of worklists returned : " numWList
              IF numWList = 0
                 DISPLAY "No worklist found"
                 PERFORM FmcjWLDeallocate
                 PERFORM FmcjESDeallocate
                 MOVE -1 TO retCode
                 GOBACK
              END-IF

              PERFORM FmcjWLVectorFirstElement
              SET hdlList TO FmcjWLHandleReturnValue
              MOVE 4097 TO bufferLength
              CALL "SETADDR" USING tInfo listNameBuffer
              PERFORM FmcjWLName
              DISPLAY "Name               : " tInfo

      * query workitems
              PERFORM FmcjWLQueryWorkitems
              MOVE intReturnValue TO retCode
              DISPLAY "Query workitems of list returns rc:" retCode
              SET hdlVector TO workitems
              CALL "SETADDR" USING tInfo itemNameBuffer
              IF retCode = FMC-OK
                 PERFORM FmcjWIVNextElement
                 SET hdlItem TO FmcjWIHandleReturnValue
                 PERFORM UNTIL pointerReturnValue = NULL
                    PERFORM FmcjWIName
                    DISPLAY "- Name             : " tInfo
                    PERFORM FmcjWIDeallocate
                    PERFORM FmcjWIVNextElement
                 END-PERFORM
              END-IF
              PERFORM FmcjWLDeallocate
              PERFORM FmcjWLVectorDeallocate
           END-IF

           PERFORM FmcjESLogoff.
           PERFORM FmcjESDeallocate.
           PERFORM FmcjGlobalDisconnect.
           MOVE FMC-OK TO retCode.
           GOBACK.

           COPY fmcperf.
      IDENTIFICATION DIVISION.
      PROGRAM-ID. "QUERYWI".

      DATA DIVISION.

        WORKING-STORAGE SECTION.

        COPY fmcvars.
        COPY fmcconst.
        COPY fmcrcs.

        01 localUserID   PIC X(30) VALUE z"USERID".
        01 localPassword PIC X(30) VALUE z"PASSWORD".
        01 numWList      PIC 9(9) BINARY VALUE 0.
        01 tInfo         PIC X(4097).

        LINKAGE SECTION.

        01 retCode            PIC S9(9) BINARY.

      PROCEDURE DIVISION USING retCode.

           CALL "FmcjGlobalConnect".
     * logon
           CALL "FmcjExecutionServiceAllocate"
              USING BY REFERENCE serviceValue
              RETURNING intReturnValue.
           MOVE intReturnValue TO retCode
           IF retCode NOT = FMC-OK
              DISPLAY "Service object could not be allocated"
              DISPLAY "rc: " retCode
              MOVE -1 TO retCode
              GOBACK
           END-IF

           CALL "SETADDR" USING localUserId userId.
           CALL "SETADDR" USING localPassword passwordValue.
           CALL "FmcjExecutionServiceLogon"
              USING BY VALUE serviceValue
                             userID
                             passwordValue
                             Fmc-SM-Default
                             Fmc-SA-Reset
              RETURNING intReturnValue.

           MOVE intReturnValue TO retCode
           IF retCode NOT = FMC-OK
              DISPLAY "Logon failed - rc: " retCode
              CALL "FmcjExecutionServiceDeallocate"
                 USING BY REFERENCE serviceValue
                 RETURNING intReturnValue
              MOVE -1 TO retCode
              GOBACK
           END-IF

     * query worklists
           CALL "FmcjExecutionServiceQueryWorklists"
              USING BY VALUE serviceValue
                    BY REFERENCE lists
              RETURNING intReturnValue.
           MOVE intReturnValue TO retCode
           IF retCode NOT = FMC-OK
              DISPLAY "QueryWorklists returns - rc: " retCode
```

```
           ELSE
              DISPLAY "QueryWorklists returns okay"
           END-IF

           IF retCode = FMC-OK
              SET hdlVector TO lists
              CALL "FmcjWorklistVectorSize"
                 USING BY VALUE hdlVector
                 RETURNING ulongReturnValue
              MOVE ulongReturnValue TO numWList
              DISPLAY "Number of worklists returned : " numWList
              IF numWList = 0
                 DISPLAY "No worklist found"
                 CALL "FmcjWorklistDeallocate"
                    USING BY REFERENCE hdlList
                    RETURNING intReturnValue
                 CALL "FmcjExecutionServiceDeallocate"
                    USING BY REFERENCE serviceValue
                    RETURNING intReturnValue
                 MOVE -1 TO retCode
                 GOBACK
              END-IF

              CALL "FmcjWorklistVectorFirstElement"
                 USING BY VALUE hdlVector
                 RETURNING FmcjWLHandleReturnValue
              SET hdlList TO FmcjWLHandleReturnValue
              MOVE 4097 TO bufferLength
              CALL "SETADDR" USING tInfo listNameBuffer
              CALL "FmcjPersistentListName"
                 USING BY VALUE hdlList
                                listNameBuffer
                                bufferLength
                 RETURNING pointerReturnValue
              DISPLAY "Name               : " tInfo

     * query workitems
              CALL "FmcjWorklistQueryWorkitems"
                 USING BY VALUE hdlList
                 BY REFERENCE workitems
                 RETURNING intReturnValue
              MOVE intReturnValue TO retCode
              DISPLAY "Query workitems of list returns rc:" retCode
              SET hdlVector TO workitems
              CALL "SETADDR" USING tInfo itemNameBuffer
              IF retCode = FMC-OK
                 CALL "FmcjWorkitemVectorNextElement"
                    USING BY VALUE hdlVector
                    RETURNING FmcjWIHandleReturnValue
                 SET hdlItem TO FmcjWIHandleReturnValue
                 PERFORM UNTIL pointerReturnValue = NULL
                    CALL "FmcjItemName"
                       USING BY VALUE hdlItem
                                      itemNameBuffer
                                      bufferLength
                       RETURNING pointerReturnValue
                    DISPLAY "- Name             : " tInfo
                    CALL "FmcjWorkitemDeallocate"
                       USING BY REFERENCE hdlWorkitem
                       RETURNING intReturnValue
                    CALL "FmcjWorkitemVectorNextElement"
                       USING BY VALUE hdlVector
                       RETURNING FmcjWIHandleReturnValue
                 END-PERFORM
              END-IF
              CALL "FmcjWorklistDeallocate"
                 USING BY REFERENCE hdlList
```

```
      RETURNING intReturnValue
   CALL "FmcjWorklistVectorDeallocate"
      USING BY REFERENCE hdlVector
      RETURNING intReturnValue
END-IF

    CALL "FmcjExecutionServiceLogoff"
   USING BY VALUE serviceValue
   RETURNING intReturnValue.
CALL "FmcjExecutionServiceDeallocate"
   USING BY REFERENCE serviceValue
   RETURNING intReturnValue.
CALL "FmcjGlobalDisconnect".
MOVE FMC-OK TO retCode.
GOBACK.
```

Chapter 51. An activity implementation

The following examples show the concept of how to query and set containers from within an activity implementation. Refer to the examples provided with the product for more details.

Programming an executable (C language)

```
#include <stdio.h>
#include <fmcjccon.h>                  /* MQ Workflow Container API */
int main()
{
  FILE                          *  file1            = 0;
  APIRET                           rc               = FMC_OK;
  FmcjReadOnlyContainerHandle      input            = 0;
  FmcjReadWriteContainerHandle     output           = 0;
  char                             stringBuffer[4097]="";

/*- keep results in a file ----------------------------------------------*/
  file1 = fopen ("sample.out", "a");
  if ( file1 == 0 )
    return -1;
  fprintf(file1,"\n----- C-API Activity Implementation called -----\n");
  fflush(file1);

  FmcjGlobalConnect();

/*-- retrieve the input container from the PEA who started the program --*/
  rc = FmcjContainerInContainer( &input );
  fprintf(file1, "Get Input Container - rc:  %u\n", rc);
  if (rc != FMC_OK)
  {
    fclose(file1);
    return 1;
  }

  fprintf(file1, "Input Container Name: %s\n",
          FmcjReadOnlyContainerType(input, stringBuffer, 4097));

/*-- retrieve the output container from the PEA who started the program -*/
  rc = FmcjContainerOutContainer( &output );
  fprintf(file1, "Get Output Container - rc:  %u\n", rc);
  if (rc != FMC_OK)
  {
    fclose(file1);
    return 1;
  }

  fprintf(file1, "Output Container Name: %s\n",
          FmcjReadWriteContainerType(output, stringBuffer, 4097));

/*----- Modify output values ---------------------------------------------*/
  rc= FmcjReadWriteContainerSetLongValue(output, "aFieldInTheOutput",42);
  fprintf(file1, "\nSetting long value returns rc: %u\n", rc);

  ...

/*-- return the output container to the PEA who started the program -----*/
  rc = FmcjContainerSetOutContainer( output );
  fprintf(file1, "\nSet Output Container - rc: %u\n",rc);
  fflush(file1);
  FmcjGlobalDisconnect();
  fclose(file1);
  return 0;                            // _RC passed to FlowMark
}
```

Programming an executable (Cobol language)

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. "EXEC".

       DATA DIVISION.


         WORKING-STORAGE SECTION.

         COPY fmcvars.
         COPY fmcrcs.

         01 stringBuffer  PIC X(4097).
         01 fieldName     PIC X(39) VALUE z"aFieldInTheOutput".

         LINKAGE SECTION.

         01 retCode            PIC S9(9) BINARY.

       PROCEDURE DIVISION USING retCode.

            DISPLAY "-- Cobol-API Activity Implementation called --"
            PERFORM FmcjGlobalConnect.

      * retrieve the input container
            PERFORM FmcjCInCtnr.
            MOVE intReturnValue TO retCode.
            DISPLAY "Get Input Container - rc: " retCode.
            IF retCode NOT = FMC-OK
               MOVE 1 TO retCode
               GOBACK
            END-IF

            CALL "SETADDR" USING stringBuffer containerTypeBuffer.
            MOVE 4097 TO bufferLength.
            SET hdlContainer TO inputValue.
            PERFORM FmcjROCType.
            DISPLAY "Input Container Name: " stringBuffer.

      * retrieve the output container
            PERFORM FmcjCOutCtnr.
            MOVE intReturnValue TO retCode.
            DISPLAY "Get Output Container - rc: " retCode.
            IF retCode NOT = FMC-OK
               MOVE 1 TO retCode
               GOBACK
            END-IF

            SET hdlContainer TO outputValue.
            PERFORM FmcjRWCType.
            DISPLAY "Output Container Name: " stringBuffer.

      * modify output values
            MOVE 42 TO intValue.
            CALL "SETADDR" USING fieldName qualifiedName.
            PERFORM FmcjRWCSetLongValue.
            MOVE intReturnValue TO retCode.
            DISPLAY "Setting long value returns rc: " retCode.

      * return the output container
            PERFORM FmcjCSetOutCtnr.
```

```
          MOVE intReturnValue TO retCode.
          DISPLAY "Set Output Container - rc: " retCode.

          PERFORM FmcjGlobalDisconnect.
          MOVE FMC-OK TO retCode.
          GOBACK.

          COPY fmcperf.
     IDENTIFICATION DIVISION.
     PROGRAM-ID. "EXEC".

     DATA DIVISION.

       WORKING-STORAGE SECTION.

       COPY fmcvars.
       COPY fmcrcs.

       01 stringBuffer  PIC X(4097).
       01 fieldName     PIC X(39) VALUE z"aFieldInTheOutput".

       LINKAGE SECTION.

       01 retCode            PIC S9(9) BINARY.

     PROCEDURE DIVISION USING retCode.

          DISPLAY "-- Cobol-API Activity Implementation called --"
          CALL "FmcjGlobalConnect".

    * retrieve the input container
          CALL "FmcjContainerInContainer"
             USING BY REFERENCE inputValue
             RETURNING intReturnValue.
          MOVE intReturnValue TO retCode.
          DISPLAY "Get Input Container - rc: " retCode.
          IF retCode NOT = FMC-OK
             MOVE 1 TO retCode
             GOBACK
          END-IF

          CALL "SETADDR" USING stringBuffer containerTypeBuffer.
          MOVE 4097 TO bufferLength.
          SET hdlContainer TO inputValue.
          CALL "FmcjContainerType"
             USING BY VALUE hdlContainer
                            containerTypeBuffer
                            bufferLength
             RETURNING pointerReturnValue.
          DISPLAY "Input Container Name: " stringBuffer.

    * retrieve the output container
          CALL "FmcjContainerOutContainer"
             USING BY REFERENCE outputValue
             RETURNING intReturnValue.
          MOVE intReturnValue TO retCode.
          DISPLAY "Get Output Container - rc: " retCode.
          IF retCode NOT = FMC-OK
             MOVE 1 TO retCode
             GOBACK
          END-IF

          SET hdlContainer TO outputValue.
          CALL "FmcjContainerType"
             USING BY VALUE hdlContainer
                            containerTypeBuffer
                            bufferLength
```

```
           RETURNING pointerReturnValue.
        DISPLAY "Output Container Name: " stringBuffer.

  * modify output values
        MOVE 42 TO intValue.
        CALL "SETADDR" USING fieldName qualifiedName.
        CALL "FmcjReadWriteContainerSetLongValue"
           USING BY VALUE hdlContainer
                          qualifiedName
                          intValue
           RETURNING intReturnValue.
        MOVE intReturnValue TO retCode.
        DISPLAY "Setting long value returns rc: " retCode.

  * return the output container
        CALL "FmcjContainerSetOutContainer"
           USING BY VALUE outputValue
           RETURNING intReturnValue.
        MOVE intReturnValue TO retCode.
        DISPLAY "Set Output Container - rc: " retCode.

        CALL "FmcjGlobalDisconnect".
        MOVE FMC-OK TO retCode.
        GOBACK.

        COPY fmcperf.
```

Part 9. Appendixes

Appendix. Audit Trail

When a process instance is executed, MQSeries Workflow writes information about each significant event into an audit trail. The audit trail is managed in the MQSeries Workflow relational database.

Whether an audit trail is written at all and if so, how much is written into the audit trail, is controlled by the ***AUDIT*** option of the process instance. There is an audit specification inheritance from the domain level of MQSeries Workflow through the system group to the system and down to the process template. Each specification can be overwritten on a lower level.

Note: There is a deviation from FlowMark Version 2: an audit trail will only be written for processes imported from Version 2.3 and previous, if AUDIT is set within the MQSeries Workflow Definition Language (FDL).

The events are written into the audit trail of the MQSeries Workflow system on which the process instance was started.

Process instances are identified by the process instance name or the process instance identifier. Both are written into the audit trail. Object identifiers are stored in their external character format.

Access to the audit trail can be done by applications that use standard SQL. Care must be taken to avoid any unintentional changes to the audit trail.

Each audit trail record is associated with a timestamp. This timestamp reflects the date and time the audit trail record was written. As such, it is filled by the underlying relational database management system (DB2 for OS/390 Special Register CURRENT_TIMESTAMP). Since it is not guaranteed that all timestamps are unique, the sequence in which audit trail records with the same timestamp are retrieved is random.

Table 16 shows the structure of the audit trail in the realational database:

Table 16. Audit Trail Record Layout

Field Name	**Type**	**Explanation**
Timestamp	TIMESTAMP Mandatory	Date and time the audit trail record is written.
Event	INTEGER Mandatory	Type of event as indicated in Table 17 on page 475.
Process Name	VARCHAR (63) Mandatory	Name of the process instance.
Process Identifier	IDENTIFIER Mandatory	Object identifier of the process instance.

Table 16. Audit Trail Record Layout (continued)

Field Name	Type	Explanation
Toplevel Name	VARCHAR (63) Mandatory	Name of the top-level process instance if the process instance is executing as subprocess, or the same as in process name if the process instance is a top-level process instance.
Toplevel Identifier	IDENTIFIER Mandatory	Object identifier of the top-level process instance if the process is executing as subprocess, or the same as in process identifier if the process instance is a top-level process instance.
Parent Process Name	VARCHAR (63) Optional	Name of the parent process instance if the process instance is executing as a subprocess.
Parent Process Identifier	IDENTIFIER Optional	Object identifier of the parent process instance if the process instance is executing as a sub-process.
Process Model Name	VARCHAR(32) Mandatory	Name of the process model.
Process Model Valid From Date	TIMESTAMP Optional	Contains the valid from time of the associated process model.
Block Names	VARCHAR(254) Optional	The concatenated names of all blocks in which the activity is contained in. The various names are separated by a dot.
User ID	VARCHAR(32) Optional	ID of the user associated with the event that caused the audit trail to be written. If the audit trail record is written by the MQSeries Workflow system, this field is not filled.
Second user ID	VARCHAR(32) Optional	ID of the second user associated with the event that caused the audit trail to be written.

Table 16. Audit Trail Record Layout (continued)

Field Name	Type	Explanation
Activity Name	VARCHAR(32) Optional	If the audit trail entry is associated with an activity, the field contains the name of the activity. If the audit trail entry is associated with a control connector, the field contains the name of the activity that is the source of the control connector.
Activity Type	INTEGER Optional	If the audit trail record is written for an activity, the field contains the type of the activity as defined in Table 18 on page 476.
Activity Status	INTEGER Optional	If the audit trail record is written for an event associated with an activity, the field contains the status of the activity encoded as shown in Table 19 on page 477.
Second Activity Name	VARCHAR(32) Optional	If the audit trail is written for an event associated with a control connector, the field contains the name of the target activity.
Command Parameters	VARCHAR(1024) Optional	If the event is the start of a program activity, the field contains the actual parameters passed when invoking the program.
Associated object	IDENTIFIER Optional	Contains the identifier of the object associated with the event. Can be used to locate the object in the MQSeries Workflow database.
Object description	VARCHAR(254) Optional	Contains the description of the object associated with the event.
Program Name	VARCHAR(32) Optional	If the event is the start of a program activity, the field contains the name of the program.

Table 16. Audit Trail Record Layout (continued)

Field Name	**Type**	**Explanation**
Activity Return Code	LONG Optional	Return code of the activity.

The contents of each audit trail record depends on the event. Table 17 on page 475 shows the contents of each field.

The audit trail level field indicates, whether the audit trail is written for a particular level. If full audit trailing is active, all audit trail records are written. The following code is used:

- C Condensed audit trail

Table 17. Audit Trail Record Contents

Code	Audit Trail Level	User Id	Second User Id	Associated object	Event
21000	C	Process starter		Process instance	Process started
21001		Issuer of suspend command		Process instance	Process suspended
21002		Issuer of resume command		Process instance	Process resumed
21003		Target of notification		Process instance	Process notification sent
21004	C			Process instance	Process ended normally
21005	C			Process instance	Process terminated
21006	C			Activity instance	Activity ready
21007	C	User on whose behalf the activity is started		Activity instance	Activity started
21008		Target of notification		Activity instance	First activity notification sent
21009		Target of transfer	Source of transfer	Work item	Work item transferred
21010		Issuer of command	User for whom work item is created	Work item	Work item created
21011	C	User on whose behalf the activity was executed		Activity instance	Activity ended normally
21012	C	Issuer of force-finish command		Activity instance	Activity force-finished
21013		Issuer of restart command		Activity instance	Activity restarted
21014	C	Issuer of finish command		Activity instance	Activity exited manually
21015					Block started
21016					Block ended
21017		Issuer of create command		Process instance	Process created
21018	C	Issuer of create and start command		Process instance	Process created and started
21020		Issuer of delete command		Process instance	Process deleted
21022	C	Issuer of checkout command		Activity instance	Checkout activity

Table 17. Audit Trail Record Contents (continued)

Code	Audit Trail Level	User Id	Second User Id	Associated object	Event
21023		Issuer of checkin command		Activity instance	Checkin activity
21024		Target of notification		Activity instance	Second notification for activity sent
21025	C			Process instance	Process ended normally and deleted
21026	C	Issuer of terminate command		Process instance	Process terminated and deleted
21027	C	Issuer of terminate command		Activity Instance	Activity terminated
21030		Issuer of delete work item command		Work item	Work item deleted
21031	C	Issuer of force restart work item command		Activity instance	Activity force restarted
21032		User on whose behalf the activity was executed.		Activity instance	Activity implementation completed
21034					Control connector evaluated to true
21037	C	Issuer of suspend command		Process instance	The specified user has issued a suspend process command.
21038	C	Issuer of terminate process command		Process instance	The specified user has issued a terminate process command.
21040	C	Issuer of resume command		Process instance	The specified user has issued a resume process command.
21041		User who has processed the activity		Activity instance	Activity automatically restarted as exit condition evaluated to false.
21056				Process instance	Block ended and loop back to the beginning because the exit condition failed.

The following table shows the encoding for activity types.

Table 18. Audit Trail Activity Type Encoding

Code	Activity Type
21100	Program activity
21101	Process activity

Table 18. Audit Trail Activity Type Encoding (continued)

Code	Activity Type
21102	Block activity

The following table shows the encoding for activity states. If an event is associated with a state change, the target state is recorded in the audit trail record.

Table 19. Audit Trail Activity State Encoding

Code	Activity State
21200	Ready
21201	Running
21202	Finished
21203	CheckedOut
21204	Force-Finished
21205	Terminated
21206	Suspended
21207	InError
21208	Executed
21209	Skipped
21210	Deleted
21211	Suspending
21212	Terminating

Glossary

This glossary defines important terms and abbreviations used in this publication. If you do not find the term you are looking for, refer to the index or the *IBM Dictionary of Computing*, New York: McGraw-Hill, 1994.

A

administration server. The MQ Workflow component that performs administration functions within an MQ Workflow system. Functions include starting and stopping of the MQ Workflow system, performing error management, and participating in administrative functions for a system group.

activity. One of the steps that make up a process model. This can be a program activity, process activity, or block activity.

activity information member. A predefined data structure member associated with the operating characteristics of an activity.

API. Application Programming Interface.

application programming interface. An interface provided by the MQ Workflow workflow manager that enables programs to request services from the MQ Workflow workflow manager. The services are provided synchronously.

audit trail. A relational table in the database that contains an entry for each major event during execution of a process instance.

authorization. The attributes of a user's staff definition that determine the user's level of authority in MQ Workflow. The system administrator is allowed to perform all functions.

B

backward mapping. Conversion of output data created by an OS/390 legacy application into an MQSeries Workflow container. This conversion is performed by the *program execution server's program mapper*.

backward mapping definition. Part of the *MDL* which connects an interface definition and structure definition.

bend point. A point at which a connector starts, ends, or changes direction.

block activity. A composite activity that consists of a group of activities, which can be connected with control and data connectors. A block activity is used to implement a Do-Until loop; all activities within the block activity are processed until the exit condition of the block activity evaluates to true. See also *composite activity.*

Buildtime. An MQ Workflow component with a graphical user interface for creating and maintaining workflow models, administering resources, and the system network definitions.

C

cardinality. (1) An attribute of a relationship that describes the membership quantity. There are four types of cardinality: One-to-one, one-to-many, many-to-many, and many-to-one. (2) The number of rows in a database table or the number of different values in a column of a database table.

child organization. An organization within the hierarchy of administrative units of an enterprise that has a parent organization. Each child organization can have one parent organization and several child organizations. The parent is one level above in the hierarchy. Contrast with *parent organization*.

cleanup server. The MQ Workflow component that physically deletes information in the MQ Workflow Runtime database, which had only been deleted logically.

composite activity. An activity which is composed of other activities. Composite activities are block activities and bundle activities.

container API. An MQ Workflow API that allows programs executing under the control of MQ Workflow to obtain data from the input and output container of the activity and to store data in the output container of the activity.

control connector. Defines the potential flow of control between two nodes in the process. The actual flow of control is determined at run time based on the truth value of the transition conditions associated with the control connector.

coordinator. A predefined role that is automatically assigned to the person designated to coordinate a role.

CPIC. An invocation type that allows the programe xecution server to run an application synchronously on an IMS service. CPIC is based on IMS/APPC.

D

data connector. Defines the flow of data between containers.

data container. Storage for the input and output data of an activity or process. See *input container* and *output container*.

data mapping. Specifies, for a data connector, which fields from the associated source container are mapped to which fields in the associated target container.

data structure. A named entity that consists of a set of data structure members. Input and output containers are defined by reference to a data structure and adopt the layout of the referenced data structure type.

data structure member. One of the variables of which a data structure is composed.

default control connector. The graphical representation of a standard control connector, shown in the process diagram. Control flows along this connector if no other control path is valid.

domain. A set of MQ Workflow system groups which have the same meta-model, share the same staff information, and topology information. Communication between the components in the domain is via message queuing.

dynamic staff assignment. A method of assigning staff to an activity by specifying criteria such as role, organization, or level. When an activity is ready, the users who meet the selection criteria receive the activity to be worked on. See also *level*, *organization*, *process administrator*, and *role*.

E

end activity. An activity that has no outgoing control connector.

EXCI. An invocation type that allows the programe xecution server to run an application synchronously on an CICS service. EXCI is based on the CICS External CICS Interface provided by CICS Version 4.1 and higher to allow non-CICS applications to call programs running under CICS.

execution server. The MQ Workflow component that performs the processing of process instances at runtime.

exit condition. A logical expression that specifies whether an activity is complete.

export. An MQ Workflow utility program for retrieving information from the MQ Workflow database and making it available in MQ Workflow Definition Language (FDL) or HTML format. Contrast with *import*.

F

fixed member. A predefined data structure member that provides information about the current activity. The value of a fixed member is set by the MQ Workflow workflow manager.

(FDL) MQ Workflow Definition Language. The language used to exchange MQ Workflow information between MQ Workflow system groups. The language is used by the import and export function of MQ Workflow and contains the workflow definitions for staff, programs, data structures, and topology. This allows non-MQ Workflow components to interact with MQ Workflow. See also *export* and *import*.

fork activity. An activity that is the source of multiple control connectors.

form. In Lotus Notes, a form controls how you enter information into Lotus Notes and how that information is displayed and printed.

formula. In Lotus Notes, a mathematical expression that is used, for example, to select documents from a database or to calculate values for display.

forward mapping. Conversion of MQSeries Workflow containers into a format accepted by an OS/390 legacy application. This conversion is performed by the *program execution server's program mapper*.

forward mapping definition. Part of the *MDL* which connects an structure definition and interface definition.

fully-qualified name. A qualified name that is complete; that is, one that includes all names in the hierarchical sequence above the structure member to which the name refers, as well as the name of the member itself.

I

import. An MQ Workflow utility program that accepts information in the MQ Workflow definition language (FDL) format and places it in an MQ Workflow database. Contrast with *export*.

input container. Storage for data used as input to an activity or process. See also *source* and *data mapping*.

interface. The definition of the data structure accepted by an OS/390 CICS or IMS legacy application. This definition is used by the *program mapper* to convert data to (and from) an MQSeries Workflow program's *structure*.

interface definition. Part of the *MDL* which defines the interface used by a legacy application.

interface element. Part of an *interface* definition. An interface element has a name, a type and a cardinality. It is mapped on to a *structure* element by a *mapping rule*.

invocation exit. The Dll specified by the invocation type. The exit is based in an invocation protocol like CICS External CICS Interface, IMS/APPC or the MQSeries CICS and IMS bridges.

invocation protocol. The way the PES connects to a service like CICS or IMS in order to invoke a program on that service.

invocation type. The way the program execution server connects to a service system (like CICS or IMS) in order to invoke a program on that service. The invocation type is part of a program mapping execution request sent to the PES. To invoke a program, the PES loads the appropiate invocation exit as defined for the invocation type. Invocation types include *EXCI* and *CPIC*.

L

level. A number from 0 through 9 that is assigned to each person in an MQ Workflow database. The person who defines staff in Buildtime can assign a meaning to these numbers such as rank and experience. Level is one of the criteria that can be used to dynamically assign activities to people.

local user. Identifies a user during staff resolution whose home server is in the same system group as the originating process.

local subprocess. A subprocess that is processed in the same MQ Workflow system group as the originating process.

logical expression. An expression composed of operators and operands that, when evaluated, gives a result of true, false, or an integer. (Nonzero integers are equivalent to false.) See also *exit condition* and *transition condition*.

M

manager. A predefined role that is automatically assigned to the person who is defined as head of an organization.

mapping definition language. The language used to define *mapping definitions* for the *programm mapping exit*.

mapping exit. Used by the PES to convert data between MQSeries Workflow and legacy applications. The exit is defined by a mpping type defined in the *PES directory* and in *Buildtime*. The exit is only called if mapping has been enabled in *Buildtime*.

mapping rules. Part of a *forward mapping* or *backward mapping* definition that defines the mapping between individual *interface elements* and *structure elements*. Mapping rules are defined using the *mapper definition language*.

mapping type. The name used to identify which mapping exit to use. The mapping type is defined in the *PES directory* and must match the *Buildtime* definitions for the legacy application. The mapping type provided with MQSeries Workflow for OS/390 is named **DEFAULT**.

MDL. See *mapping definition language*.

message queuing. A communication technique that uses asynchronous messages for communication between software components.

MQCICS. An invocation type that allows the program execution server to run an application asynchronously on a CICS service. The corresponding invocation exit uses the MQSeries CICS Bridge as invocation protocol.

MQIMS. An invocation type that allows the program execution server to run an application asynchronously on an IMS service. The corresponding invocation exit uses the invocation protocol MQSeries IMS Bridge.

N

navigation. Movement from a completed activity to subsequent activities in a process. The paths followed are determined by control connectors, their associated transition conditions, and by the start conditions of activities. See also *control connector*, *exit condition*, *transition condition*, and *start condition*.

node. (1) The generic name for activities within a process diagram. (2) The operating system image that hosts MQ Workflow systems.

notification. An MQ Workflow facility that can notify a designated person when a process or activity is not completed within the specified time.

notification work item. A work item that represents an activity or process notification.

O

organization. An administrative unit of an enterprise. Organization is one of the criteria that can be used to dynamically assign activities to people. See *child organization* and *parent organization*.

output container. Storage for data produced by an activity or process for use by other activities or for evaluation of conditions. See also *sink*.

P

parent organization. An organization within the hierarchy of administrative units of an enterprise that has one or more child organizations. A child is one level below its parent in the hierarchy. Contrast with child *child organization*.

parent process. A process instance that contains the process activity which started the process as a subprocess.

pattern activity. A single and simple activity in a bundle activity from which multiple instances, called pattern activity instances, are created at run time.

person (pl. people). A member of staff in an enterprise who has been defined in the MQ Workflow database.

PES. See *program execution server*.

PES directory. See *program execution server directory*.

predefined data structure member. A data structure member predefined by MQ Workflow and used for communication between user applications and MQ Workflow Runtime.

process. Synonymously used for a process model and a process instance. The actual meaning is typically derived from the context.

process activity. An activity that is part of a process model. When a process activity is executed, an instance of the process model is created and executed.

process administrator. A person who is the administrator for a particular process instance. The administrator is authorized to perform all operations on a process instance. The administrator is also the target for staff resolution and notification.

process category. An attribute that a process modeler can specify for a process model to limit the set of users who are authorized to perform functions on the appropriate process instances.

process definition. Synonym for *process model*.

process diagram. A graphical representation of a process that shows the properties of a process model.

process instance. An instance of a process to be executed in MQ Workflow Runtime.

process instance list. A set of process instances that are selected and sorted according to user-defined criteria.

process instance monitor. An MQ Workflow client component that shows the state of a particular process instance graphically.

process management. The MQ Workflow Runtime tasks associated with process instances. These consist of creating, starting, suspending, resuming, terminating, restarting, and deleting process instances.

process model. A set of processes represented in a process model. The processes are represented in graphical form in the process diagram. The process model contains the definitions for staff, programs, and data structures associated with the activities of the process. After having translated the process model into a process template, the process template can be executed over and over again. *Workflow model* and *process definition* are synonyms.

process monitor API. An application programming interface that allows applications to implement the functions of a process instance monitor.

process-relevant data. Data that is used to control the sequence of activities in a process instance.

process status. The status of a process instance.

process template. A fixed form of a process model from which process instances can be created. It is the translated form in MQ Workflow Runtime. See also *process instance*.

process template list. A set of process templates that have been selected and sorted according to user-defined criteria.

program. A computer-based application that serves as the implementation of a program activity or as a support tool. Program activities reference executable programs using the logical names associated with the programs in MQ Workflow program registrations. See also *program registration*.

program activity. An activity that is executed by a registered program. Starting this activity invokes the program. Contrast with *process activity*.

program execution agent. The MQ Workflow component that manages the implementations of program activities on LAN platforms.

program execution server. The MQ Workflow component that manages the implementations of program activities on OS/390, such as CICS and IMS programs.

program mapping. Program mapping definitions passed and supported into the mapping database and used by the program mapper at runtime to transform data from legacy applications.

program mapping DB. Database used by the PES exit which contains program mappings imported by the program mapping import tool. Used at runtime by the exit to perform the forward and backward mapping.

program mapping exit. PES exit used to transform MQSeries Workflow for OS/390 containers into a format acceptable by legacy applications and vice versa.

program mapping import tool. Component of the MQSeries Workflow program mapping exit which reads the result of the program mapping parser and inputs the compiled program mapping definitions into the program mapping DB.

program mapping parser. Component of the MQSeries Workflow for OS/390 program mapping exit which parses the MDL and creates an intermediate file which is used by the program mapping import tool.

program registration. Registering a program in MQ Workflow so that sufficient information is available for managing the program when it is executed by MQ Workflow.

R

role. A responsibility that is defined for staff members. Role is one of the criteria that can be used to dynamically assign activities to people.

S

scheduling server. The MQ Workflow component that schedules actions based on time events, such as resuming suspended work items, or detecting overdue processes.

server. The servers that make up an MQ Workflow system are called Execution Server, Administration Server, Scheduling Server, and Cleanup Server.

sink. The symbol that represents the output container of a process or a block activity.

source. The symbol that represents the input container of a process or a block activity.

specific resource assignment. A method of assigning resources to processes or activities by specifying their user IDs.

standard client. The MQ Workflow component, which enables creation and control of process instances, working with worklists and work items, and manipulation of personal data of the logged-on user.

start activity. An activity that has no incoming control connector.

start condition. The condition that determines whether an activity with incoming control connectors can start after all of the incoming control connectors are evaluated.

structure. The definition of the MQSeries Workflow structure passed into or out of an *activity* implementation.

structure definition. Part of the *MDL* which defines the structure used by a program activity.

structure element. Part of an *structure* definition. A structure element has a name, a type and a cardinality. It is mapped on to a *interface* element by a *mapping rule*.

subprocess. A process instance that is started by a process activity.

substitute. The person to whom an activity is automatically transferred when the person to whom the activity was originally assigned is declared as absent.

support tool. A program that end users can start from their worklists in the MQ Workflow MQ Workflow Client to help complete an activity.

symbolic reference. A reference to a specific data item, the process name, or activity name in the description text of activities or in the command-line parameters of program registrations. Symbolic references are expressed as pairs of percent signs (%) that enclose the fully-qualified name of a data item, or either of the keywords _PROCESS or _ACTIVITY.

system. The smallest MQ Workflow unit within an MQ Workflow domain. It consists of a set of the MQ Workflow servers.

system group. A set of MQ Workflow systems that share the same database.

system administrator. (1) A predefined role that conveys all authorizations and that can be assigned to exactly one person in an MQ Workflow system. (2) The person at a computer installation who designs, controls, and manages the use of the computer system.

T

top-level process. A process instance that is not a subprocess and is started from a user's process instance list or from an application program.

transition condition. A logical expression associated with a conditional control connector. If specified, it must be true for control to flow along the associated control connector. See also *control connector*.

translate. The action that converts a process model into a Runtime process template.

U

user ID. An alphanumeric string that uniquely identifies an MQ Workflow user.

V

user type definition. Part of the *MDL* which defines the interface used by a user type.

user type interface. A user defined interface type. If you need to map a data type that is not supported by the default mapper type, you can define a user type, and write a type conversion program which handles the conversion of the particular data type. This must use the user type exit.

verify. The action that checks a process model for completeness.

W

workflow. The sequence of activities performed in accordance with the business processes of an enterprise.

Workflow Management Coalition (WfMC). A non-profit organization of vendors and users of workflow management systems. The Coalition's mission is to promote workflow standards for workflow management systems to allow interoperability between different implementations.

workflow model. Synonym for *process model*.

work item. Representation of work to be done in the context of an activity in a process instance.

work item set of a user. All work items assigned to a user.

worklist. A list of work items and notifications assigned to a user and retrieved from a workflow management system.

worklist view. List of work items selected from a work item set of a user according to filter criteria which are an attribute of a worklist. It can be sorted according to sort criteria if specified for this worklist.

Bibliography

To order any of the following publications, contact your IBM representative or IBM branch office.

MQSeries Workflow for OS/390 publications

This section lists the publications included in the MQSeries Workflow for OS/390 library.

- *IBM MQSeries Workflow for OS/390: Customization and Administration*, SC33-7030-00, explains how to customize and administer an MQ Workflow system.
- *IBM MQSeries Workflow for OS/390: Programming Guide*, SC33-7031-00, explains the C and COBOL application programming interface (APIs) and the program exits.
- *IBM MQSeries Workflow for OS/390: Messages*, SC33-7032-00, explains the MQSeries Workflow for OS/390 system messages.
- *IBM MQSeries Workflow for OS/390: Program Directory*, GI10-0483, explains how to install MQSeries Workflow for OS/390.

MQSeries Workflow publications

This section lists the publications included in the MQSeries Workflow library.

- *IBM MQSeries Workflow: List of Workstation Server Processor Groups*, GH12-6357, lists the processor groups for MQSeries Workflow.
- *IBM MQSeries Workflow: Concepts and Architecture*, GH12-6285, explains the basic concepts of MQ Workflow. It also describes the architecture of MQ Workflow and how the components fit together.
- *IBM MQSeries Workflow: Getting Started with Buildtime*, SH12-6286, describes how to use Buildtime of MQ Workflow.
- *IBM MQSeries Workflow: Getting Started with Runtime*, SH12-6287, describes how to get started with the MQ Workflow Client.
- *IBM MQSeries Workflow: Programming Guide*, SH12-6291, explains the application programming interfaces (APIs).
- *IBM MQSeries Workflow: Installation Guide*, SH12-6288, contains information and procedures for installing and customizing MQ Workflow.
- *IBM MQSeries Workflow: Administration Guide*, SH12-6289, explains how to administer an MQ Workflow system.

Related publications

- *IBM Systems Journal, Vol. 36. No. 1, 1997 by Frank Leymann, Dieter Roller*— Also availbale at: http://www.almaden.ibm.com/journal/sj/361/leymann.html
- *Workflow Handbook 1997* published in association with WfMC. Edited by Peter Lawrence.

Note: The licensed books that were declassified in OS/390 Version 2 Release 4 appear on the OS/390 Online Library Collection, SK2T-6700. The remaining licensed books for OS/390 Version 2 appear on the OS/390 Licensed Product library, LK2T-2499, in unencrypted form.

Index

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

IBM MQSeries Workflow for OS/390
Programming Guide
Version 3 Release 1

Publication No. SC33-7031-00

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

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

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

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

Please tell us how we can improve this book:

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

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name	Address
Company or Organization	
Phone No.	

Readers' Comments — We'd Like to Hear from You
SC33-7031-00

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Deutschland Entwicklung GmbH
Information Development
Department 3248
Schoenaicher Strasse 220
71032 Boeblingen
Germany

Fold and Tape **Please do not staple** Fold and Tape

SC33-7031-00

Cut or Fold
Along Line

Program Number: 5655–A96

Printed in Denmark by IBM Danmark A/S

SC33-7031-00