

IBM FlowMark



Application Integration Guide

Version 2 Release 3

IBM FlowMark



Application Integration Guide

Version 2 Release 3

Note!

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

Second Edition, December 1996

This is a major revision of, and obsoletes, SH12-6267-00.

This edition applies to Version 2 Release 3 of IBM FlowMark (5697-216) and to all subsequent releases and modifications until otherwise indicated in new editions or technical newsletters.

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

IBM welcomes your comments. A form for readers' comments may be provided at the back of this publication, or you may address your comments to the following address:

IBM Deutschland Entwicklung GmbH
Information Development, Dept. 0446
Postfach 1380
71003 Boeblingen
Germany

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.

© **Copyright International Business Machines Corporation 1996. All rights reserved.**

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	ix
Trademarks and service marks	ix
About this book	xi
Who should read this book	xi
How this book is organized	xi
Conventions used in this book	xii
Chapter 1. Integration overview	1
Integration concepts	1
About Service Broker Manager	2
About service brokers	2
About services and service functions	4
About service requesters	4
How these pieces fit together	5
Chapter 2. Using the Service Broker Manager for OS/2	7
Service Broker Manager modes	7
Starting Service Broker Manager	8
Stopping Service Broker Manager	9
Defining the Service Broker Manager resources	10
Displaying brokers	10
Displaying different views of brokers	10
Changing the details view	12
Refreshing the display of brokers	12
Viewing the monitor	13
Managing brokers	15
Displaying settings for a broker	16
Registering a new broker	16
Deleting a broker	20
Loading and unloading a broker	20
Starting and stopping a broker	21
Displaying services for a broker	22
Displaying different views of services	23
Refreshing the display of services	24
Managing broker services	25
Displaying settings for a service	26
Registering a new service	26
Deleting a service	29
Loading and unloading a service	30
Starting and stopping a service	30
Dynamic start of services	31
Chapter 3. FlowMark service broker	33
FlowMark service broker library	33

FlowMark requester	34
Using the FlowMark requester	34
FlowMark service functions	35
Starting a process instance	36
Suspending a running process instance	37
Resuming a suspended process instance	37
Terminating a running process instance	38
Restarting a process instance	39
Changing the status of an activity	39
Sample service requester calling function FMRequest()	40
FlowMark service broker demo	42
Chapter 4. FlowMark—Lotus Notes interface	43
Lotus Notes service broker library	43
Lotus Notes broker	44
Lotus Notes requester	44
Using the Lotus Notes requester	44
Lotus Notes service functions	45
Type mapping	50
Chapter 5. FlowMark—VisualAge integration	51
Installing the VisualAge source code	51
VisualAge parts for the service broker	51
VisualAge parts for the FlowMark C language API	52
VisualAge broker programming examples	53
Service broker library	55
Developing VisualAge applications for FlowMark	56
Registering a VisualAge application	56
Requirements for VisualAge applications	58
Accessing the FlowMark C container API	60
Accessing the FlowMark C process API	62
Testing your VisualAge application	63
Parts provided with the package	64
Function Data	64
FmBrokerTemplate	66
FmBrokerWindow	67
FmFunctionData	68
Current Activity	68
Input Container	73
Output Container	75
FlowMark Session	78
FlowMark Activity	79
FlowMark Process	82
FmApiLibrary	86
FmError	88
FmContainer	89
FmInputContainer	90
FmOutputContainer	91

FmContainerItem	92
FmStartDataItem	93
FmExmApiBegin	94
FmExmApiTypeInfo	95
FmExmApiStructureData	96
FmMaintainProcess	97
FmDisplayActivity	98
FmMaintainContainer	99
FmMaintainData	100
Chapter 6. Creating your own service brokers	101
Designing service brokers and services	101
C language conventions	101
Implementing a service broker	102
Implementation	102
Description of functions	103
Using the C language service broker API	110
Implementing a service	116
Implementation	116
Description of functions	118
Implementing a service requester	125
Using the C language service requester API	126
Using the REXX language service requester API	136
Using the standard service requester	144
Using the standard external controller	145
Debugging service brokers and services	145
Sample files	146
Sample service broker DLL (SAMPBROK.C)	146
Sample service DLL (SAMPSEV.C)	148
Sample C language service requester (SAMPREQ.C)	150
Sample REXX language service requester (EXMP3SRX.CMD)	152
Chapter 7. The Service Broker Manager for Windows	155
About the Service Broker Manager for Windows	155
Starting the Service Broker Manager for Windows	156
Implementing a service broker on Windows	157
Broker_GetDllVersion function	158
Broker_GetVersion	158
Broker_Init	159
Broker_Exit	160
Broker_Logon	161
Broker_Logoff	162
LibMain (for Windows 3.1)	163
Building the service broker DLL	164
Implementing a service on Windows	165
Service_CheckBroker	167
Service_Init	168
Service_Exit	169

Service_Start	169
Service_Stop	170
LibMain (for Windows 3.1)	171
Implementing a service function	172
Building the service DLL	176
Implementing a service requester	177
Calling a service function	177
Sample service requester for Windows 3.1	179
Using the standard service requester on Windows 3.1	180
Using the FlowMark requester	180
Chapter 8. Building block for MQSeries support	181
Restrictions	181
MQSeries definitions	181
Customizing the MQSeries definitions	182
Setting up FlowMark	186
Preparing the sample processes	186
Using the sample processes	186
Sample scenarios	187
Starting FlowMark for MVS/ESA from FlowMark on OS/2 or AIX	188
Controlling FlowMark for MVS/ESA from FlowMark on OS/2 or AIX	190
Starting FlowMark on OS/2 or AIX from FlowMark for MVS/ESA	191
Controlling FlowMark on OS/2 or AIX from FlowMark for MVS/ESA	193
Starting FlowMark on OS/2 or AIX from FlowMark on OS/2 or AIX	194
Controlling FlowMark on OS/2 or AIX from FlowMark on OS/2 or AIX	196
General considerations	197
Unique process-instance ID	197
Process tracking and alert events	197
Predefined data structures	197
Sample FDL	198
Sample MQSeries definitions	198
EXMP2ASD	198
EXMP2ASD return data	199
EXMP2ASD message handling	200
EXMP2ASP	200
EXMP2ASP return data	200
EXMP2ARM	200
EXMP2ARM return data	201
EXMP2ARM message handling	201
EXMP2ARV	201
EXMP2ARV return data	202
EXMP2ARV message handling	202
EXMP2ASV	202
EXMP2ASV return data	203
EXMP2ASV message handling	203
Chapter 9. Building block for AS/400 support	205
MQSeries definitions for AS/400 support	205

Customizing the MQSeries definitions for AS/400 access	206
Customizing the MQSeries definitions for controlling FlowMark processes from the AS/400	211
Accessing AS/400 applications from FlowMark processes	211
Sample scenario for AS/400 access	213
General considerations	215
EXMP24SD	215
EXMP24RC	216
FlowMark Program Access	217
Controlling FlowMark processes from AS/400 applications	218
EXMP24SV	219
Send FlowMark request (SNDFLMRQS)	220
Appendix A. Ways to integrate FlowMark and Lotus Notes	223
Overview	223
FlowMark service broker	223
Lotus Notes service broker	223
Runtime client for Lotus Notes	224
Tips for selecting the appropriate component	224
Scenario 1: Two organizations, one using FlowMark, the other Lotus Notes	224
Scenario 2: Two organizations using FlowMark and Lotus Notes	225
Scenario 3: One organization using FlowMark and Lotus Notes	225
Appendix B. Migrating from a previous version of the Service Broker Manager	227
Migrating the Service Broker Manager on OS/2	227
Migrating the Service Broker Manager on Windows 3.1	228
Glossary	231
Bibliography	237
FlowMark publications	237
Related publications	237
Index	239

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood NY 10594, U.S.A.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Deutschland Informationssysteme GmbH, Department 3982, Pascalstrasse 100, 70569 Stuttgart, Germany. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Trademarks and service marks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AIX	AS/400
C Set ++	DB2
FlowMark	IBM
MQ	MQSeries
MVS	MVS/ESA
OS/2	OS/400
Presentation Manager	VisualAge
VisualInfo	

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

C-bus is a trademark of Corollary, Inc.

Microsoft, Windows, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

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

About this book

This book describes how you can use the Service Broker Manager and the service brokers to integrate specific software products with IBM FlowMark. It also contains information about the MQSeries-support building block and the building block for AS/400 support.

Who should read this book

This book is for programmers and modelers who build solutions involving several different products by using the integration capabilities of FlowMark.

It is assumed that readers are familiar with AIX, OS/2, OS/400, Windows 3.1, and with the concepts of workflow management systems and FlowMark.

To perform the tasks described in Chapter 6, "Creating your own service brokers" on page 101 and in the sections about implementing service brokers and services in Chapter 7, "The Service Broker Manager for Windows" on page 155, programming knowledge is required.

To perform the tasks described in Chapter 8, "Building block for MQSeries support" on page 181 and Chapter 9, "Building block for AS/400 support" on page 205, you need to have a good knowledge of MQSeries in general. You also need to understand the MQSeries product for the specific platforms you will be using, and the communications for those platforms.

How this book is organized

This book contains the following chapters:

- Chapter 1, "Integration overview" on page 1 describes service broker concepts and considerations.
- Chapter 2, "Using the Service Broker Manager for OS/2" on page 7 describes how to use the Service Broker Manager, service brokers, and services under OS/2.
- Chapter 3, "FlowMark service broker" on page 33 describes the functions that are included in the FlowMark service broker.
- Chapter 4, "FlowMark—Lotus Notes interface" on page 43 describes how to control FlowMark processes from Lotus Notes and how to retrieve data stored in a FlowMark container, use the data to fill in a Lotus Notes form, and then store the form in a Lotus Notes database.
- Chapter 5, "FlowMark—VisualAge integration" on page 51 describes how to integrate VisualAge applications with FlowMark.
- Chapter 6, "Creating your own service brokers" on page 101 describes how to design and implement own OS/2 service brokers and services.

- Chapter 7, “The Service Broker Manager for Windows” on page 155 describes how to use the Service Broker Manager and how to implement your own service brokers under Windows 3.1.
- Chapter 8, “Building block for MQSeries support” on page 181 describes how to start and control a FlowMark process remotely.
- Chapter 9, “Building block for AS/400 support” on page 205 describes how to use this building block to access AS/400 applications from FlowMark processes and to control FlowMark processes from AS/400 applications.
- Appendix A, “Ways to integrate FlowMark and Lotus Notes” on page 223 gives some guidance on selecting the appropriate way to integrate Lotus Notes with FlowMark.
- Appendix B, “Migrating from a previous version of the Service Broker Manager” on page 227 describes the steps necessary for migration.

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

Conventions used in this book

The following typographical conventions are used throughout this book:

Element	Examples
Book titles are shown in italics.	<i>Installation and Maintenance</i>
Buttons, fields, and menu choices in the graphical interface are shown in boldface.	Path and file name, Audit trail
File and path names that are case-sensitive are shown in their appropriate case, otherwise they are shown in capitals.	Profile, /usr/lpp/exm, CONFIG.SYS, D:\EXM\API
Program and message variables are shown in italics.	<i>filename, linenumber</i>
Names of variables whose values you are to supply are shown in an italic font.	... and passes <i>myprog</i> as a parameter.
Commands and text you are to enter are shown in boldface. Commands that are case-sensitive are shown in their appropriate case.	Enter install to start the installation program.
Field text, sample code, event names, and data structures are shown in a monospace font.	Include the protocol definition, for example, COMPROTOCOL = 03NETTCP.

You interpret the syntax diagrams by following the arrows from left to right. The conventions are:

	<p>A set of alternatives, one of which you must code.</p>
	<p>A set of alternatives, any or none of which you may code.</p>
	<p>Items shown on branches above the main path are default values.</p>
	<p>Repeatable operand. The separator is shown on the branch above the path.</p>
<p>►► A Fragment ◄◄</p> <p>A Fragment: — <i>KEYWORD=value</i> — </p>	<p>Syntax diagrams can be broken into fragments. A fragment is indicated by vertical bars with the name of the fragment between the bars.</p>
<p>Items whose literal value varies (shown in italics)</p>	<p>Replace with the appropriate literal value.</p>
<p>Keywords, separators, operators, and delimiters</p>	<p>Code exactly as shown.</p>

Chapter 1. Integration overview

This chapter explains the various concepts for integrating software products with the IBM FlowMark workflow manager.

Integration concepts

FlowMark is an open workflow manager that enables you to use the application that is most appropriate for your business solution. To provide the flow control for applications through tools and people, it is essential to integrate the tools with FlowMark.

FlowMark includes several integration facilities:

- A programming interface—C, C++, REXX, and COBOL for OS/2, C and REXX for AIX, and C, C++, and VisualBasic for Windows 3.1 and Windows NT—to control processes and to access data containers
- An interface for exporting a process definition into a text file and importing a process definition from a text file
- Program registration capabilities that enable the invocation of executable files
- The service broker concept

Companion products can be integrated with FlowMark in the following ways:

- The service broker concept is designed to allow users of workflow systems or other applications to work with multiple tools in multiple interactions without the need to reload the tool each time or perform multiple logons to server sessions. The aim is to allow all required sessions and tools to be available during the work session without the users needing to be aware of the application execution or logic.

For each application that you want to make available via the service broker concept, set up the following dynamic link libraries (DLLs):

- Service broker DLL managing the connection to the application
- Service DLL providing service functions

Furthermore, you can implement a customized service requester to call service functions of these service DLLs, if they expect a special input or output format.

- The building block concept provides the possibility to write separate programs for integration that use published interfaces of both the product and FlowMark.
- The MQSeries support is documented in this book together with some scenarios.

The Service Broker Manager and the FlowMark service broker are available for the OS/2 and Windows 3.1 platforms. The Lotus Notes service broker is available for the OS/2 and Windows 3.1 platforms. The VisualAge service broker is available for the OS/2 platform.

A service broker for VisuallInfo is part of the VisuallInfo product. Sample code to communicate via DDE and MQSeries is available on request. Contact your IBM representative.

In the future, IBM will develop code for other applications to allow rapid integration with FlowMark processes.

About Service Broker Manager

The Service Broker Manager controls the operation of service broker sessions. This includes the interaction between the service requester and services, between the service broker and services, and also the initialization of the service brokers and services.

The Service Broker Manager needs to be installed on any system on which a FlowMark activity calls a service function provided by a service broker and its services.

About service brokers

The service broker establishes and maintains a logon session with the base product (for example, Lotus Notes).

To minimize the number of simultaneous connections to the server, connections are established by a service broker (also referred to as *broker*, for example, the Lotus Notes broker). When the service broker is started, it establishes the connection to the server of the base product (depending on the implementation, the service broker might prompt you for logon information before the connection is established), and keeps the connection open as long as the service broker is running. If the service broker is stopped, it automatically logs off and removes the connection.

The service broker can share this connection with several services via a shared structure that is accessible from all registered services of this particular service broker. Whenever a service function is called, it can use the existing connection and does not need to log on again.

Often, service functions for a base product are coded by more than one person. Some standard functions can be useful in a wide range of situations, while others are specific to a certain environment. Enhancements and extensions to existing service functions might come from various sources. For these reasons, it is not recommended to put all service functions for a base product into one service. Instead the Service Broker Manager can simultaneously manage a number of services for a service broker. However, these different services can be administrated separately.

Service functions of the same service can be executed in parallel. The administrator can assign threads to the whole Service Broker Manager (for example, global threads), to a service broker, or to a single service. Service functions are then executed inside these threads.

When a service function is to be executed, the Service Broker Manager provides the thread to be used in the following order:

1. A thread assigned to the service
2. A thread assigned to the service broker
3. A global thread

Note: As memory requirements for service functions might differ significantly, the administrator also has to specify the stack size for these service threads.

If all threads are in use, the service function is queued until the next thread is freed. If no thread becomes available within a certain period of time (this period is defined with the TimeOut parameter in the respective API function call), the service function is stopped by the Service Broker Manager.

The service broker concept together with the flexible use of service threads allows not only concurrent invocations of the same function, but also synchronization. If access to a resource (for example, a scanner or a 3270 emulation) must be serialized, it could be managed by a service broker with a single thread assigned. Such a service broker is referred to as a *synchronized service broker*. Simultaneous access to a synchronized service broker and its services is then prevented by the Service Broker Manager and access attempts by requesters are queued.

Multiple instances of a service broker can be registered under different logical names (for example, if you want to support different versions of the base product).

About services and service functions

A service interfaces to the integrated product. A service function receives the user data from the Service Broker Manager and calls the appropriate product APIs to perform the work. The results are returned via the Service Broker Manager to the service requester and then back to the user application.

In the context of the service broker concept, a service function is a subroutine that provides a certain functionality. In the case of a Lotus Notes service function this might be to create a document or to read a document. Several related service functions (for example, all Lotus Notes service functions) can be stored within one service.

This functionality can be attached directly to a FlowMark activity or linked to an application. In both cases the code is loaded whenever the service is invoked. Moreover, this service function might need a connection to the server of the base product (for example, an image server). This means that, whenever FlowMark directly invokes a service function, the user is prompted for logon information and a new connection to the server is established. To avoid this, service functions are not directly attached to FlowMark, they are invoked by a service requester (see “Implementing a service requester” on page 125), and make use of an open connection managed by a service broker.

Multiple instances of a service can be registered under different logical names.

About service requesters

A service requester is the interface to the user application. The user application calls the service requester APIs to request the product to perform some work. The service requester formats the user data and issues a request to the Service Broker Manager function which forwards the request to the appropriate service function.

Service functions are not attached directly to FlowMark activities or other applications; they are invoked by a service requester (also referred to as *requester*, for example, the FlowMark requester). Consequently, a communication mechanism must exist between service requesters and the Service Broker Manager. However, the actual implementation is hidden from service requesters by means of the service requester API and might be changed in future implementations.

How these pieces fit together

Figure 1 shows how FlowMark can be integrated with other applications using the service broker concept.

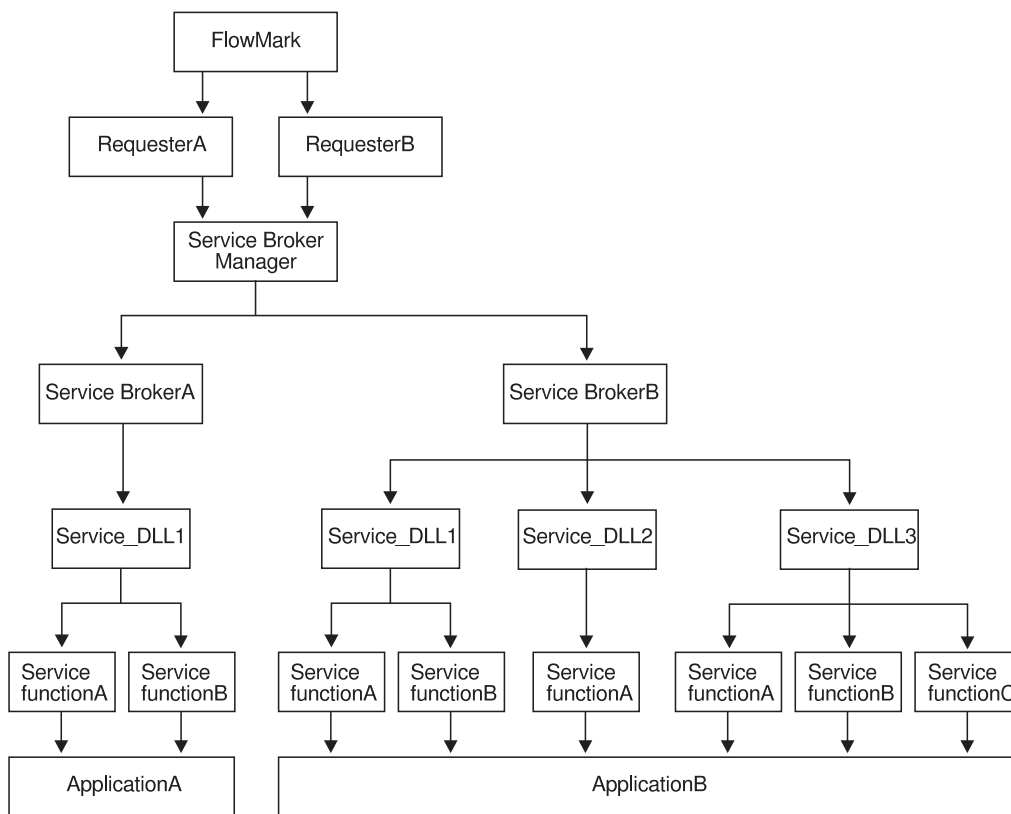


Figure 1. Using the service broker concept to integrate FlowMark with other applications

Chapter 2. Using the Service Broker Manager for OS/2

This chapter explains the different modes in which the Service Broker Manager for OS/2 can be used and how to use the Service Broker Manager, service brokers, and service DLLs.

Service Broker Manager modes

The information that you see in the Service Broker Manager windows and the tasks you can perform is controlled by the level of authorization that you have.

If you have installed the User Profile Management version, the level of authorization is assigned to you by your administrator using the OS/2 User Profile Management facility.

If you have installed the User version, you can work in *user mode*.

If you have installed the Administration version, you can work in *administrator mode*.

User mode

If you have user authorization, you can:

- Display service broker and service details
- Control the display format and refresh options
- Start and stop service brokers and services

You cannot work with the administrative functions of Service Broker Manager.

Administrator mode

If you have administrator authorization, you can:

- Display all levels of information
- Control the display format and refresh options
- Start and stop service brokers and services
- Load and unload service brokers and services
- Add and delete service brokers and services

Starting Service Broker Manager

To access the Service Broker Manager to work with or display service broker and service details:

1. Double-click on the Service Broker Manager folder icon to display the open folder.
2. Double-click on the Service Broker Manager program icon to start the Service Broker Manager.

Depending on your installation, you have one of these program icons in the Service Broker Manager folder:

- Service Broker Manager User Profile Management



Service Broker Manager
User Profile Management

- Service Broker Manager User



Service Broker Manager
User

- Service Broker Manager Administration



Service Broker Manager
Administration

The scope of functions that are available to you depends on which version you have installed:

- User Profile Management version

If you have a Service Broker Manager User Profile Management program icon, you have to log on to User Profile Management as soon as Service Broker Manager is started. Depending on the user ID with which you log on, you are granted user or administrator authorization.

If you have already logged on to your local system, for example from another OS/2 session, the Service Broker Manager window is displayed, containing details of all brokers that are currently registered.

If you have not previously logged on, a logon panel is displayed first. Type your user ID and password in the fields, and select **OK** or press Enter to log on. The default user ID and password are **userid** and **password**.

When the logon is complete, the Service Broker Manager window is displayed.

- User version

If you have a Service Broker Manager User program icon, user authorization is granted to you when you start Service Broker Manager. Double-click on the icon to start Service Broker Manager. The Service Broker Manager window is displayed.

- Administration version

If you have a Service Broker Manager Administration program icon, administrator authorization is granted to you when you start Service Broker Manager. Double-click on the icon to start Service Broker Manager. The Service Broker Manager window is displayed.

Note: If no brokers are currently registered to the Service Broker Manager (for example, the first time you start the Service Broker Manager if you did not install the supplied brokers), a message is displayed, which indicates that a new profile is created.

The message refers to the profile (INI file) that contains details of all registered brokers. Because you have no registered brokers, the profile does not yet exist. As soon as the first broker is registered, the profile is created and placed in the working directory. The working directory is the directory from which you start Service Broker Manager. If you start Service Broker Manager from the Service Broker Manager folder on the desktop, the working directory is the \BIN subdirectory of the directory where FlowMark is installed.

Select **OK** to remove the message and continue.

Stopping Service Broker Manager

Service Broker Manager can be stopped at any time. However, all currently running service brokers and services are then stopped immediately. This can cause problems for applications using the active service brokers and services (for example, data loss).

To stop Service Broker Manager, do the following:

1. Select **Close** from the window menu of the Service Broker Manager window.
A message box is displayed asking if you really want to quit Service Broker Manager.
2. Select **Yes** to stop Service Broker Manager or **No** to cancel and return to Service Broker Manager.

If you selected **Yes**, the current settings for the Service Broker Manager are automatically saved. These settings are used when Service Broker Manager is started the next time.

Defining the Service Broker Manager resources

To view or set details of resource requirements for the service brokers, use the Setup page of the Settings notebook.

In the **Resources** field, you can view or set the resources that are allocated for the Service Broker Manager. These resources can be used by all service brokers.

In the **Stack size** field, you can view or specify the stack size for the broker threads. The stack size has to be larger than 8 KB (where KB equals 1 024 bytes) and only multiples of 4 KB are allowed. If the stack size is lower than 8 KB, no thread can be started. The default stack size is 16 KB. The stack size depends on the number of variables that are coded in automatic storage.

In the **Threads** field, you can view or set the number of threads that are used for the registered service brokers. This can be any number between 1 and 20. The default value is 1.

If you select the check box **Enforce for all brokers**, all service brokers have to use these resources. If this check box is not selected, you can define own resources for the service brokers and make these resources available to the services.

Displaying brokers

All registered brokers are automatically displayed in the Service Broker Manager window when the application is started.

The number of threads currently defined (the sum of all threads configured for all brokers) and the number of threads currently active (summed up for all currently running brokers) are displayed in the status line at the bottom of the Service Broker Manager window. By default, the brokers are shown in a details view and are refreshed automatically every three seconds. The following sections describe how you can change the display and refresh options for the Service Broker Manager window.

Displaying different views of brokers

These are the types of display you can select for broker details in the Service Broker Manager window:

- Details view (this is the default)
- Tree view

To change the display of brokers, select **Details** or **Tree** as appropriate from the **View** menu in the Service Broker Manager window.

Details

Displays a list of brokers and the following information for each:

- Broker** The name of the service broker
- Threads** The total number of service threads for this broker's services
- Active** The number of active threads
- Status** The current status of the broker. This can be one of the following:
- Not loaded (Stopped in user mode)
 - Loaded (Pending in user mode)
 - Running
 - Disabled

From the details view you can:

- Double-click mouse button 1 on a service broker to display its service DLLs in a separate window
- Click mouse button 2 on a broker to display a pop-up menu for that broker

Tree

Displays an icon for each broker. The Tree view can be expanded to show an icon for each available service DLL. There are different types of icons:

Service broker icon

Indicates the name of the broker and its status:



Not loaded (Stopped in user mode)



Loaded (Pending in user mode)



Running



Disabled

If a plus (+) sign appears next to the service broker icon, this indicates that services exist for that broker.

Services icon

One icon for each service registered to each broker. The icon indicates the name of the service and its status.

From the Tree view you can:

- Click mouse button 1 on a plus sign to display service icons for that broker
- Click mouse button 2 on a broker or service icon to display a pop-up menu for that broker or service
- Double-click mouse button 1 on a broker icon to display that broker's services in a separate window
- Double-click mouse button 1 on a services icon to display that service's settings

Changing the details view

You can change the details view in the Service Broker Manager by selecting or deselecting columns to be displayed.

To define which columns are displayed, do the following:

1. Select **Settings** from the **Manager** menu to display the Settings notebook for the Service Broker Manager.
2. Select the View page of the Settings notebook.
3. Select the columns you want to be displayed in the details view by clicking on the item in the list box. The broker names are always displayed in the details view. You can include in the display also the number of defined threads (select Threads), active threads (select Active), or the status of the brokers (select Status). The list box allows multiple selections, so you can select a combination of the columns. The default is to display all columns.

Refreshing the display of brokers

You can refresh broker details in the Service Broker Manager either automatically at a specified interval or manually.

Use the following choices of the **View** menu to select the refresh setting you require:

Refresh Select **Refresh** (option checked) to activate the automatic refresh facility. The window is refreshed automatically at the interval specified on the View page of the Service Broker Manager Settings notebook.

When you start Service Broker Manager, automatic refresh is set on by default, and the refresh interval is three seconds.

Deselect **Refresh** (option not checked) to deactivate the automatic refresh facility. The window is not refreshed again until you either restart automatic refresh or select **Refresh now**.

Refresh now Use this option to immediately refresh the status of all brokers in the Service Broker Manager window.

Defining the refresh interval

To define the interval at which the Service Broker Manager window is automatically refreshed when **Refresh** is selected, do the following:

1. Select **Settings** from the **Manager** menu to display the Settings notebook for the Service Broker Manager.
2. Select the View page of the Settings notebook.
3. Use the radio buttons to define either that the Service Broker Manager window is to be automatically refreshed at the interval shown, or that the window is not to be refreshed until **Refresh now** is selected (**Initiated by user** radio button).

The value you specify for the refresh interval must be a number of seconds in the range 0 (never refreshed) to 999. The default is to refresh every three seconds.

Viewing the monitor

You can use the Service Broker Manager monitor facility to see the messages produced by the brokers and services you are using.

Use the following choices of the **Monitor** menu to work with the monitor:

Show

Select **Show** (option checked on the left side) to activate the monitor. The Service Broker Manager window is split into two panes. The upper pane still displays the brokers while the lower pane displays the messages produced by Service Broker Manager and registered brokers and services (see Figure 2).

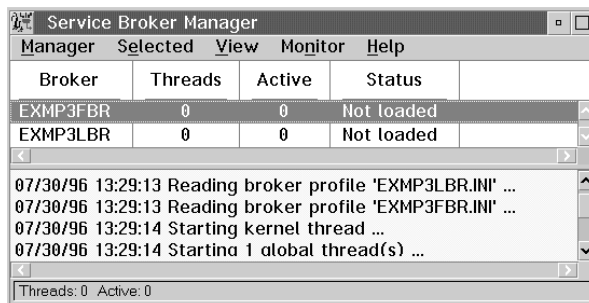


Figure 2. Service Broker Manager monitor

The messages are displayed at the level of detail specified on the Monitor page of the Service Broker Manager Settings notebook. All messages that have occurred since Service Broker Manager has been started are displayed, even if the monitor has previously been switched off. The font selected in the Settings notebook is used.

Each message is displayed on a separate line of the monitor, starting at the top of the pane. Date and time when the message occurred are displayed together with the message text.

You can size the message monitor window. Use the horizontal and vertical scroll bars to see the complete message text if necessary.

Deselect **Show** (option not checked on the left side) to deactivate the message monitor.

By default, the message monitor is not activated when you start Service Broker Manager.

Copy

Use this option to copy the content of the message monitor to the clipboard. You can open an editor window and use the paste option of the editor to paste the contents of the clipboard into the editor and save it as a file.

Erase

Use this option to delete all messages currently displayed from the message monitor window.

Defining the detail level for messages

You can change the level of detail for the messages displayed in the message monitor as well as the font that is used to display the messages.

To define these settings, do the following:

1. Select **Settings** from the **Manager** menu to display the Settings notebook for the Service Broker Manager.
2. Select the Monitor page of the Settings notebook.
3. Use the radio buttons to define the level of detail at which you want the messages to be displayed in the monitor. Select:
 - **Minimum** to display error messages only
 - **Medium** to display error messages and general status information
 - **Maximum** to display error messages, status information, and problem determination messages

As soon as you confirm the changes made in the Service Broker Manager Settings notebook by pressing the **OK** button, the Service Broker Manager messages occurring from that moment on are displayed according to the selected detail level.

Note: Messages issued by brokers do not necessarily comply with the message format of the Service Broker Manager. In this case, all messages are displayed despite the detail level you specified.

4. You can change the font to be used for displayed messages by clicking on the **Change font...** button. A font window is displayed where you can choose from different fonts and font sizes:
- Select the font you want by selecting its name from the **Name** list.
 - Define the font size by selecting one of the predefined sizes from the **Size** list. The available font sizes depend on the selected font.
 - Select the style you want to be used from the **Style** list.
 - Use the check boxes in the **Emphasis** group to define if the messages should be displayed outlined, stroked out, or underlined. Some of these options can be deactivated depending on the selected font.
 - A sample of the font currently selected is displayed in the **Sample** box.
 - Press **OK** to set a font for the message monitor, choose **Cancel** to cancel the font selection.

A sample of the currently selected font is displayed in a window in the Settings notebook left to the **Change font...** button. The default OS/2 system font is used as default font.

Managing brokers

You can manage and register service brokers from the Service Broker Manager, using the choices from the **Selected** menu of the Service Broker Manager window:

- | | |
|-------------------------|--|
| Open → | Open a window containing services registered to a selected broker. |
| Settings | Display the properties of a selected broker. |
| Create another → | Register a new broker or service.
Note: This option is available in administrator mode only. |
| Delete | Delete a selected broker.
Note: This option is available in administrator mode only. |
| Load | Load a selected broker.
Note: This option is available in administrator mode only. |
| Start | Start a selected broker. |
| Stop | Stop a selected broker that is currently running. |
| Unload | Unload a selected broker that is currently loaded or running.
Note: This option is available in administrator mode only. |

You can also select these actions from a pop-up menu. Select a broker and click mouse button 2 once to display this menu.

The actions are described in the following sections.

Displaying settings for a broker

To display the settings for a specific broker:

1. Select this broker from the list in the Service Broker Manager window.
2. Select **Settings** from the **Selected** menu or pop-up menu to open the Settings notebook for this broker.

This notebook, and details of each option, are described in “Registering a new broker.”

Note that you cannot change the logical name for an existing broker.

Registering a new broker

You can either register a new broker by interactively entering all details required or by using an already existing broker profile. Details of how to register brokers and services are described in the following sections.

Registering a new broker interactively

To register a new broker interactively:

1. Select **Create another**→**Broker** from the **Selected** menu or pop-up menu of the Service Broker Manager window. This choice is the default choice for the **Create another**→ menu option. A blank Settings notebook is displayed, as shown in Figure 3.

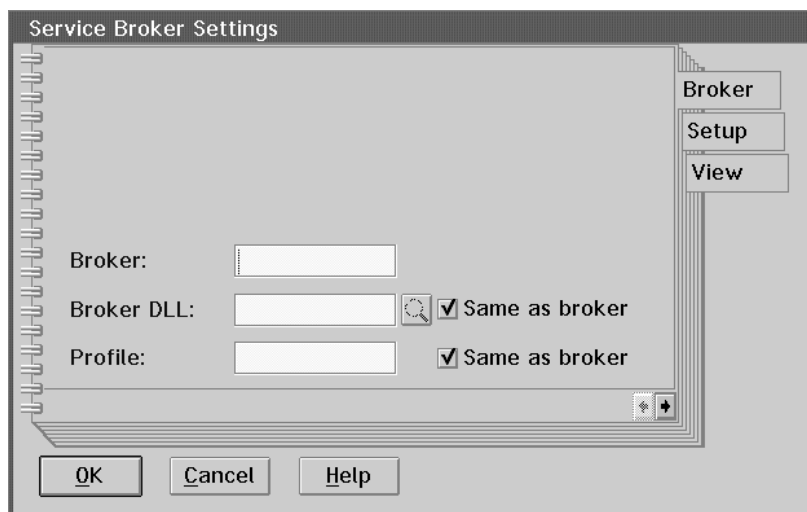


Figure 3. Broker registration notebook

2. Type the appropriate details for your broker on each page and select **OK** to register the broker. Or, select **Cancel** to cancel the registration.

On the Broker page you can specify the following:

Broker Logical name of the broker. This is the name that appears in the Details and Tree view of the broker.

Broker DLL Name of the dynamic link library (without extension) containing the broker functionality (for example, logon, logoff, versioning). By default, it is assumed that this is the same name as the logical name, so you do not need to type the name again.

If the DLL name is different from the logical name, deselect **Same as broker** and type the DLL name (without the .DLL extension) or the fully qualified path and name. For example, assuming the working directory is D:\EXM\BIN either of the following would be valid:

```
EXMP3FBR
D:\EXM\DLL\EXMP3FBR.DLL
..\DLL\EXMP3FBR
```

Note: If you specify the DLL name without extension and path, the path where the DLL is located must be included in the LIBPATH in your CONFIG.SYS file.

Profile The name of the profile used to store details of all services that are registered for this broker. By default, it is assumed that this is the same name as the logical name, so you do not need to type the name again.

If the profile name is different from the logical name, deselect the **Same as broker** and type the profile name (without the .INI extension) or the fully qualified path and name.

For example, to register a broker DLL called EXMP3FBR.DLL, with the same logical name and a profile FM.INI, you would specify the values shown in Figure 4.

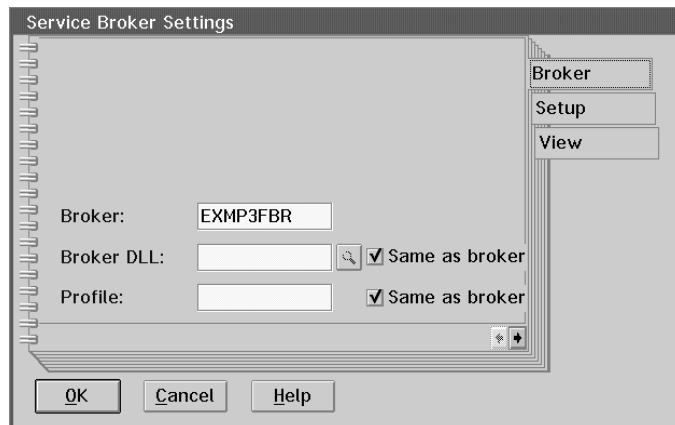


Figure 4. Example of broker registration

On the Setup page, you can specify the following:

Separate resources

If required, select the check box **Separate resources** to specify that this broker needs separate resources (that is, other resources than those provided by the Service Broker Manager). This selection enables the **Resources** to be filled in.

Note: You can specify separate resources only if the resource setup of the Service Broker Manager is not enforced for all brokers.

Resources

Stack size The stack size provided for the service threads. The stack size has to be larger than 8 KB and only multiples of 4 KB are allowed. If the stack size is lower than 8 KB, no thread can be started. The default stack size is 16 KB. The stack size depends on the number of variables that are coded in automatic storage.

Threads The number of threads that are used for service requests for this service. This can be any number from 1 to 20; the default value is 2.

Select **Enforce for all services** to make sure that all services are using the resources specified in the broker settings. If this check box is not marked, services can define their own resources. The default is that the resources are enforced for all services.

See Figure 5 for an example of a resource configuration.

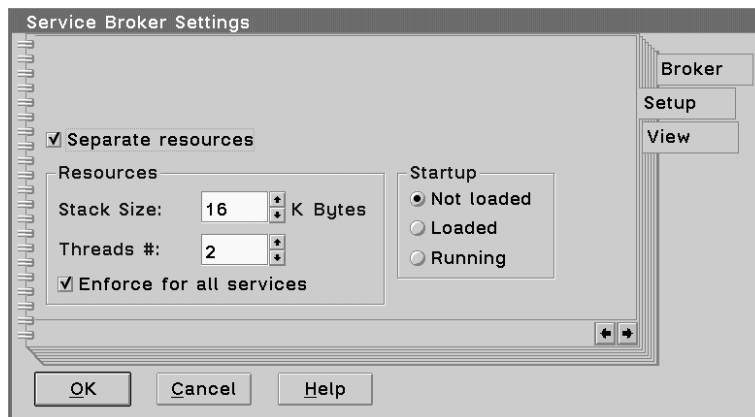


Figure 5. Example of broker resource settings

Startup

Not loaded	Broker is not loaded when Service Broker Manager is started. This is the default value.
Loaded	Broker is loaded but not started when Service Broker Manager is started.
Running	Broker is loaded and started when Service Broker Manager is started.

On the View page, you can specify the following:

Refresh

Select **Refresh** to activate the automatic refresh facility. The window is refreshed automatically at the interval specified in the settings for this window.

By default, automatic refresh is set on and the refresh interval is three seconds.

Deselect **Refresh** to deactivate the automatic refresh facility. The window is not refreshed until you either restart automatic refresh or request an immediate refresh.

Changing the details view

The details view can be modified in the same way as for the Service Broker Manager window. See “Changing the details view” on page 12 for details.

Registering a new broker from file

To register a new broker from file:

1. Select **Create another**→**Broker from file** from the **Selected** menu or pop-up menu of the Service Broker Manager window. A file selection box is displayed.
2. Select the profile (*.INI) containing the definitions for the new broker you want to register. The broker is automatically registered together with the services defined in the profile.

Deleting a broker

To delete a broker from the Service Broker Manager:

1. Select the respective broker from the list in the Service Broker Manager window.
2. Select **Delete** from the **Selected** or pop-up menu.

You are prompted to confirm the deletion, as shown in Figure 6.



Figure 6. Delete broker confirmation window

3. Select **Yes** if you want to delete the broker or **No** to cancel. If you press Enter, the deletion is canceled.

Note: You cannot delete a broker while it is running.

Loading and unloading a broker

Once a broker has been registered, you can load it or unload it as required.

To load a broker with the status Not loaded:

1. Select the broker from the list in the Service Broker Manager window.
2. Select **Load** from the **Selected** or pop-up menu. The status of the broker is changed to Loaded.

To unload a broker with the status Loaded or Running:

1. Select the respective broker from the list in the Service Broker Manager window.
2. Select **Unload** from the **Selected** or pop-up menu. The status of the broker is changed to Not loaded.

Starting and stopping a broker

Once a broker has been registered, you can start it whenever its status is shown as Loaded (Stopped in user mode) or Not loaded in administrator mode (Pending in user mode). When the broker is no longer required, you can stop it from running.

To start a broker that has the status Loaded (Stopped) or Not loaded (Pending):

1. Select the broker you want to start from the list in the Service Broker Manager window.
2. Select **Start** from the **Selected** or pop-up menu. The status of the broker is changed to Running.

If you are in user mode, the status of the broker is Pending until it has been loaded and the start process has been completed.

If the start of a broker is not successful, its status is changed to pending if you are in user mode; in administrator mode, the broker remains loaded. In this case, do one of the following:

- Correct the problem that occurred when you started the broker and select **Start** from the **Selected** or pop-up menu. The status of the broker is changed to Running.
- In user mode, select **Stop** from the **Selected** menu to stop the broker. The status of the broker is changed to Stopped.
- In administrator mode, select **Unload** from the **Selected** menu to unload the broker. The status of the broker is changed to Unloaded.

To stop a broker that has the status Running:

1. Select the respective broker from the list in the Service Broker Manager window.
2. Select **Stop** from the **Selected** or pop-up menu. The status of the broker is changed to Loaded in administrator mode or to Stopped in user mode.
3. If you are in administrator mode, you can, if required, now unload the broker.

If a running broker is stopped abnormally and cannot be restarted, it is disabled. The status of the broker is changed from Running to Disabled. You have to restart Service Broker Manager to be able to restart a disabled broker.

You can use the message monitor to determine the problem that causes a broker to become pending or disabled. Refer to “Viewing the monitor” on page 13 for details on how to use the message monitor.

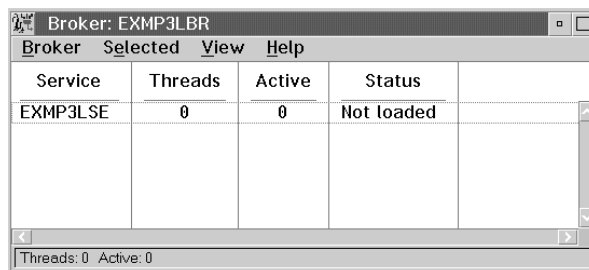
Displaying services for a broker

For every registered broker, several broker services can be registered. These services are the components that make the actual API calls to base products in the service broker concept (see Chapter 6, “Creating your own service brokers” on page 101 for details).

To display the services that are currently registered to a broker, you can do one of the following:

- Double-click on the broker name in the Service Broker Manager window.
- Select the broker and select **Open**→**Details View** or **Open**→**Icon View** from the **Selected** or pop-up menus.
- Click on the plus sign to the left of the broker icon, if you are working in Tree view.

Details of all services for the selected broker are displayed in a window titled with the broker name. An example is shown in Figure 7.



Service	Threads	Active	Status
EXMP3LSE	0	0	Not loaded

Figure 7. Broker details window

The figure shows the details view of this window. This is the default. You can change the display and refresh choices for this window, if required, as described in “Displaying different views of brokers” on page 10 and “Refreshing the display of brokers” on page 12.

Displaying different views of services

To change the display of services for a selected broker, select **Details** or **Tree** as appropriate from the **View** menu in the Service Broker window.





Details Displays a list of services and the following information for each:

Service	The name of the service
Threads	The number of threads waiting for service requests to this service (idle threads)
Active	The number of active threads
Status	The status of each service. This can be: <ul style="list-style-type: none">• Not loaded or Loaded in administrator mode• Stopped in user mode• Running• Disabled

From the details view, you can:

- Click mouse button 2 on a service to display a pop-up menu for that service
- Double-click mouse button 1 on a service to display that service's settings

Tree Displays an icon for each registered service that shows its status:

	Not loaded (Stopped in user mode)
	Loaded (Pending in user mode)
	Running
	Disabled

From the Tree view, you can:

- Click mouse button 2 on a service icon to display a pop-up menu for that service
- Double-click mouse button 1 on a service icon to display that service's settings

Refreshing the display of services

You can select to refresh the contents of a broker window either automatically at a specified interval or manually.

Use the following choices of the **View** menu of the broker window to select the refresh option you want:

- Refresh** Select **Refresh** to activate the automatic refresh facility. The window is refreshed automatically at the interval specified in the settings for this window.
- By default, automatic refresh is set on and the refresh interval is three seconds.
- Deselect **Refresh** to deactivate the automatic refresh facility. The window is not refreshed until you either restart automatic refresh or request an immediate refresh using **Refresh now**.
- Refresh now** Use this function to refresh immediately the status of all services in the window.

To define the interval at which a broker window is automatically refreshed when **Refresh** is selected for that window, do the following:

1. In the Service Broker Manager window, select the respective broker. Then, select **Settings** from the **Selected** menu.
Or, display the broker window by double-clicking on the broker and then select **Settings** from the **Broker** menu.
In either case, the Settings notebook for the selected broker is displayed.
2. Select the **View** tab to display the current settings. An example is shown Figure 8 on page 25.

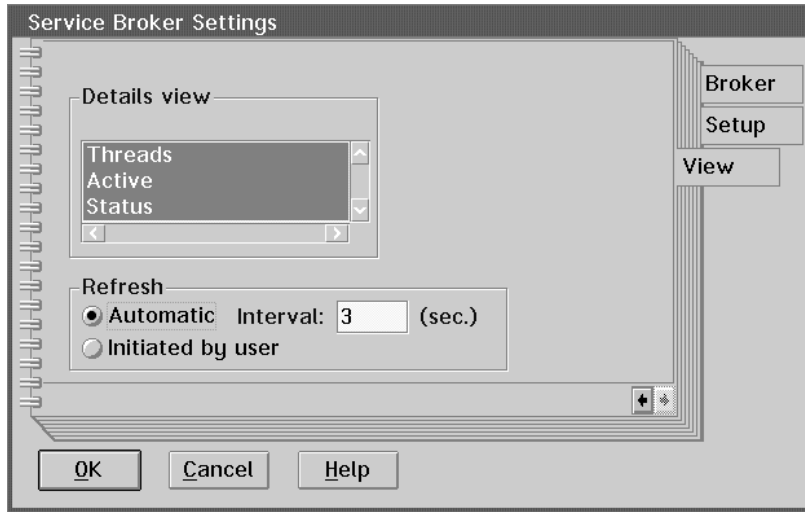


Figure 8. Service Broker Settings notebook

- Use the radio buttons to define either that the service details are to be automatically refreshed, at the interval shown in the box, or that the window is not to be refreshed until **Refresh now** is selected (**Initiated by user** radio button).

The value you specify for the refresh interval must be a number of seconds in the range 0 (never refreshed) to 999. The default is to refresh every three seconds.

Managing broker services

Use the choices of the **Selected** menu in the broker window to work with services:

- | | |
|-------------------------|---|
| Settings | Display the properties of a registered broker service.
Note: This option is available in administrator mode only. |
| Create another → | Register a new service to this broker.
Note: This option is available in administrator mode only. |
| Delete | Delete a service from this broker.
Note: This option is available in administrator mode only. |
| Load | Load a registered service.
Note: This option is available in administrator mode only. |
| Start | Start a service. |
| Stop | Stop a service that is currently running. |
| Unload | Unload a broker service that is currently loaded or running.
Note: This option is available in administrator mode only. |

Each of these choices is described in detail in the following sections.

Displaying settings for a service

To display the settings for a specific service:

1. Select the service from the list in the broker window.
2. Select **Settings** from the **Selected** or pop-up menu to open the Settings notebook for this service.

This notebook, and details of each option, are described in “Registering a new service.” Note that you cannot change the logical name for an existing service.

Registering a new service

To register a new broker service, do the following:

- From the Broker window:
 1. Display the service details for the broker to which you want to add the service (as described in “Displaying services for a broker” on page 22).
 2. Select **Create another** from the **Selected** or pop-up menu.
- From the Service Broker Manager window:
 1. Select the broker for which you want to register a new service from the Service Broker Manager window.
 2. Select **Create another**→**Service** from the **Selected** menu or pop-up menu.

A blank Settings notebook is displayed, as shown in Figure 9.

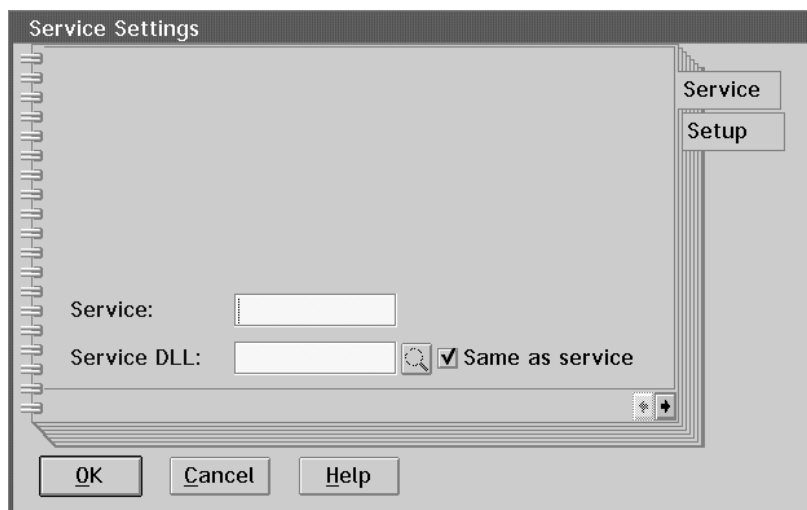


Figure 9. Service registration notebook

Type appropriate details for your service on each page as described in the following and select **OK** to register the service to this broker. Select **Cancel** to cancel the registration.

On the Service page, you can specify the following:

Service The logical name of the service. This is the name that appears in the Details and Tree view of the broker window.

Service DLL

The name of the dynamic link library (without extension) containing the product-specific functions that are provided by the service. By default, this is the same name as the logical name, so you do not need to type the name again.

If the DLL name is different from the logical name, deselect **Same as service** and type the DLL name (without the .DLL extension) or the fully qualified path and name. For example, either of the following would be valid, assuming the working directory is D:\EXM\BIN:

D:\EXM\DLL\EXMP3FFM.DLL
..\DLL\EXMP3FFM.DLL

If you do not know the exact name of the service DLL, select **Same as service** and click on the find button. A file selection box is displayed where you can select the appropriate service DLL file. Press **OK** in the file selection box to make your selection effective or choose **Cancel** to cancel your selection.

For example, to register a service DLL called EXMP3FFM.DLL with the same logical name and six threads waiting for service requests, you would specify the values shown in Figure 10 and Figure 11 on page 29.

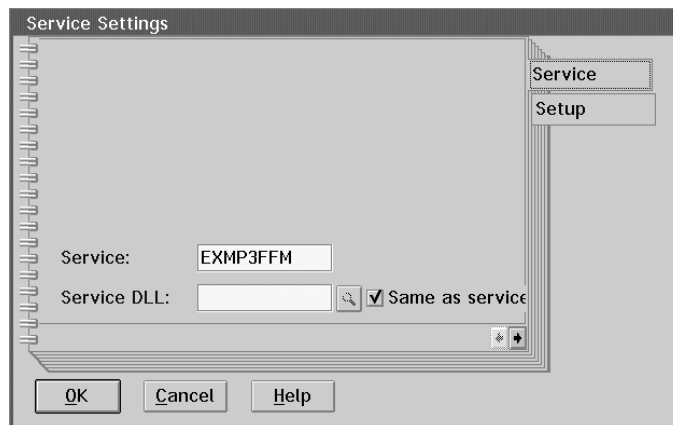


Figure 10. Example of service registration

On the Setup page, you can specify the following:

Startup of service

Not loaded The service is not loaded automatically when the broker is started.

This is the default value.

Loaded The service is loaded automatically but not started when the broker is started.

Running The service is loaded and started when the broker is started.

Separate resources

Select **Separate resources** to specify that this service needs separate resources (that is, other resources than those provided by the Service Broker Manager). This selection activates the **Resources** fields.

Note: You can specify separate resources only if the resource setup of the service broker is not enforced for all services.

Resources

Stack size The stack size provided for the service threads. The stack size has to be larger than 8 KB and only multiples of 4 KB are allowed. If the stack size is lower than 8 KB, no thread can be started. The default stack size is 16 KB. The stack size depends on the number of variables that are coded in automatic storage.

Threads The number of threads that are used for service requests for this service. This can be any number from 1 to 20; the default value is 2.

Note: To synchronize access to a service (for example, for a host logon service that can accept only one logon at a time), set the value of **Threads** to 1. Requests are then queued and access the service one at a time.

For an example of service resource settings see Figure 11.

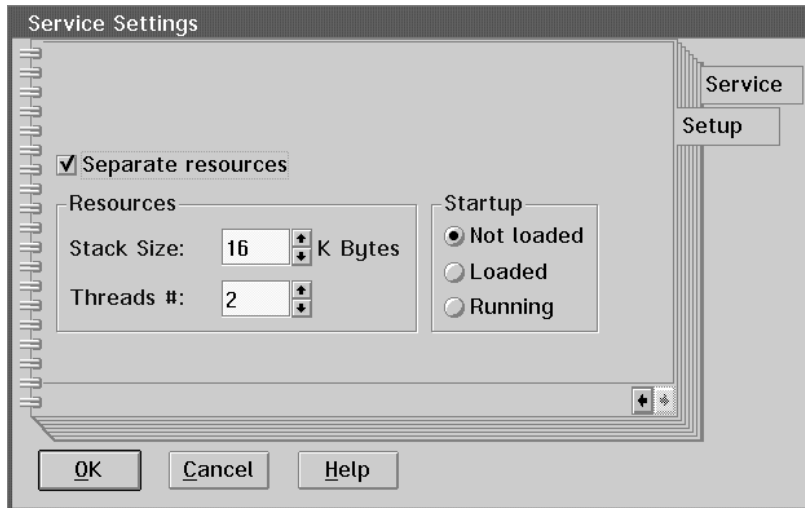


Figure 11. Example of service resource settings

Note: If you want to register a service that does not require a dedicated broker, you can register this service to the standard broker EXMP3CST. For example, if you do not have to log on to a base application or if you want to implement all the logic in the service DLL.

The EXMP3CST broker is a broker without an underlying base application. It contains only the mandatory broker functions for the Service Broker Manager.

If you want to use the standard broker, you have to:

1. Register the standard broker EXMP3CST (DLL name EXMP3CST.DLL) which is part of Service Broker Manager installation—see “Registering a new broker” on page 16 for information about how to register a broker.
2. Display the service details for the broker EXMP3CST (as described in “Registering a new broker” on page 16).
3. Select **Create another** from the **Selected** or pop-up menu of the EXMP3CST broker window.
4. Register the new service as described previously in this section.

Deleting a service

To delete a service from a broker:

1. Select the respective service from the list in the broker window.
2. Select **Delete** from the **Selected** or pop-up menu.

You are prompted to confirm the deletion, as shown in Figure 12.



Figure 12. Delete Object(s) confirmation window

3. Select **Yes** if you want to delete the service or **No** to cancel. If you press Enter, the deletion is canceled.

Note: You cannot delete a service while it is running.

Loading and unloading a service

Once a service has been registered, you can load or unload it as required. To load a service that has the status Not loaded, follow these steps:

1. Select the respective service from the list in the broker window.
2. Select **Load** from the **Selected** or pop-up menu. The status of the service is changed to Loaded.

To unload a service with the status Loaded or Running, do the following:

1. Select the respective service from the list in the broker window.
2. Select **Unload** from the **Selected** or pop-up menu. The status of the service is changed to Not loaded.

Starting and stopping a service

Once a service has been registered, you can start it whenever its status is Loaded or Not loaded. When the service is no longer required, you can stop it.

To start a service that has the status Loaded or Not loaded in administrator mode or Stopped in user mode:

1. Select the service from the list in the broker window.
2. Select **Start** from the **Selected** or pop-up menu. The status of the service is changed to Running.

If you are in user mode, the status of the service is Pending until it has been loaded and the start process has been completed.

If the start of a service was not successful, its status is changed to Pending if you are in user mode. In administrator mode, the service remains loaded. In this case, you can do one of the following:

- Correct the problem that occurred when you started the service and select **Start** from the **Selected** or pop-up menu. The status of the service is changed to Running.
- In user mode, select **Stop** from the **Selected** menu to stop the service. The status of the service is changed to Stopped.
- In administrator mode, select **Unload** from the **Selected** menu to unload the service. The status of the service is changed to Unloaded.

To stop a service that has the status Running, do the following:

1. Select the respective service from the list in the broker window.
2. Select **Stop** from the **Selected** or pop-up menu. The status of the service is changed to Loaded in administrator mode and to Stopped in user mode.
3. If you are in administrator mode, you can now unload the service, if required, as described in “Loading and unloading a service” on page 30.

If a running service is stopped abnormally and cannot be restarted, it is disabled. The status of the service is changed from Running to Disabled. You have to restart Service Broker Manager to be able to restart a disabled service.

You can use the message monitor to determine the problem that caused a service to become pending or disabled. Refer to “Viewing the monitor” on page 13 for details on how to use the message monitor.

Dynamic start of services

If a service is requested which has not been started yet, it is started dynamically by Service Broker Manager. If the broker to which the service is registered has not been started yet, the broker itself is started and afterwards the requested service is activated.

Services and brokers can also be started and stopped dynamically by using the service requester API functions (C, REXX) or the program EXMP3UCT.EXE.

The service requester API functions are described in “Implementing a service” on page 116. For more information about the program EXMP3UCT.EXE refer to “Using the standard external controller” on page 145.

Chapter 3. FlowMark service broker

This chapter describes the interface code for the FlowMark service broker that enables you to manipulate FlowMark processes. The interface code is provided for OS/2 and for Windows 3.1.

This component consists of:

- A FlowMark service broker library
- A service library to manipulate FlowMark processes
- A C header file defining the structure for the session data:
 - EXMP3FBR.H for OS/2
 - EXMW3FBR.H for Windows
- An FDL (FlowMark definition language) file containing a demo process
- A REXX program to manipulate the FlowMark demo process

You can find information about when to use this broker in Appendix A, “Ways to integrate FlowMark and Lotus Notes” on page 223.

FlowMark service broker library

The following table shows the DLLs for the different components:

	OS/2	Windows 3.1
FlowMark service broker	EXMP3FBR.DLL	EXMW3FBR.DLL
FlowMark service	EXMP3FFM.DLL	EXMW3FSE.DLL
FlowMark requester	EXMP3FRQ.DLL	EXMW3FRQ.DLL

Note: The provided library can only be used if a FlowMark Runtime client is installed on your workstation.

FlowMark requester

You can use the FlowMark requester to invoke service functions from a FlowMark program activity and also to pass FlowMark container data to services and retrieve data from services (see Figure 13).

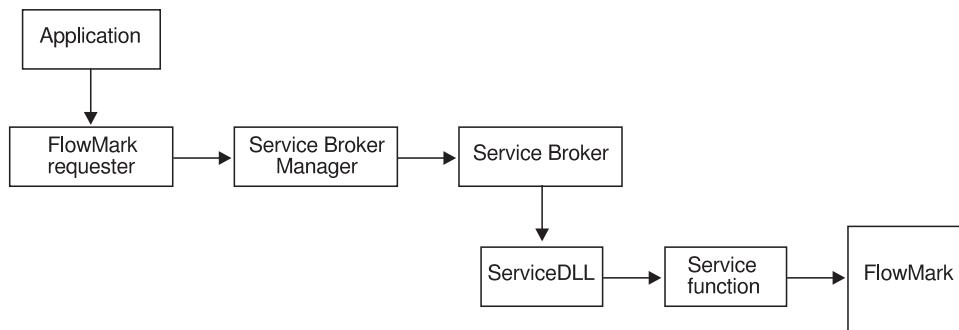


Figure 13. Service broker concept for FlowMark

Using the FlowMark requester

1. Register the service broker DLL as a broker within the Service Broker Manager.
2. Register the service DLL as a service within the Service Broker Manager.
3. On the OS/2 or Windows page of the FlowMark Program Settings notebook, specify:
 - In the field **Path and file name**, the requester DLL
 - In the field **Entry point**, one of the following functions:

Request

This function passes *parameters* to *function*. No return parameter is expected.

Specify as command-line parameters in the FlowMark program registration:

broker service function parameters

Where:

broker Is the name of the FlowMark broker registered in Service Broker Manager

service Is the FlowMark service registered to the broker

function Is the name of the service function to be invoked

parameters Is the input parameter required by the function

Refer to “FlowMark service functions” on page 35 for information about the standard service functions and the parameters expected by these functions.

FMRequest

This function passes the FlowMark session ID to *function*. This requester function can only be used to invoke service functions specifically designed to work with FlowMark container data. The FlowMark session ID which can be used to access FlowMark container data using the FlowMark Container API functions is passed to the invoked service function as the first parameter in the list of *parameters*.

Specify as command-line parameters in the FlowMark program registration:

broker service function parameters

RequestString

This function passes *parameters* to *function*. The service function has to return a string that is stored in *variable name*.

Specify as command-line parameters in the FlowMark program registration:

broker service function parameters variable_name

Where *variable_name* represents the name of a variable in the data structure that is specified as output data structure of the program definition in FlowMark.

Note: Under Windows, enclose the command line parameters in quotes:

"broker service function parameters [variable_name]"

FlowMark service functions

The FlowMark service broker provides six standard service functions, which are described in the following sections (for further information you can also refer to Chapter 6, "Creating your own service brokers" on page 101). The feedback of these service functions depends on the entry point you specified in the FlowMark program settings.

All parameters described in the following sections are general text strings that are constructed as follows:

- If the first character of the string is not a single quote ('), the parameter is defined by all characters up to the first blank (ASCII code 32) or the end of string character (ASCII code 0). No further restrictions are placed on the kind of characters in the string.
- If the first character of the string is a single quote ('), the parameter consists of all characters up to the next single quote or the end of string character. All other characters in the parameter are valid. To specify a single quote in between the string type two consecutive single quotes (').

Note: Although every character can be specified in a parameter (and can be read by the FlowMark service broker) this does not mean that all these characters are allowed in the FlowMark API functions (for detailed information refer to Chapter 6, “Creating your own service brokers” on page 101) Also, the maximal length of the parameter string depends on the respective FlowMark API function.

Starting a process instance

The function StartProcess creates a process instance from an existing process template and then starts the resulting process instance. It can also transfer initial values for data items defined by the data structure of the input container.

The syntax is:

```
►►—StartProcess—TemplateName—InstanceName—MemberDataName:MemberDataType=Value—◄◄
```

The parameters are:

TemplateName

Specifies the name of an existing process template.

InstanceName

Specifies a name for the process instance.

MemberDataName

Specifies the fully qualified data item to be set (spaces between the MemberDataName, MemberDataType or Value are optional).

Note: If the MemberDataName is defined without single quotes notation, then a colon (:) is also a delimiter.

MemberDataType

Specifies the data type and must be one of the following:

- S or s for String
- L or l for Long
- F or f for Float

Note: If the MemberDataType is defined without single quotes notation, then an equal sign (=) is also a delimiter.

Value

Specifies a value appropriate to the MemberDataType.

Example:

```
StartProcess 'FlowMark Template1' 'Heinrich IV' Pi:F=3.14159 'abc & d':L=123456
```

This starts the process FlowMark Template1 giving it the instance name Heinrich IV and passes the float value 3.14159 to the data container member Pi.

Suspending a running process instance

The function `SuspendProcess` suspends a running process instance.

The syntax is:

```
▶▶—SuspendProcess—InstanceName—└─Mode—┘────────────────────────────────────────▶▶
```

The parameters are:

InstanceName

Specifies the name of the process instance to be suspended.

Mode

If defined it must be one of the following (not case-sensitive):

CURRENT Suspends only the specified process instance.

ALL Suspends the specified process instance and all its subprocesses.

QUERY Returns the name of the top-level process. The process instance is not suspended.

Note: The default value for *Mode* is **CURRENT**.

Example:

```
SuspendProcess 'Heinrich IV' 'all'
```

This suspends the process instance `Heinrich IV` and all its subprocesses.

```
SuspendProcess Hamlet
```

This suspends the process instance `Heinrich IV` only.

Resuming a suspended process instance

The function `ResumeProcess` resumes a process instance that has been suspended. The process instance continues from where it was suspended.

The syntax is:

```
▶▶—ResumeProcess—InstanceName—└─Mode—┘────────────────────────────────────────▶▶
```

The parameters are:

InstanceName

Specifies the name of the process instance to be resumed.

Mode

If defined it must be one of the following (not case-sensitive):

CURRENT Resumes only the specified process instance.

ALL Resumes the specified process instance and all its subprocesses.

QUERY Returns the name of the top-level process. The process instance is not resumed.

Note: The default value for *Mode* is CURRENT.

Example:

```
ResumeProcess 'Oberon & Miranda' current
```

This resumes the process instance Oberon & Miranda.

Terminating a running process instance

The function `TerminateProcess` terminates a process instance that is running or has been suspended. Only a top-level process can be specified.

The syntax is:

```
►—TerminateProcess—InstanceName—┐  
└─┬─┘  
  Mode
```

The parameters are:

InstanceName

Specifies the name of the process instance to be terminated.

Mode

If defined, it must be one of the following (not case-sensitive):

ALL Terminates the process instance and all its subprocesses.

QUERY Returns the name of the top-level process. The process instance is not terminated.

Note: The default value for *Mode* is ALL.

Example:

```
TerminateProcess Hamlet
```

This terminates the process instance Heinrich IV and all its subprocesses.

Restarting a process instance

The function `RestartProcess` restarts a finished or terminated process instance. Only a top-level process can be specified. The process instance starts from the beginning. The contents of the input container of the process instance as used for the first execution are used for the restart.

The syntax is:

```
▶▶ RestartProcess InstanceName Mode ◀◀
```

The parameters are:

InstanceName

Specifies the name of the process instance to be restarted.

Mode

If defined, it must be one of the following (not case-sensitive):

ALL Restarts the specified process instance.

QUERY Returns the name of the top-level process. The process instance is not restarted.

Note: The default value for *Mode* is **ALL**.

Example:

```
RestartProcess 'So long & thanx for the fish' all
```

This restarts the process instance `So long & thanx for the fish` and all its subprocesses.

Changing the status of an activity

The function `ChangeActivityState` changes the status of an activity. It is possible to start or finish a ready activity, or to restart or finish a running activity.

The syntax is:

```
▶▶ ChangeActivityState InstanceName QualifiedActivityName Mode ◀◀
```

The parameters are:

InstanceName

Specifies the name of the ready or running process instance.

QualifiedActivityName

Specifies the fully qualified name of the activity whose status is to be changed.

Mode

Must be one of the following (not case-sensitive):

- START** Starts the specified activity. The activity has to be in the ready status.
- RESTART** Forces a restart of the specified activity. The activity has to be in the running status.
- FINISH** Forces the specified activity to finish. The activity has to be in either the ready or running status.

Example:

```
ChangeActivityState Dornroeschen sleep RESTART
```

This restarts the activity `sleep` in the process instance `Dornroeschen`.

Sample service requester calling function `FMRequest()`

If you want to use FlowMark service functions, such as `StartProcess`, in your application, you can write your own FlowMark service requester. The following example shows how to do that with the function `FMRequest()`.

`FMRequest` has these parameters:

- The session ID, which you must retrieve with a call to `ExmcGetSessionID()`
- A parameter string with the following sequence:
 1. `pszBroker`
This must be `EXMP3FBR` (OS/2), `EXMW3FBR` (Windows 3.1), or `EXMB3FBR` (Windows NT).
 2. `pszService`
This must be `EXMP3FFM` (OS/2), `EXMW3FSE` (Windows 3.1), or `EXMB3FSE` (Windows NT).
 3. `pszFunction`
This must be a FlowMark service function, such as `StartProcess` or `SuspendProcess`.
 4. `pszInArea`
This specifies the input for the function, for example, `FlowMark Template1 Heinrich IV`.

Example for Windows 3.1

```
VOID CallFMRequester()
{
    char        pszSessionID[EXMPJ_SESS_BUF_LEN];
    char        pszActivityString[ACTIVITY_LENGTH];
    char        pszmsgbuf[256];
    unsigned    ulCurrentLength;
    RETURNCODE rc;

    char FAR * pszBroker   = "EXMW3FBR";
    char FAR * pszService  = "EXMW3FSE";
    char FAR * pszFunction = "StartProcess";
    char FAR * pInAreaPtr = "FlowMark Template1";

    /******
    /* Fill the activity string in the sequence:          */
    /* 1) Broker                                          */
    /* 2) Service                                        */
    /* 3) Function                                       */
    /* 4) Inarea                                        */
    /******
    strcpy (pszActivityString, pszBroker );
    strcat (pszActivityString, " " );
    strcat (pszActivityString, pszService );
    strcat (pszActivityString, " " );
    strcat (pszActivityString, pszFunction );
    strcat (pszActivityString, " " );
    strcat (pszActivityString, pInAreaPtr );

    /******
    /* Get FM's session ID and call FMRequest to handle  */
    /* the request.                                       */
    /******
    rc = ExmcGetSessionID(pszSessionID);
    if (rc == EXMPJ_OK)
    {
        apiRc = FMRequest(pszSessionID, pszActivityString);
    }

    return;
}
```

FlowMark service broker demo

The FlowMark service broker demo is available for OS/2 and consists of two files:

EXMP3SRX.FDL

The FDL file containing the Sample FlowMark process. Import this file into a FlowMark database.

EXMP3SFM.CMD

A REXX program that allows the user to start, suspend, resume, terminate, and restart the sample process defined in EXMP3SRX.FDL. Moreover, the activity contained in this process can be started, restarted, and finished.

Start the demo by entering **exmp3sfm** at an OS/2 command prompt. The procedure lets you select one of the following activities:

- Start instance
- Suspend instance
- Resume instance
- Terminate instance
- Start activity
- Restart activity
- Finish activity

Each activity calls the corresponding FlowMark service function and passes a return code back to the procedure.

Chapter 4. FlowMark—Lotus Notes interface

This chapter describes how FlowMark can be integrated with Lotus Notes. It describes the Lotus Notes broker and the FlowMark requester for the Lotus Notes services. Figure 14 shows the service broker concept for the FlowMark—Lotus Notes interface.

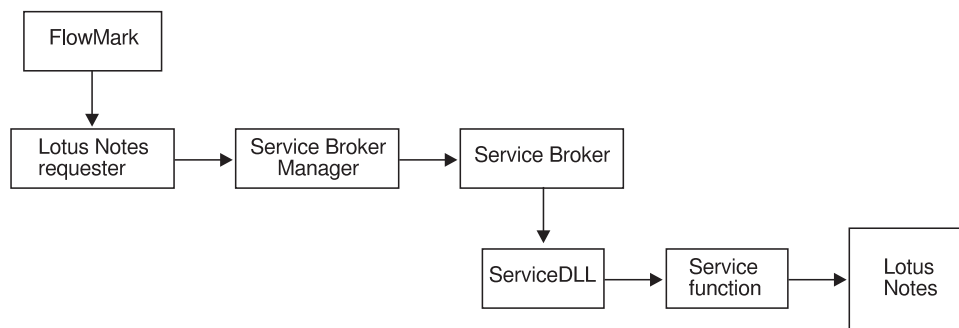


Figure 14. Service broker concept for FlowMark—Lotus Notes

Note: You must use the Lotus Notes requester to work with the Lotus Notes broker.

You can find information about when to use this broker in Appendix A, “Ways to integrate FlowMark and Lotus Notes” on page 223.

Lotus Notes service broker library

The following table shows the DLLs for the different components:

	OS/2	Windows
Lotus Notes service broker	EXMP3LBR.DLL	EXMW3LBR.DLL
Lotus Notes service	EXMP3LSE.DLL	EXMW3LSE.DLL
Lotus Notes requester	EXMP3LRQ.DLL	EXMW3LRQ.DLL

Note: The provided library can only be used if a FlowMark Runtime client and a Lotus Notes client are installed on your workstation.

Lotus Notes broker

The Lotus Notes broker enables a logon to a Lotus Notes database. The names of the database and the server are passed to the Lotus Notes service broker via the FlowMark input container (see DBOpen function on page 46). Within one process, you can work with several databases sequentially.

Important for OS/2: When you register a Lotus Notes broker within the Service Broker Manager, enter the following information in the Settings notebook:

- **Separate resources** enabled
- **Enforce for all services** enabled
- **Threads #:** 1

When the Lotus Notes broker is started, it accesses your mail server on your Lotus Notes server, so that the Lotus Notes logon panel is displayed. If no mail server is found, an informational message is written to the log file and processing continues.

If a server name is provided via FlowMark input container with the DBOpen function call, the Lotus Notes logon panel is displayed.

Note: The Lotus Notes server and database are not opened until the DBOpen function is called. Therefore, always define the DBOpen function call as the first activity in your process.

Lotus Notes requester

The Lotus Notes requester makes it possible to call the Lotus Notes service functions directly from a FlowMark process.

Using the Lotus Notes requester

1. Register the service broker DLL as a broker within the Service Broker Manager.
2. Register the service DLL as a service of the Lotus Notes broker within the Service Broker Manager.
3. On the OS/2 or Windows page of the FlowMark Program Settings notebook, specify:
 - The requester DLL in the field **Path and file name**
 - The entry point, Request, in the field **Entry point**

Specify as **Command line parameters:**

broker service function parameters

Where:

broker Is the name of the Lotus Notes broker registered in Service Broker Manager

service Is the name of the Lotus Notes service registered to the Lotus Notes broker

function Is the name of the service function (described in "Lotus Notes service functions") to be invoked

parameters Are the input parameters required by the function

Note: Under Windows, enclose the command line parameters in quotes:

"broker service function parameters"

You can find more information about registering programs in FlowMark in the *Modeling Workflow* manual.

Lotus Notes service functions

The following Lotus Notes service functions are available for FlowMark (the functions are described in alphabetical order):

Create

Creates a new note (in the database that was opened before by the DBOpen function) and returns the ID of this note (document). Furthermore, the values of the variables in the FlowMark input container are read and then inserted into items (with the same name as the FlowMark variables) in the new note.

This function requires the following variables:

NoteID

Represents the name of the FlowMark variable in the output container to which the returned ID is written. This container variable must be of type long.

This variable can also be used as command line parameter.

Form

Represents the name of the FlowMark variable in the input container. The value of this variable specifies a Form defined in the Lotus Notes database and is checked against the Form definitions in the already opened Lotus Notes database. If the Form name is not found, an error message is written to the log file.

This container variable must be of type string and must be specified exactly as shown (case-sensitive).

CreateEncryptable

CreateEncryptable works like the Create function with the addition that the created items have an *encryption flag* set. Later on these items can be encrypted with the Encrypt function.

This function requires the following variables:

NoteID

Represents the name of the FlowMark variable in the output container to which the returned ID is written. This container variable must be of type long.

This variable can also be used as command line parameter.

Form

Represents the name of the FlowMark variable in the input container. The value of this variable specifies a Form defined in the Lotus Notes database and is checked against the Form definitions in the already opened Lotus Notes database. If the Form name is not found, an error message is written to the log file.

This container variable must be of type string and must be specified exactly as shown (case-sensitive).

DBCclose

Closes a Lotus Notes database that has been opened with the DBOpen function.

DBOpen

Opens a database on a Lotus Notes server. This function must be used before any other service function (for example, Create or Delete) can be executed.

This function requires the following variables:

SBLN_Server_Name

Represents the name of the FlowMark variable in the input container that contains the name of the Lotus Notes server in the FlowMark input container. If a local database is used, set the value of this variable to LOCAL. This container variable must be of type string.

SBLN_Path_Name

Represents the name of the FlowMark variable in the input container that contains the path for the Lotus Notes database that is located on the specified Lotus Notes server. The path specifies the directory where the database is located, if it is a subdirectory of the Lotus Notes data directory, and the name of the database.

For example, if the Lotus Notes data directory is D:\NOTES\DATA, the name of the database to be opened is HELP.NSF, and the database is located in the directory D:\NOTES\DATA\DOC, the variable must be set to DOC\HELP.NSF.

This container variable must be of type string.

Decrypt

Decrypts the note with the given note ID.

This function requires the following variable:

NoteID

Represents the name of the FlowMark variable in the input container that specifies the note ID. This container variable must be of type long.

This variable can also be used as command line parameter.

Delete

Deletes the note with the given note ID.

This function requires the following variables:

NoteID

Represents the name of the FlowMark variable in the input container that contains the note ID. This container variable must be of type long.

This variable can also be used as command line parameter.

Encrypt

Encrypts the note with the given note ID. That means that all items which are encryption enabled (with CreateEncryptable or UpdateEncryptable) are sealed with the user's public key.

This function requires the following variables:

NoteID

Represents the name of the FlowMark variable in the input container that contains the note ID. This container variable must be of type long.

This variable can also be used as command line parameter.

Read

Reads the note with the given note ID. The values of the items in the specified document are then set into the variables of the FlowMark output container (these variables must have the same name as the items which have to be read).

This function requires the following variables:

NoteID

Represents the name of the FlowMark variable in the input container that contains the note ID. This container variable must be of type long.

This variable can also be used as command line parameter.

SearchDoc *SelectionFormula ViewName NoteID*

Searches for information that is stored in the Lotus Notes database. The search is based on formulas as they are used in Lotus Notes Views such as **SelectionFormula**.

This function requires the following variables:

SelectionFormula

Represents the name of the FlowMark variable in the input container that contains the search formula.

This container variable must be of type string.

ViewName

Represents the name of the FlowMark variable in the input container that contains the name of a Lotus Notes View in the Lotus Notes database.

This container variable must be of type string.

The specified View is used as a template for the selection dialog to display the document contents in a structured way.

If no document is found or the number of documents found is greater than 256, an error message is displayed and you can make the selection formula more specific.

If one document is found, the note ID of that document is written into the variable *NoteID* and is passed to a FlowMark output container.

If the number of documents found is from 2 to 256, the selection dialog is displayed. The selection dialog shows the contents of one document per line. Select one of the documents shown. The note ID of the selected document is written into the variable *NoteID* and is passed to a FlowMark output container.

NoteID

Represents the name of the FlowMark variable in the output container to which the returned ID is written. This container variable must be of type long.

This note ID can be used as input for other requester functions.

Sign

Signs the note with the given note ID. The note is signed with the user ID of the Lotus Notes client on the workstation.

This function requires the following variable:

NoteID

Represents the name of the FlowMark variable in the input container that contains the note ID. This container variable must be of type long.

This variable can also be used as command line parameter.

SignedOrEncrypted

Checks if the note with the given note ID is signed or encrypted and writes the return value into the variable *Result*.

This function requires the following variables:

NoteID

Represents the name of the FlowMark variable in the input container that contains the note ID. This container variable must be of type long.

This variable can also be used as command line parameter.

Result

Represents the name of the FlowMark variable in the output container to which the return value is written.

Possible return values are:

- | | |
|----------|------------------------------|
| 0 | Not signed and not encrypted |
| 1 | Signed and not encrypted |
| 2 | Not signed and encrypted |
| 3 | Signed and encrypted |

This container variable must be of type string.

Unsign

Unsigns the note with the given note ID. Only the original signer can unsign the note.

This function requires the following variable:

NoteID

Represents the name of the FlowMark variable in the input container that contains the note ID. This container variable must be of type long.

This variable can also be used as command line parameter.

Update

Updates the note with the given note ID. The values from the variables in the FlowMark input container are read and then inserted into items (with the same name like the FlowMark variables) in the note.

This function requires the following variable:

NoteID

Represents the name of the FlowMark variable in the input container that contains the note ID. This container variable must be of type long.

This variable can also be used as command line parameter.

Note: The FlowMark variable *NoteID* is not mapped into the note.

UpdateEncryptable

UpdateEncryptable works like the Update function with the addition that the updated items have an *encryption flag* set. Later on, these items can be encrypted with the Encrypt function.

This function requires the following variable:

NoteID

Represents the name of the FlowMark variable in the input container that contains the note ID. This container variable must be of type long.

This variable can also be used as command line parameter.

Type mapping

In the service functions, the following type mappings are valid:

FlowMark	-->	Lotus Notes	Lotus Notes	-->	FlowMark
String	-->	Text	Text	-->	String
String[]	-->	Textlist	Textlist	-->	String[]
Float	-->	Number	Number	-->	Float/Long/String
Long	-->	Number	Timedate	-->	String

It is possible to enforce an explicit type mapping from FlowMark to Lotus Notes with the following technique:

If the name of a FlowMark container variable of type string starts with a character defined below (not case-sensitive) followed by two underscores (_), these conversions are made:

D__	-->	Timedate
N__	-->	Number
T__	-->	Text
L__	-->	Textlist (first element)
A__	-->	Textlist (append element)

For an example for type mapping, refer to the sample FDL file for Lotus Notes (EXMP3SLN.FDL) that is provided with the product. The samples are located in the subdirectory \SAMPLES\LNOTES of the Service Broker Manager directory.

Chapter 5. FlowMark—VisualAge integration

This chapter describes the interface code that enables you to integrate VisualAge applications with FlowMark. This interface code is also known as the service broker for VisualAge. It is available for OS/2.

The service broker for VisualAge consists of:

- A service broker library for VisualAge applications compliant with the Service Broker Manager architecture (see “Service broker library” on page 55)
- VisualAge parts that provide the broker functionality to a VisualAge application (see “VisualAge parts for the service broker”)
- VisualAge parts that enable VisualAge applications to use the FlowMark API (see “VisualAge parts for the FlowMark C language API” on page 52)
- Programming examples (see “VisualAge broker programming examples” on page 53)

Installing the VisualAge source code

The VisualAge code is copied to the appropriate VisualAge directories during installation.

To make this code available in VisualAge, you must load it into the VisualAge library using the **Load Features** command from the System Transcript window. The name of the installable feature for the FlowMark interface code is *FlowMark Base*.

Loading this feature adds the following applications to your image:

- FmApiBase
- FmEditApiBase
- FmServiceBroker
- FmEditServiceBroker
- FmArchivalCode

VisualAge parts for the service broker

Several VisualAge parts are provided with this package that enable you to build VisualAge applications that interact with the Service Broker Manager.

These parts are provided in application FmServiceBroker. In addition, a related edit application named FmEditServiceBroker is provided, containing the code required to use these parts in the composition editor.

The provided parts can be split into these groups:

- Parts that are to be used in the composition editor (see “Function Data” on page 64)
- Template parts to be used to inherit from when creating new parts (see “FmBrokerTemplate” on page 66)
- Support parts that are used internally to provide the service broker functionality (see “FmBrokerWindow” on page 67 and “FmFunctionData” on page 68)

These parts are described in “Parts provided with the package” on page 64.

VisualAge parts for the FlowMark C language API

VisualAge parts provided with this package enable you to use the FlowMark C language API from VisualAge applications.

These parts are provided in application FmApiBase. In addition, a related edit application named FmEditApiBase is provided, containing the code required to use these parts in the composition editor.

The provided parts can be split into these groups:

- Parts that are to be used in the composition editor:
 - Current Activity (see “Current Activity” on page 68)
 - Input Container (see “Input Container” on page 73)
 - Output Container (see “Output Container” on page 75)
 - FlowMark Session (see “FlowMark Session” on page 78)
 - FlowMark Activity (see “FlowMark Activity” on page 79)
 - FlowMark Process (see “FlowMark Process” on page 82)
- Support parts that are used internally to access the FlowMark C language API:
 - FmApiLibrary (see “FmApiLibrary” on page 86)
 - FmError (see “FmError” on page 88)
 - FmContainer (see “FmContainer” on page 89)
 - FmInputContainer (see “FmInputContainer” on page 90)
 - FmOutputContainer (see “FmOutputContainer” on page 91)
 - FmContainerItem (see “FmContainerItem” on page 92)
 - FmStartDataItem (see “FmStartDataItem” on page 93)
 - FmExmApiBegin (see “FmExmApiBegin” on page 94)
 - FmExmApiTypeInfo (see “FmExmApiTypeInfo” on page 95)
 - FmExmApiStructureData (see “FmExmApiStructureData” on page 96)

For most applications, the public interface of the parts available in the FlowMark category in the composition editor should be sufficient to interface with FlowMark.

There may be special requirements, however, that make it necessary to use the provided support classes directly to interface with FlowMark or even to extend their functionality or to create new parts from them.

Keep in mind that you must not change any of these classes to keep the parts in the FlowMark category consistent. If you need to change anything, you should create your own subclasses and implement the changed functionality there.

VisualAge broker programming examples

Programming examples are distributed to demonstrate the usage of the provided code when creating a VisualAge application integrated with FlowMark.

Installing the programming examples

Note: Before you install the programming examples, make sure that you have installed the FlowMark—VisualAge interface code (see “Installing the VisualAge source code” on page 51).

Installation of the programming examples consists of several steps:

1. Install the VisualAge example code.

The VisualAge example code provided with this package is copied to the appropriate VisualAge directories.

To make this code available in VisualAge, you must load it into the VisualAge library via the **Load Feature** command from the System Transcript window. The name of the installable feature for the programming examples is *FlowMark Samples*.

Loading this feature adds the following applications to your image:

- FmBrokerSamples
- FmSamplesArchivalCode

2. Create the VisualAge runtime image.

To run the FlowMark example process provided with this package, you need to create a VisualAge runtime image that contains the example application. To create this image, do the following:

- a. In the VisualAge organizer, select application FmBrokerSamples and then invoke menu item **Make executable**.
- b. When prompted for the initial view to be displayed, select **<None>**.
- c. Store the runtime image as file EXMP3VSM.IMG in the EXM\BIN directory.

Besides the views that implement the activities contained in the example process, the example application contains an additional view, FmMaintainProcess, which can be used to create and start instances of the FlowMark example process. This function can be run outside of FlowMark. You can start it from within VisualAge by issuing the command **FmMaintainProcess newPart openWidget**, or you can create another runtime image and select this view as the initial view to be opened.

3. Install the FlowMark example process.

To install the FlowMark example process, log on to a FlowMark Buildtime client, double-click on the Import icon, and select file EXMP3VSM.FDL from the \EXM\SBM\SAMPLES\VISAGE directory. This creates the example process VisualAge Sample in your Buildtime Processes folder. Translate this process with option **Replace template** to create a Runtime template with the same name.

4. Register the example application in the Service Broker Manager.

To register the example application in the Service Broker Manager, open the Service Broker Manager, select **Create another broker from file**, and select file EXMP3VSM.INI from the \EXM\SBM\SAMPLES\VISAGE directory. This creates the broker and service definitions required for the example application.

FlowMark example process

A FlowMark example process that is provided with this package makes use of the VisualAge example classes.

When you install file EXMP3VSM.FDL, the following items are added to your FlowMark database:

- Process VisualAge Sample
- Program VisualAge Display Activity
- Program VisualAge Maintain Container
- Program VisualAge Maintain Data
- Data structure VisualAge Sample

Process VisualAge Sample consists of the activities Maintain Container and Check Container (performed by tool VisualAge Maintain Container) and Maintain Data (performed by tool VisualAge Maintain Data). All three activities have VisualAge Display Activity as a support tool.

The process and the contained activities use VisualAge Sample as data structure.

When you start the process, you are prompted for input data. Then, activities Maintain Container and Maintain Data are put on your work list. Both activities display the input container (which contains the data values that you entered when starting the process), and allow you to set values for the output container.

When you have finished both activities, Check Container appears on your work list. When you open the activity and select radio button **Input**, you can see that the input container of this activity contains the values that you entered in the output container of the previous activities (in fact, the values that you entered in the activity that you performed last).

Invoke support tool Display Activity while you are working on any of the activities. This results in a window showing some general information on the activity.

VisualAge example code

Four visual parts contained in application FmBrokerSamples show you how to use the interface code to build VisualAge applications integrated with FlowMark.

One part, FmMaintainProcess (see “FmMaintainProcess” on page 97), shows how to use the FlowMark process API. The other parts show how to create views that interact with the service broker and access the FlowMark container API. These parts have been developed according to the rules stated in service broker requirements:

- FmDisplayActivity (see “FmDisplayActivity” on page 98)
- FmMaintainContainer (see “FmMaintainContainer” on page 99)
- FmMaintainData (see “FmMaintainData” on page 100)

Service broker library

A dynamic link library, EXMP3VBR.DLL, is used both as service broker library and as service library for VisualAge applications in the Service Broker Manager.

Besides the standard functions required by the Service Broker Manager, this library contains a service function, InvokeClass, which is used to open a specific view in the brokered VisualAge applications.

InvokeClass

Service function InvokeClass is used to open a specific view in an active VisualAge application. The parameters for this function are:

sessionId class [parameterString]

Where:

<i>sessionId</i>	Is the identifier of the FlowMark session that the view is started from.
<i>class</i>	Is the name of the class that implements the view to be opened.
<i>parameterString</i>	Is an optional parameter string that is to be passed to the opened view.

Notes:

1. If you want to invoke this service function from a FlowMark activity, use the FlowMark requester library provided with the Service Broker Manager called EXMP3FRQ.DLL. In this case, the `sessionId` parameter is inserted automatically by the requester. See "Registering a VisualAge application" for details on how to specify the FlowMark program registration.
2. If you want to invoke this service function from outside of FlowMark, use the standard service requester, EXMP3FFR.EXE. In this case, you have to specify a dummy argument in place of the `sessionId` parameter.

The following sample command invokes view AnyView in the VisualAge application *myapp*:

```
exmp3ffr myapp myapp InvokeClass "0 AnyView"
```

Where *myapp myapp* are the logical names of the service broker and the service.

Note: Views that are opened this way do not have access to the FlowMark container API and can access the FlowMark process API only if they establish a new FlowMark session.

Developing VisualAge applications for FlowMark

The following sections describe what you need to consider when developing VisualAge applications for FlowMark.

Registering a VisualAge application

To integrate your VisualAge application with FlowMark using the Service Broker Manager architecture, several definitions are required both in FlowMark and in the Service Broker Manager.

Service Broker Manager definitions

In the Service Broker Manager, a service broker has to be created for every VisualAge image that you want to run with the Service Broker Manager. The name under which you register each service broker must match the name of the image file that contains your VisualAge application, without the extension IMG. This is required to allow the service broker DLL to load the correct image when it receives a start request from the Service Broker Manager.

The service broker issues the following command to load a VisualAge application:

```
exmp3vnd.exe -ibroker-name.img
```

The DLL name for the service broker library is EXMP3VBR.DLL and must be the same for each instance of the VisualAge service broker.

Notes:

1. All VisualAge images that are to be loaded by the service broker must reside in the working directory of the Service Broker Manager.
2. The service broker uses file EXMP3VND.EXE as a startup executable file for all VisualAge applications. This file is a copy of the file NODIALOG.EXE provided with VisualAge. If you want a product logo screen to be displayed at application startup, you can either replace this file with a copy of file ABT.EXE, or you can create your own startup executable, as described in the VisualAge documentation.

For each VisualAge service broker you have defined, you must also define the service library. The name you use for the service definition is not significant. The DLL name for the service library is also EXMP3VBR.DLL.

FlowMark definitions

If you register a program that intends to invoke a function via VisualAge service broker, you have to specify:

Field name	Value
Path and file name	EXMP3FRQ.DLL (standard requester)
Entry point	FMRequest, used as entry point in EXMP3FRQ.DLL
Command-line parameters	<i>broker service</i> InvokeClass <i>class-name parameters</i>
	Where:
<i>broker</i>	Is the logical name you used when defining the service broker library.
<i>service</i>	Is the logical name you used when defining the service library.
<i>class-name</i>	Is the name of the application class in your VisualAge image that provides the desired function.
<i>parameters</i>	Is an optional parameter string that is to be passed to the function.

Requirements for VisualAge applications

To make use of the Service Broker Manager architecture and the VisualAge service broker, a VisualAge application has to meet the following requirements:

1. Start the service broker at application startup:

When the VisualAge image is loaded, the service broker must be started. This is done automatically by the `runtimeStartUp` method of the application `FmServiceBroker`. Thus, when you package your application, you need not select any particular view to be initially displayed.

Note: If you plan to create a reduced runtime image from your application, you have to provide a `packagerIncludeClasses` method for each of your applications included in the image that returns an array of all view classes contained in the application. This is necessary to prevent VisualAge from removing these classes from the image (your view classes are not referenced in the image, since they are invoked dynamically from FlowMark).

In the provided sample application, a class method is provided for `FmBrokerSamples` containing the following:

```
packagerIncludeClasses
  ^Array with: FmDisplayActivity with: FmMaintainContainer
    with: FmMaintainData
```

2. Split the application into separately accessible function parts:

Each functional part of the application that is to be accessible via a service broker request must be represented by an application class. Each of these classes must implement an `openOn:` method accepting an instance of class `FmFunctionData` as argument. This object can be used by the application to obtain the FlowMark session identifier and the invocation parameters for the function and to set the return code of the function.

When the application function ends, it must send message `signal` to the function data object to indicate the end of the function to the service broker.

When you implement your application classes as visual parts in VisualAge, you can meet these requirements by:

- Defining an `openOn` event that has one parameter of type `FmFunctionData`. Do this as follows:
 - a. In the public interface editor, select **Event**.
 - b. Specify **openOn** as event name and select **Add**.
 - c. Click mouse button 2 on **Parameters and their types**.
 - d. As name, you can enter any name, for example, `functionData`. Enter **FmFunctionData** as class.

- Implementing an `openOn:` method that just signals event `openOn` with the received input parameter. Do this as follows:
 - a. In the script editor, select **New Method Template** from **Methods**.
 - b. Specify:


```
openOn: name
self signalEvent: #openOn with name
```

 Where *name* is the parameter name for the `openOn` event.
 - c. From **File**, select **Save Part**.
- Including a Function Data part from the FlowMark category in the composition editor.
- Connecting the `openOn` event of your application to the initialize action of the Function Data part and to the `openWidget` action of your window part.

To connect the event to the initialize action, do the following:

- a. In the composition editor, point to the free-form surface and click mouse button 2.
- b. From the popup menu, select **Connect** then **More**.
- c. Select event `openOn`.
- d. Click mouse button 2 on Function Data.
- e. Select **Initialize**.

To connect the event to the `openWidget` action, do the following:

- a. Place a Current Activity on the desktop.
- b. Click mouse button 2 on **Current Activity**.
- c. Select **Connect**.
- d. Select **Initialized** and connect it with **Window**.
- e. Select **openWidget**.

- Connecting the `closedWidget` event of your window part to the signal action of the Function Data part.

Part `FmBrokerTemplate` in application `FmServiceBroker` is provided as a template that contains all of the described elements.

Note: When designing and coding a VisualAge application that uses the Service Broker Manager, consider the following:

- The Service Broker Manager creates multiple threads for each service broker. Thus, it is possible that more than one of your application functions is invoked at the same time or even that one function is simultaneously invoked twice. Thus, be careful when using global or class variables.
- From a FlowMark point of view, your application is invoked via an entry point in a DLL. Therefore, you have to provide the FlowMark session identifier whenever you issue a FlowMark API call.

Accessing the FlowMark C container API

The provided VisualAge parts for the FlowMark C language API enable you to make full use of the FlowMark C container API from VisualAge applications. The use of these parts varies strongly, depending on whether you already know the data structure of the FlowMark activity when you build your VisualAge application.

Accessing the container when you know the structure

To access the FlowMark container, include the following parts from the FlowMark category in the composition editor view of your application: current activity, input container, and output container. The steps are these:

1. Initialization
 - a. To initialize the Current Activity part, do one of the following:
 - Connect event `sessionId` of a Function Data part to the `initForSession` action of the Current Activity part if your application uses the VisualAge service broker.
 - Connect event `aboutToOpenWidget` of your window part to the `initContainers` action of the Current Activity if your application does *not* use the VisualAge service broker.
 - b. To initialize the container parts, connect the `inputContainer` and `outputContainer` events of the Current Activity part to the `setContainer` action of the corresponding container parts.
 - c. To display meaningful error messages in case of error conditions returned by the FlowMark C language API, you can create the following connections:
 - Event `initError` of part Current Activity to action `displayInitError` of part Current Activity
 - Event `getError` of part Input Container to action `displayGetError` of part Current Activity
 - Event `setError` of part Output Container to action `displaySetError` of part Current Activity

2. Accessing the FlowMark data

Open the Settings view for your container parts, and define the data structure of the corresponding FlowMark container. You need not define all data structure members that exist in the FlowMark container if you plan to use only a subset.

To get access to the data structure members in a FlowMark container, do the following:

- a. In FlowMark, export the FDL.
- b. In VisualAge, click mouse button 2 on the icon for the input or output container.
- c. Select **Open Settings**.
- d. Select **Import**.

When you define the data structure, the public interface of the container parts is dynamically changed to include attributes for all data items that you specify. Use these attributes to access and update the data in the FlowMark container.

If you want to update data in the FlowMark output container, do not forget to connect the `setItems` action of the part Output Container.

Accessing the container when you do not know the structure

If you do not know the data structure of the FlowMark container when you build your activity, you cannot use the Input Container and Output Container parts, as described in “Accessing the container when you know the structure” on page 60. However, you can still use the full FlowMark C container API.

To access the FlowMark container, include a Current Activity part from the FlowMark category in the composition editor. The steps are:

1. Initialization
 - a. To initialize the Current Activity part, do one of the following:
 - Connect event `sessionId` of a Function Data part to the `initForSession` action of the Current Activity part if your application uses the VisualAge service broker.
 - Connect event `aboutToOpenWidget` of your window part to the `initContainers` action of the Current Activity if your application does *not* use the VisualAge service broker.
 - b. To display meaningful messages to the user in case of error conditions returned by the FlowMark API, you should draw the following connections within the Current Activity part:
 - Event `initError` to action `displayInitError`
 - Event `getError` to action `displayGetError`
 - Event `setError` to action `displaySetError`

Note: Part `FmBrokerTemplate` is provided as a template that contains all of the described elements.

2. Accessing the FlowMark data

You can use the Current Activity part for the following tasks:

To get the value of a data item from FlowMark, draw a connector to action `getContaineritem`. You must specify the name of the data item that is to be gotten as a parameter to the connector. The result of the connection is the value of the data item and may be connected, for example, to a text field in a window part.

- Update data in the FlowMark output container

To update the value of a data item in FlowMark, draw a connector to action `setContainerItem`. You must specify the name of the data item to be updated, and the new value for it, as parameters to the connector.

To copy the value of a data item from the input container to the output container, you can use actions `copyContainerItem` and `copyItemToNewName`.
- Query the structure of the FlowMark container

Several actions are provided to obtain information about the structure of the FlowMark containers:

 - The actions `inputMembers` and `outputMembers` return a collection containing the names of all data structure members.
 - The actions `inputItems` and `outputItems` return a collection containing the names of all data items.
 - The actions `typeOfInputItem` and `typeOfOutputItem` return the type of a data structure member or a data item.
 - The actions `cardinalityOfInputItem` and `cardinalityOfOutputItem` return the cardinality of a data structure member.

Accessing the FlowMark C process API

The provided VisualAge parts for the FlowMark API allow you to make full use of the FlowMark process API from VisualAge applications.

To access the FlowMark process API, you first need to establish a FlowMark process control session. This can be achieved by including a FlowMark Session part from the FlowMark category in the composition editor.

There are different methods for using this part, depending on the type of your application:

- If your application is invoked via the VisualAge service broker, start the process control session by connecting event `sessionId` of a Function Data part to the `startForSessionId` action of the FlowMark Session part.
- If your application is invoked by FlowMark as a stand-alone program, without the VisualAge service broker, connect the `aboutToOpenWidget` event of your application to the `startForSessionId` action of the FlowMark Session part.
- If your application is not invoked by FlowMark, you have to start an external process-control session, which includes a logon to FlowMark. You can achieve this by either of the following methods:
 - Using action `startExternal` of the FlowMark Session part, which prompts the user for the required logon values before starting the process-control session.
 - Using action `startExternalWith` of the FlowMark Session part, and providing the required logon values in form of an `FmExmApiBegin` structure as parameter, if your application already has the logon parameters for FlowMark.

To display meaningful messages to the user in case of error conditions returned by the FlowMark C language API, you should draw the following connections within the FlowMark Session part:

- Event `startError` to action `displayStartError`
- Event `closeError` to action `displayCloseError`

Now that your FlowMark Session represents a FlowMark process control session, you can add FlowMark Process parts to perform process control functions.

For each FlowMark Process part that you include in your application, connect the `setHandle` action to the `started` event of the FlowMark Session.

You can specify the name of the FlowMark process that is to be manipulated either in the Settings view of the FlowMark Process part (if you already know the name when you build the application), or by drawing connections to the `processName` attribute of the part, if the name is to be dynamically set at run time.

If you want to use the FlowMark Process part to start a new process instance in FlowMark, you also have to specify a template name and, optionally, the input data for the process.

The template name can be entered either in the Settings view or via connections to the `templateName` attribute. The input data for the process can be set either in the Settings view or via connections to the `addItemValue` or `addItemData` actions.

You can use the FlowMark Process part to start new processes or to suspend, resume, terminate, or restart existing processes.

To display meaningful messages to the user in case of error conditions returned by the FlowMark C language API, you should draw the following connection within each FlowMark Process part:

- Event `processError` to action `displayError`

Testing your VisualAge application

To be able to test the views of your application that are to be invoked by the Service Broker Manager without having to generate a runtime image, use the following procedure (if your view is not invoked from a FlowMark activity, you need to perform steps 5 on page 64 and 6 on page 64 only):

1. Define and translate a FlowMark process that invokes your view (see “Registering a VisualAge application” on page 56 for details on how to register your view to FlowMark).
2. Start FlowMark runtime.
3. From the FlowMark worklist, start the activity that invokes the view you want to test. However, do not start the Service Broker Manager.

4. This results in a message box indicating that the requested service is not available. Near the bottom of that message box, you see a heading Input: followed by some text (actually, the FlowMark session identifier, the name of the view to be invoked, and maybe some input parameters that are to be passed to your view). Do not close the message box.
5. Enter the following code in some VisualAge window:

```

| data |
data := FmFunctionData session: sessionId parms: parmString hab: 0.
ViewName new openOn: data.
[ data wait.
  CwMessagePrompter message: 'Return code is ',
  data returnCode printString ] fork

```

Replace *sessionId* with the FlowMark session identifier from the message box (specify '0' if your view is not to be invoked from FlowMark). For *parmString*, specify the string of parameters you want to pass to your view, or String new if your view does not require any input parameters. Finally, replace *ViewName* with the name of the VisualAge class that implements your view.

6. Execute the code. This should give you the following result:
 - Display your view with full access to the FlowMark data container.
 - When you close the view, display a message box with the return code of your function on it.

Note: If you do not get a message box after you close your view, the end of your function is not correctly signaled. This would result in hanging activities in FlowMark if you execute the function in a runtime image. See “Requirements for VisualAge applications” on page 58 to find out how to signal the end of your function.
7. Close the message box that was displayed when you started the FlowMark activity from the worklist. If you want your activity to remain on the worklist to be able to repeat your test, and this is not automatically achieved by an exit condition (for example, `_RC = 0`), you should force restart of the activity from the FlowMark worklist before you close the message box.

Parts provided with the package

The following sections describe the parts that are provided with the package.

Function Data

Select **Function Data** to add a part that can be used as communication medium between the VisualAge service broker and the application functions invoked by it.

You can use this part to access the input parameters passed to your application function by FlowMark and to set the return code of your function that is to be passed back to FlowMark.

You must place a Function Data on the free-form surface outside the bounds of parts from the Canvas category, such as Window and Form parts.

Public interface:

- Actions (see “Function Data—Actions”)
- Events (see “Function Data—Events”)

Class name: FmBrokerData

Category: FlowMark

Function Data—Actions

initialize *functionData*

Initializes the part with the FmFunctionData object passed as parameter *functionData*. This action has to be invoked before any other action of the receiver is invoked. You can connect this action to the `openOn` event of your visual part if you derived or copied that part from FmBrokerTemplate.

Calling this action signals events `sessionId`, `parmString`, and `returnCode`.

signal

Signals the end of the function and returns control back to FlowMark. The return code passed to FlowMark is the one that has been set by the last invocation of action `setReturnCode`, or 0 if no other return code has been set.

Typically, this action is connected to the `closedWidget` event of your function’s main window.

signalWithCode *returnCode*

Signals the end of the function and returns control back to FlowMark. The return code passed back to FlowMark is the integer value specified by parameter *returnCode*.

Typically, this action is connected to events that indicate an error condition like the `initError` event of part FlowMark Activity.

setReturnCode *returnCode*

Sets the return code that is passed back to FlowMark when action `signal` is invoked to the integer value specified by parameter *returnCode*.

Function Data—Events

sessionId *session*

The part has been initialized. Parameter *session* has class String and contains the FlowMark session identifier for the invoked function.

You can connect this event to the `initForSession` action of part FlowMark Activity.

parmString *parameters*

The part has been initialized. Parameter *parameters* has class String and contains the command line parameters that are passed as input to the function by FlowMark.

returnCode *returnCode*

The return code that is to be passed back to FlowMark when action *signal* is invoked has been modified. Parameter *returnCode* has class Integer and contains the new return value.

FmBrokerTemplate

Part FmBrokerTemplate is provided as a template that you can use when building visual parts that are to be invoked as application functions by FlowMark.

You can use this part by:

- Specifying it as super class to inherit from when creating a new visual part
- Copying it to a new part
- Viewing it to find out which elements you need to include in your new part

The following functionality is provided by this template:

- Open the part via a service function request from the VisualAge service broker.
- Initialize the connection to FlowMark to be able to access the container data.
- Open the application window.
- Return control to FlowMark when the application window is closed.
- Display message boxes in case of non-zero return codes from the FlowMark API.

This functionality is achieved by the following elements:

- Instance method `openOn: functionData`
Signals event `openOn` with *functionData* as parameter. This method is invoked by the VisualAge service broker whenever it receives a function request for a class.
- Event `openOn functionData`
Indicates that the application function has been opened. The parameter *functionData* has class FmFunctionData.
- The required subparts and connections in the composition editor

Subparts:

- A Function Data used to communicate with the VisualAge service broker
- A Current Activity used to access the FlowMark data container

Connections:

Source	Event	Target	Action
FmBrokerTemplate	openOn	Function Data	initialize
Function Data	sessionId	Current Activity	initForSession
Current Activity	initialized	Window	openWidget
Window	closedWidget	Function Data	signal
Current Activity	initError	Function Data	signalWithCode
Current Activity	initError	Current Activity	displayInitError
Current Activity	getError	Current Activity	displayGetError
Current Activity	setError	Current Activity	displaySetError

FmBrokerWindow

Part FmBrokerWindow is a visual part that displays a control window showing information about the VisualAge service broker.

This part must be used as the initial view to be opened when creating VisualAge applications for use with the Service Broker Manager.

When this part is opened, it:

- Opens the service broker control window
- Establishes communication with the Service Broker Manager
- Waits for function requests from the Service Broker Manager

For each function request, this part:

- Accesses the shared memory area established by the service broker to retrieve the invocation parameters
- Extracts the FlowMark session identifier and the name of the class that is to be invoked from the parameter string
- Checks whether such a class exists in the image and whether it implements an openOn: method
- Creates an instance of class FmFunctionData initialized with all data concerning the function request
- Creates a new instance of the requested class and sends message openOn: with the FmFunctionData as parameter
- Waits for the application function to finish
- Signals the end of the application function and the return code to FlowMark

Note: This part is opened automatically when a VisualAge service broker is started from the Service Broker Manager and is closed when the service broker is stopped in the Service Broker Manager. You should not manually close this window except in cases where the Service Broker Manager has abnormally ended for some reason.

FmFunctionData

Class FmFunctionData implements the communication protocol between the VisualAge service broker and the application functions invoked by it.

An instance of this class is created whenever a service function request is received from the Service Broker Manager and is passed as argument on the openOn: message that is sent to the class implementing the requested function.

Objects of this class are used as parameter for the initialize action of part Function Data.

Your application should not directly use this class but rather interface with it via the public interface of the Function Data part.

Current Activity

Select **Current Activity** to add a part that can be used to access the FlowMark container data from your application.

You can use this part to query the structure of the input and output container of the activity in FlowMark that your application is started from, to query data from the input container, and to update data in the output container.

You must place a Current Activity on the free-form surface outside the bounds of parts from the Canvas category, such as Window and Form parts.

Public interface:

- Actions (see “Current Activity—Actions”)
- Events (see “Current Activity—Events” on page 72)

Class name: FmCurrentActivity

Category: FlowMark

Current Activity—Actions

initForSession *sessionId*

Initializes the part by establishing the connection to FlowMark and querying the structure of the FlowMark data containers. Parameter *sessionId* has class String and must contain the session identifier assigned to your application by FlowMark.

The part has to be initialized either with this action or with `initContainers` before any other action is invoked. You can connect `initForSession` to the `sessionId` event of a Function Data part to make sure that this part is initialized as soon as the Function Data is initialized.

Calling this action signals events `initialized` (together with `inputContainer` and `outputContainer`) or `initError`, depending on the success of the operation.

getContainerItem *item*

Gets the value for a data item from FlowMark. Parameter *item* has class String and contains the name of the data item that is to be retrieved.

The result of the action is the value of the data item and has class Integer, String, or Float. If an error occurs, event `getError` is signaled, and the result of the action is the default value for the type of the data item, or an integer 0 if the data item does not exist in the FlowMark data container.

setContainerItem *item, value*

Updates the value of a data item in the FlowMark output container. Parameter *item* has class String and contains the name of the data item that is to be set. Parameter *value* has class Integer, String, or Float, depending on the type of the data item in FlowMark, and contains the new value for the data item.

If an error occurs, event `setError` is signaled.

getOutputItem *item*

Gets the previously set value for a data item from the output container. Parameter *item* has class String and contains the name of the data item that is to be retrieved.

The result of the action is the value of the data item and has class Integer, String, or Float. If the data item has not been set yet, the default value for the data item is returned.

If the data item does not exist in the output container, event `getError` is signaled, and the result of the action is an integer 0.

copyContainerItem *item*

Copies the value of a data item from the FlowMark input container to the FlowMark output container. Parameter *item* has class String and contains the name of the data item that is to be copied.

If an error occurs, event `getError` or `setError` is signaled.

copyItemToNewName *inputItem, outputItem*

Copies the value of a data item from the FlowMark input container to a different data item in the FlowMark output container. Parameter *inputItem* has class String and contains the name of the data item that is to be copied. Parameter *outputItem* has class String and contains the name of the data item in the output container that is to receive the value.

If an error occurs, event `getError` or `setError` is signaled.

copyMatchingItems

Copies the values of all data items from the input container to the data items in the output container that have identical names.

If an error occurs, event `getError` or `setError` is signaled.

initContainers

Initializes the part by establishing the connection to FlowMark and querying the structure of the FlowMark data containers.

The part has to be initialized with either this action, or with `initForSession`, before any other action is invoked. If your application uses the Service Broker Manager architecture, you must use `initForSession` to initialize this part. Action `initContainers` is useful only for applications that are to be invoked directly from FlowMark activities.

Calling this action signals events `initialized` (together with `inputContainer` and `outputContainer`) or `initError`, depending on the success of the operation.

displayInitError *error*

Displays a message box describing an error that has occurred when attempting to connect to FlowMark and obtain the structure of the data containers. Parameter *error* has class `FmError` and contains an object describing the type of error that occurred.

You should connect this action to the `initError` event of this part.

displayGetError *item, error*

Displays a message box describing an error that has occurred when attempting to get the value of a data item from FlowMark. Parameter *item* has class `String` and contains the name of the data item that could not be gotten. Parameter *error* has class `FmError` and contains an object describing the type of error that occurred.

You should connect this action to the `getError` event of this part.

displaySetError *item, value, error*

Displays a message box describing an error that has occurred when attempting to set the value of a data item in the FlowMark output container. Parameter *item* has class `String` and contains the name of the data item that could not be set. Parameter *value* has class `Integer`, `String`, or `Float` and contains the value that could not be set for the data item. Parameter *error* has class `FmError` and contains an object describing the type of error that occurred.

You should connect this action to the `displaySetError` event of this part.

inputMembers

Returns an object of class `OrderedCollection` containing the names of all data structure members in the FlowMark input container.

outputMembers

Returns an object of class `OrderedCollection` containing the names of all data structure members in the FlowMark output container.

inputItems

Returns an object of class `OrderedCollection` containing the names of all data items in the FlowMark input container.

outputItems

Returns an object of class `OrderedCollection` containing the names of all data items in the FlowMark output container.

typeOfInputItem *item*

Gets the type of a data structure member in the FlowMark input container. Parameter *item* has class `String` and contains the name of the data structure member whose type is to be returned. If the parameter contains a data item name, it is translated to the related data structure member name internally.

The result of the action is a string describing the type of the data structure member. Possible values are `Long`, `String`, and `Float`, or `unknown` if the data structure member does not exist in the FlowMark container.

typeOfOutputItem *item*

Gets the type of a data structure member in the FlowMark output container. Parameter *item* has class `String` and contains the name of the data structure member whose type is to be returned. If the parameter contains a data item name, it is translated to the related data structure member name internally.

The result of the action is a string describing the type of the data structure member. Possible values are `Long`, `String`, and `Float`, or `unknown` if the data structure member does not exist in the FlowMark container.

cardinalityOfInputItem *item*

Gets the cardinality of a data structure member in the FlowMark input container. Parameter *item* has class `String` and contains the name of the data structure member whose cardinality is to be returned. If the parameter contains a data item name, it is translated to the related data structure member name internally.

The result of the action is an integer containing 0, if the data structure member holds a single data item, or the number of elements in the array described by the data structure member.

cardinalityOfOutputItem *item*

Gets the cardinality of a data structure member in the FlowMark output container. Parameter *item* has class `String` and contains the name of the data structure member whose cardinality is to be returned. If the parameter contains a data item name, it is translated to the related data structure member name internally.

The result of the action is an integer containing 0, if the data structure member holds a single data item, or the number of elements in the array described by the data structure member.

Current Activity—Events

initialized *activity*

The part has been successfully initialized. The connection to FlowMark is established, and the structure of the data containers has been obtained from FlowMark.

Parameter *activity* has class `FmCurrentActivity` and contains the part itself.

You should connect this event to the `openWidget` action of your application window to ensure that the window is opened only when the connection to FlowMark has been set up successfully.

inputContainer *input*

Always signaled together with event `initialized`.

Parameter *input* has class `FmInputContainer` and can be used to access the input data of the FlowMark activity.

If your application contains an Input Container part, you should connect this event to the `setContainer` action of that part to initialize it.

outputContainer *output*

Always signaled together with event `initialized`.

Parameter *output* has class `FmOutputContainer` and can be used to access the output data of the FlowMark activity.

If your application contains an Output Container part, you should connect this event to the `setContainer` action of that part to initialize it.

initError *error*

An error has occurred when attempting to connect to FlowMark and obtain the structure of the data containers. Parameter *error* has class `FmError` and contains an object describing the type of error that occurred.

You should connect this event to the `displayInitError` action of this part to display an error message to the user, and to action `signalWithCode` of a Function Data to end your application in case of an initialization error (do not forget to set the parameter for the `signalWithCode` action).

getError *item, error*

An error has occurred when attempting to get the value of a data item from FlowMark. Parameter *item* has class `String` and contains the name of the data item that could not be gotten. Parameter *error* has class `FmError` and contains an object describing the type of error that occurred.

You should connect this event to the `displayGetError` action of this part to display an error message to the user.

setError *item, value, error*

An error has occurred when attempting to set the value of a data item in the FlowMark output container. Parameter *item* has class String and contains the name of the data item that could not be set. Parameter *value* has class Integer, String, or Float and contains the value that could not be set for the data item. Parameter *error* has class FmError and contains an object describing the type of error that occurred.

You should connect this event to the displaySetError action of this part to display an error message to the user.

Input Container

Select **Input Container** to add a part that you can use to access the input data of the FlowMark activity in cases where you know the structure of the data already when you build the application.

You must place an Input Container on the free-form surface outside the bounds of parts from the Canvas category, such as Window and Form parts.

Public interface:

- Attributes (see “Input Container—Attributes”)
- Actions (see “Input Container—Actions” on page 74)
- Events (see “Input Container—Events” on page 74)

Settings:

- General (see “Input Container—Settings” on page 75)

Class name: FmInputCtnrPart

Category: FlowMark

Input Container—Attributes

dataMembers (*OrderedCollection*)

Contains the definition of the data structure used for the container. Each element of the collection has class FmContainerItem and describes one data structure member in the container.

When you specify items for the dataMembers collection in the settings view, additional attributes are dynamically generated for the part. For each data item defined by the data structure members specified, an attribute is created whose name is equal to the name of the data item. These attributes do not have set methods, so you cannot update their value. You can only draw connections to these attributes to obtain the values of these data items from the FlowMark input container.

If any of the connections you draw results in VisualAge to obtain the value of such an attribute before the container is initialized, the constant value **nil** is returned. As soon as the container is initialized, the value for the data item is obtained from FlowMark and stored in the attribute.

Input Container—Actions

setContainer *container*

Initializes the part with the initialized instance of class `FmInputContainer` passed as parameter *container*.

This action signals events `structureOk` or `structureMismatch`, depending on whether the data structure that has been defined for the part at build time matches the data structure of the FlowMark input container at runtime.

You should connect this action to the `inputContainer` event of a Current Activity part to ensure that the container is correctly set as soon as the activity is initialized.

When you specify items for the `dataMembers` collection in the settings view, additional actions are dynamically generated for the part. For each data item defined by the data structure members specified, an action is created whose name is built by concatenating the string *get* with the name of the data item.

The methods implementing the interface to the FlowMark container from this part make sure that data is obtained from FlowMark only when it is really needed (that means, when the corresponding attributes are connected to other objects). This can cause the problem that, if a value for a data item is required only to be used in a script (via an event-to-script connection), the value is not obtained from FlowMark, and the event is never signaled.

For these cases, the `get....` actions have been defined to force the retrieval of a data-item value from FlowMark.

Example:

If you want to use the value of the predefined FlowMark data item `_ACTIVITY` in a script that is to be run as soon as your window is opened, you need to draw the following connections:

1. From the `openedWidget` event of your window part to the `get_ACTIVITY` action of the input container part.
2. From the `_ACTIVITY` event of the input container part to the script that is to be run.

Input Container—Events

structureOk

During initialization of the part, it has been verified that all predefined data items really exist in the FlowMark input container.

structureMismatch

During initialization of the part, one or more predefined data items could not be found in the FlowMark input container. Any subsequent attempt to access one of these items results in the FlowMark error condition `ExmpjInvalidItemName`.

getError *item, error*

An error has occurred when attempting to get the value of a data item from FlowMark. Parameter *item* has class String and contains the name of the data item that could not be retrieved. Parameter *error* has class FmError and contains an object describing the type of error that occurred.

You can connect this event to the displayGetError action of a Current Activity part to display an error message to the user.

When you specify items for the dataMembers collection in the settings view, additional events are dynamically generated for the part. For each data item defined by the data structure members specified, an event is created whose name is built by concatenating the string *error* with the name of the data item.

Whenever the attempt to retrieve the value of a data item from FlowMark fails, the corresponding error... event is signaled, in addition to the common getError event. An object of class FmError describing the type of error that occurred is passed as parameter on these events.

Input Container—Settings Structure

Defines the data structure of the input container of the FlowMark activity. Click mouse button 2 somewhere on the table part to get a pop-up menu that you can use to modify the displayed list.

The values you enter are stored in attribute dataMembers.

You can use the **Import** button to import a data structure from an FDL file.

Output Container

Select **Output Container** to add a part that you can use to modify the output data of the FlowMark activity in cases where you know the structure of the data already when you build the application.

You must place an Output Container on the free-form surface outside the bounds of parts from the Canvas category, such as Window and Form parts.

Public interface:

- Attributes (see “Output Container—Attributes” on page 76)
- Actions (see “Output Container—Actions” on page 76)
- Events (see “Output Container—Events” on page 77)

Settings:

- General (see “Output Container—Settings” on page 77)

Class name: FmOutputCtnrPart

Category: FlowMark

Output Container—Attributes

dataMembers (*OrderedCollection*)

Contains the definition of the data structure used for the container. Each element of the collection has class `FmContainerItem` and describes one data structure member in the container.

When you specify items for the `dataMembers` collection in the settings view, additional attributes are dynamically generated for the part. For each data item defined by the data structure members specified, an attribute is created whose name is equal to the name of the data item.

The initial value of all these attributes is **nil**. When you update the value of these attributes, those values are not actually written to the FlowMark output container, but only stored in internal memory.

You need to invoke action `setItems` to move these values from internal storage to the FlowMark container.

Output Container—Actions

setContainer *container*

Initializes the part with the initialized instance of class `FmOutputContainer` passed as parameter *container*.

This action signals events `structureOk` or `structureMismatch`, depending on whether the data structure that has been defined for the part at build time matches the data structure of the FlowMark output container at runtime.

You should connect this action to the `outputContainer` event of a Current Activity part to ensure that the container is correctly set as soon as the activity is initialized.

setItems

Copies the values of all data items that have been stored internally via connections to the dynamically generated attributes to the output container of the FlowMark activity.

You can connect this action, for example, to the `aboutToCloseWidget` event of your application window to make sure that all data is saved to FlowMark when the window is closed.

When you specify items for the `dataMembers` collection in the settings view, additional actions are dynamically generated for the part. For each data item defined by the data structure members specified, an action is created whose name is built by concatenating the string *set* with the name of the data item.

You can use these methods instead of the corresponding attributes to copy a value to FlowMark immediately. You have to pass the new value for the container item as parameter. Invoking these action does not update the values of the corresponding attributes.

It is strongly recommended that you use the dynamically generated attributes, in conjunction with the *setItems* method, to move data to the FlowMark output container, rather than to use the *set...* methods, to ensure that the FlowMark API is invoked only once for each data item.

Output Container—Events

structureOk

During initialization of the part, it has been verified that all predefined data items really exist in the FlowMark output container.

structureMismatch

During initialization of the part, one or more predefined data items could not be found in the FlowMark output container. Any subsequent attempt to access one of these items results in a FlowMark error condition *ExmpjInvalidItemName*.

setError *item, value, error*

An error has occurred when attempting to set the value of a data item in the FlowMark output container. Parameter *item* has class *String* and contains the name of the data item that could not be set. Parameter *value* has class *Integer*, *String*, or *Float* and contains the value that could not be set for the data item. Parameter *error* has class *FmError* and contains an object describing the type of error that occurred.

You can connect this event to the *displaySetError* action of a Current Activity part to display an error message to the user.

When you specify items for the *dataMembers* collection in the settings view, additional events are dynamically generated for the part. For each data item defined by the data structure members specified, an event is created whose name is built by concatenating the string *error* with the name of the data item.

Whenever the attempt to set the value of a data item in FlowMark fails, the corresponding *error...* event is signaled, in addition to the common *setError* event. An object of class *FmError* describing the type of error that occurred is passed as parameter on these events.

Output Container—Settings

Structure

Defines the data structure of the output container of the FlowMark activity. Click mouse button 2 somewhere on the table part to get a pop-up menu that you can use to modify the displayed list.

The values you enter are stored in attribute *dataMembers*.

You can use the **Import** button to import a data structure from an FDL file.

FlowMark Session

Select **FlowMark Session** to add a part that represents a process control session in FlowMark Runtime to your application.

You can use this part to start and stop a process control session in FlowMark, which you need if you plan to invoke any of the process control functions defined in the FlowMark process API.

You must place a FlowMark Session on the free-form surface outside the bounds of parts from the Canvas category, such as Window and Form parts.

Public interface:

- Actions (see “FlowMark Session—Actions”)
- Events (see “FlowMark Session—Events” on page 79)

Class name: FmSession

Category: FlowMark

FlowMark Session—Actions

startForSessionId *sessionId*

Starts a process control session for an application that has been invoked under the control of FlowMark. Parameter *sessionId* has class String and must contain the session identifier that has been assigned to your application by FlowMark.

If your application has been invoked directly by FlowMark without the VisualAge service broker, you can specify **nil** as value for *sessionId*, which causes the part to obtain the current session identifier from FlowMark.

If the operation is successful, event *started* is signaled, with the session handle as parameter. If the operation fails, event *startError* is signaled.

startExternal

Starts a process control session for an application that has not been invoked under the control of FlowMark.

This action displays a logon dialog that prompts the user to enter the required parameters to start a FlowMark session and then logs on to FlowMark using the supplied values.

If the operation is successful, event *started* is signaled, with the session handle as parameter. If the operation fails, event *startError* is signaled.

startExternalWith *logonData*

Starts a process control session for an application that has not been invoked under the control of FlowMark. Parameter *logonData* has class FmExmApiBegin.

This action tries to log on to FlowMark using the logon parameters provided in *logonData*.

If the operation is successful, event *started* is signaled, with the session handle as parameter. If the operation fails, event *startError* is signaled.

endSession

Terminates the FlowMark process control session.

If the operation is successful, event `closed` is signaled. If the operation fails, event `closeError` is signaled.

displayStartError *error*

Displays a message box describing an error that has occurred when attempting to start a process control session. Parameter *error* has class `FmError` and contains an object describing the type of error that occurred.

You should connect this action to the `startError` event of this part.

displayCloseError *error*

Displays a message box describing an error that has occurred when attempting to close a process control session. Parameter *error* has class `FmError` and contains an object describing the type of error that occurred.

You should connect this action to the `closeError` event of this part.

FlowMark Session—Events**started** *handle*

A FlowMark process control session has been started. Parameter *handle* has class `LargeInteger` and contains the handle of the FlowMark session.

closed

The FlowMark process control session has been terminated.

startError *error*

An error has occurred when attempting to start a FlowMark process control session. Parameter *error* has class `FmError` and contains an object describing the type of error that occurred.

You should connect this event to the `displayStartError` action of this part to display an error message to the user.

closeError *error*

An error has occurred when attempting to close the FlowMark process control session. Parameter *error* has class `FmError` and contains an object describing the type of error that occurred.

You should connect this event to the `displayCloseError` action of this part to display an error message to the user.

FlowMark Activity

Select **FlowMark Activity** to add a part that represents an activity in FlowMark Runtime to your application. You can use this part to change the status of the activity in FlowMark.

You must place a FlowMark Activity on the free-form surface outside the bounds of parts from the Canvas category, such as Window and Form parts.

Public interface:

- Attributes (see “FlowMark Activity—Attributes”)
- Actions (see “FlowMark Activity—Actions”)
- Events (see “FlowMark Activity—Events” on page 81)

Settings: General (see “FlowMark Activity—Settings” on page 81)

Class name: FmActivity

Category: FlowMark

FlowMark Activity—Attributes

processName (*String*)

The name of the FlowMark Runtime process instance that contains the activity that is to be manipulated by this part.

activityName (*String*)

The fully qualified name of the activity in FlowMark that is to be manipulated by this part.

FlowMark Activity—Actions

setHandle *handle*

Assigns the handle of an active FlowMark process control session to the part. Parameter *handle* has class `LargeInteger`.

A process control session handle has to be assigned to the part using this action before the status of the associated activity can be changed. You can connect `setHandle` to the `started` event of a FlowMark Session part to make sure that the handle is available as soon as the process control session is started.

start

Starts the activity in FlowMark Runtime.

This action works only if the current status of the activity in FlowMark is `ready`.

If the operation is successful, event `running` is signaled. If the operation fails due to an invalid status of the activity, event `wrongState` is signaled. If the operation fails for any other reason, event `apiError` is signaled with a parameter indicating the type of the error that occurred.

restart

Restarts the activity in FlowMark Runtime.

This action works only if the current status of the activity in FlowMark is `running` or `pending`.

If the operation is successful, event `running` is signaled. If the operation fails due to an invalid status of the activity, event `wrongState` is signaled. If the operation fails for any other reason, event `apiError` is signaled with a parameter indicating the type of the error that occurred.

finish

Finishes the activity in FlowMark Runtime.

This action works only if the current status of the activity in FlowMark is ready, running, or pending.

If the operation is successful, event `finished` is signaled. If the operation fails due to an invalid status of the activity, event `wrongState` is signaled. If the operation fails for any other reason, event `apiError` is signaled with a parameter indicating the type of the error that occurred.

displayError *error*

Displays a message box describing an error that has occurred when attempting to change the activity status. Parameter *error* has class `FmError` and contains an object describing the type of error that occurred.

You should connect this action to the `apiError` event of this part.

FlowMark Activity—Events**running**

The activity has been successfully started or restarted.

finished

The activity has been successfully finished.

wrongState *state*

The activity status could not be changed due to an invalid current status of the activity. Parameter *state* has type `Integer` and contains the current status of the activity.

apiError *error*

An error has occurred when attempting to change the status of the activity. Parameter *error* has class `FmError` and contains an object describing the type of error that occurred.

You should connect this event to the `displayError` action of this part to display an error message to the user.

FlowMark Activity—Settings**Process**

The name of the FlowMark Runtime process instance that contains the activity that is to be manipulated by this part.

The value you enter is stored in attribute `processName`.

Activity

The fully qualified name of the FlowMark activity that is to be manipulated by this part.

The value you enter is stored in attribute `activityName`.

FlowMark Process

Select **FlowMark Process** to add a part that represents a process instance in FlowMark Runtime to your application.

You can use this part to invoke the process control functions defined in the FlowMark process API, like starting a new process from a process template or suspending an active process.

You must place a FlowMark Process on the free-form surface outside the bounds of parts from the Canvas category, such as Window and Form parts.

Public interface:

- Attributes (see “FlowMark Process—Attributes”)
- Actions (see “FlowMark Process—Actions” on page 83)
- Events (see “FlowMark Process—Events” on page 85)

Settings:

- General (see “FlowMark Process—Settings” on page 86)

Class name: FmProcess

Category: FlowMark

FlowMark Process—Attributes

processName (*String*)

The name of the FlowMark Runtime process instance that is to be manipulated by this part. If a new process instance is created, the default name that is to be used for the new instance.

templateName (*String*)

If the part is to be used to create a new FlowMark process instance, this is the name of the process template that is to be used to create the instance from.

startData (*OrderedCollection*)

If the part is to be used to create a new FlowMark process instance, this is the input data that is passed to the new process. Each element in the collection must be an instance of class FmStartDataItem.

When you specify items for the startData collection in the settings view, additional attributes are dynamically generated for the part. For each data item specified, an attribute is created whose name is equal to the name of the data item. You can draw connections to these attributes to update the values of these data items.

FlowMark Process—Actions

setHandle *handle*

Assigns the handle of an active FlowMark process control session to the part. Parameter *handle* has class `LargeInteger`.

A process control session handle has to be assigned to the part using this action before any of the process control functions can be invoked. You can connect `setHandle` to the `started` event of a FlowMark Session part to make sure that the handle is available as soon as the process control session is started.

start

Starts a new process instance in FlowMark Runtime.

The name of the template used to start the process from is taken from attribute `templateName`. Attribute `processName` is the default name for the new process, and attribute `startData` is the input data that is passed to the input container of the new process.

If the operation is successful, event `started` is signaled. Furthermore, since the name of the new process in FlowMark can be different from the default name passed to the API, event `processName` is signaled with the actual name of the new process as parameter. If the operation fails, event `processError` is signaled with a parameter indicating the type of the error that occurred.

startFromTemplate *template*

Sets the value of attribute `templateName` to the value of parameter *template*, and then starts a new process instance in FlowMark Runtime, as described for action `start`.

suspend

Suspends the process in FlowMark Runtime. The name of the process to suspend is taken from attribute `processName`.

If the operation is successful, event `suspended` is signaled. Furthermore, event `topLevelProcess` is signaled with the name of the top-level process in FlowMark that contains the process that has been suspended. If the operation fails, event `processError` is signaled with a parameter indicating the type of the error that occurred.

suspendAll

Like `suspend`, with the difference that, in addition to the process from `processName`, all of its subprocesses are suspended too.

resume

Resumes the process in FlowMark Runtime. The name of the process to resume is taken from attribute `processName`.

If the operation is successful, event `resumed` is signaled. Furthermore, event `topLevelProcess` is signaled with the name of the top-level process in FlowMark that contains the process that has been resumed. If the operation fails, event `processError` is signaled with a parameter indicating the type of the error that occurred.

resumeAll

Like `resume`, with the difference that, in addition to the process from `processName`, all of its subprocesses are resumed too.

terminate

Terminates the process in FlowMark Runtime. The name of the process to terminate is taken from attribute `processName`.

If the operation is successful, event `terminated` is signaled. Furthermore, event `topLevelProcess` is signaled with the name of the top-level process in FlowMark that contains the process that has been terminated. If the operation fails, event `processError` is signaled with a parameter indicating the type of the error that occurred.

restart

Restarts the process in FlowMark Runtime. The name of the process to restart is taken from attribute `processName`.

If the operation is successful, event `restarted` is signaled. Furthermore, event `topLevelProcess` is signaled with the name of the top-level process in FlowMark that contains the process that has been restarted. If the operation fails, event `processError` is signaled with a parameter indicating the type of the error that occurred.

queryTopLevelName

Obtains the name of the top-level process in FlowMark that contains the process whose name is stored in attribute `processName`.

If the operation is successful, event `topLevelProcess` is signaled with the name of the top-level process in FlowMark. If the operation fails, event `processError` is signaled with a parameter indicating the type of the error that occurred.

displayError *error*

Displays a message box describing an error that has occurred when attempting to perform a process control function. Parameter *error* has class `FmError` and contains an object describing the type of error that occurred.

You should connect this action to the `processError` event of this part.

addItemValue *name, value*

Creates an instance of class `FmStartDataItem` and adds it to the `startData` collection of the part. Parameter *name* has class `String` and contains the name of the data item. Parameter *value* has class `String`, `Integer`, or `Float` and contains the value that is to be assigned to the data item when the process is started.

addDataItem *dataItem*

Adds a data item to the `startData` collection of the part. Parameter *dataItem* has class `FmStartDataItem` and holds the data item that is to be added to the start data of the process.

If the `startData` already contains an item with the same name, the value of that item is updated with the new value from the *dataItem* parameter.

removeDataItem *dataItem*

Removes a data item from the startData collection of the part. Parameter *dataItem* has class FmStartDataItem and holds the data item that is to be removed from the start data of the process.

The value of *dataItem* is not significant. If the startData collection contains a data item with the same name, it is removed without checking the value.

FlowMark Process—Events**started**

A FlowMark process instance has been successfully created and started, using the values from the attributes of the part. The name of the new process is contained in attribute processName. This event is always signaled together with event processName.

suspended

The FlowMark process instance whose name is stored in attribute processName has been successfully suspended. This event is always signaled together with event topLevelProcess.

resumed

The FlowMark process instance whose name is stored in attribute processName has been successfully resumed. This event is always signaled together with event topLevelProcess.

terminated

The FlowMark process instance whose name is stored in attribute processName has been successfully terminated. This event is always signaled together with event topLevelProcess.

restarted

The FlowMark process instance whose name is stored in attribute processName has been successfully restarted. This event is always signaled together with event topLevelProcess.

topLevelProcess *tlp*

One of the process control functions, which, as a side effect, return the name of the top-level process, has been successfully executed. Parameter *tlp* has class String and contains the name of the top-level process in FlowMark that contains the process whose name is stored in the processName attribute.

processError *error*

An error has occurred when attempting to perform any FlowMark process control function. Parameter *error* has class FmError and contains an object describing the type of error that occurred.

You should connect this event to the displayError action of this part to display an error message to the user.

FlowMark Process—Settings

Instance name

The name of the FlowMark Runtime process instance that is to be manipulated by this part. If a new process instance is created, this is the default name that is to be used for the new instance.

The value you enter is stored in attribute `processName`.

Template name

If the part is to be used to create a new FlowMark process instance, this is the name of the process template that is to be used to create the instance from.

The value you enter is stored in attribute `templateName`.

Start Data

If the part is to be used to create a new FlowMark process instance, this is the input data that is passed to the new process. Click mouse button 2 somewhere on the table part to get a pop-up menu that you can use to modify the displayed list.

You can use the **Import** button to import a data structure from an FDL file.

The value you enter is stored in attribute `startData`.

FmApiLibrary

Class `FmApiLibrary` provides the necessary methods to call the external functions that make up the FlowMark API. An initialized instance of this class is created whenever application `FmApiBase` is loaded.

Subclass of:

`Object`

Instance variables:

*id query get set logon pass logoff start suspend resume terminate restart
changeAct*

Class variables:

DefaultLib

Pool dictionaries:

`FmConstants`

Public instance methods:

getContainerItem: *session* **name:** *name* **dataSize:** *size* **dataArea:** *data*
dataLength: *length*

Invoke FlowMark function `ExmcGetContainerItem` to get the data item *name*. Returns an integer containing the return code issued by FlowMark.

getSessionId: *session*

Invoke FlowMark function `ExmcGetSessionID` to obtain a session identifier. Returns an integer containing the return code issued by FlowMark.

logout: *handle*

Invoke FlowMark function ExmcLogout to terminate the process control session represented by *handle*. Returns an integer containing the return code issued by FlowMark.

login: *logonData handle: handle*

Invoke FlowMark function ExmcLogin to start a process control session with the logon values contained in *logonData*. Returns an integer containing the return code issued by FlowMark.

passThru: *session handle: handle*

Invoke FlowMark function ExmcPassThru to start an internal process control session for the session identifier *session*. Returns an integer containing the return code issued by FlowMark.

queryContainerStructure: *session type: type dataSize: size data: data
dataLength: length*

Invoke FlowMark function ExmcQueryDataStructure to obtain the structure of the FlowMark data container. Returns an integer containing the return code issued by FlowMark.

restart: *handle name: name tlp: tName mode: mode*

Invoke FlowMark function ExmcRestartProcess to restart the process instance identified by *name*. Returns an integer containing the return code issued by FlowMark.

resume: *handle name: name tlp: tName mode: mode*

Invoke FlowMark function ExmcResumeProcess to resume the process instance identified by *name*. Returns an integer containing the return code issued by FlowMark.

setContainerItem: *session name: name dataSize: size dataArea: data*

Invoke FlowMark function ExmcSetContainerItem to set the data item *name* to the value in *data*. Returns an integer containing the return code issued by FlowMark.

start: *handle from: template name: name with: data*

Invoke FlowMark function ExmcStartProcess to create and start a new process instance from template *template*, giving it name *name* and input data *data*. Returns an integer containing the return code issued by FlowMark.

suspend: *handle name: name tlp: tName mode: mode*

Invoke FlowMark function ExmcSuspendProcess to suspend the process instance identified by *name*. Returns an integer containing the return code issued by FlowMark.

terminate: *handle name: name tlp: tName mode: mode*

Invoke FlowMark function ExmcTerminateProcess to terminate the process instance identified by *name*. Returns an integer containing the return code issued by FlowMark.

Public class methods:

default

Returns the initialized instance of the receiver that has been created when the application was loaded.

FmError

Class FmError provides the means to implement a generalized error-handling mechanism for all interactions with FlowMark. All methods in any of the classes of this integration package that invoke a FlowMark API function returns an instance of this class when FlowMark returns an error code.

Subclass of:

Object

Instance variables:

code text

Class variables:

none

Pool dictionaries:

FmConstants

Public instance methods:

display

Display a message box describing the error condition stored in the receiver.

errorCode

Returns an integer containing the error code that has been issued by FlowMark.

errorCodeAsString

Returns a string that is the textual representation (the name of the variable in the *FmConstants* dictionary) of the error code that has been issued by FlowMark.

errorCondition

Returns a string that contains the textual representation of the error code, followed by the numeric error code in parentheses.

isDatabaseError

Returns true if the receiver holds an error code indicating an invalid database field in the logon data structure of an ExmcLogon API call.

isFmError

Returns true. A similar method has been implemented in class Object returning false. This enables sending message `isFmError` to any object to check whether it is an instance of this class.

isPasswordError

Returns true if the receiver holds an error code indicating an invalid password field in the logon data structure of an ExmcLogon API call.

isServerError

Returns true if the receiver holds an error code indicating an invalid server field in the logon data structure of an ExmcLogon API call.

isUserIdError

Returns true if the receiver holds an error code indicating an invalid user ID field in the logon data structure of an ExmcLogon API call.

Public class methods:**forCode:** *anInteger*

Returns an initialized instance of the receiver that describes the FlowMark error condition indicated by code *anInteger*.

FmContainer

Class FmContainer provides the common interface for its subclasses FmInputContainer and FmOutputContainer.

It serves as an abstract base class, so you should not create instances of this class. Instead, use the provided subclasses.

Subclass of:

Object

Instance variables:

attrCount attrInfos itemValues sessionId

Class variables:

none

Pool dictionaries:

FmConstants

Public instance methods:**cardinalityOf:** *item*

Returns an integer containing the cardinality of the data item or data structure member with name *item*. The value is 0 if the data structure member for *item* does not describe an array of data. Otherwise, it is the number of elements in the array.

dataMembers

Returns an ordered collection containing the names of all data structure members in the container.

defaultValueFor: *item*

Returns the default value for the data item *item*, which is an integer 0 if *item* has type *Long*, a null string for type *String*, and a floating point 0 for type *Float*. If *item* is not included in the container, the return value is an integer 0.

includesAttribute: *aString*

Returns true if *aString* is the name of a data structure member in the container.

includesDataItem: *aString*

Returns true if *aString* is the name of a data item in the container.

items

Returns an ordered collection containing the names of all data items in the container.

queryStructure

Query the structure of the FlowMark data container. Returns an instance of class *FmError* if FlowMark returns an error condition.

setSessionId: *anId*

Set the session identifier that is to be used for all FlowMark API calls issued by the receiver to *anId*, which must be a string holding the FlowMark session identifier assigned to your application by FlowMark.

stringTypeOf: *item*

Returns a string, the type of the data item or data structure member with name *item*. Possible values are *String*, *Long*, and *Float*, or *unknown* if *item* does not exist in the container.

Public class methods:

none

FmInputContainer

Class *FmInputContainer* can be used to access the data in the input container of a FlowMark activity.

Subclass of:

FmContainer

Instance variables:

none

Class variables:

none

Pool dictionaries:

none

Public instance methods:

getAllItems

Get the value for all data items in the container from FlowMark and store them in the internal data collection. Returns an instance of FmError if FlowMark returns any error code other than ExmpjItemNotSet.

getItem: *item*

Returns the value of data item *item* in the container. The value is obtained from the internal data collection, or, if it has not yet been retrieved, the FlowMark API is invoked to get the value from the container. If the FlowMark API returns an error code, an instance of FmError is returned.

Public class methods:

none

FmOutputContainer

Class FmOutputContainer can be used to update the data in the output container of a FlowMark activity.

Subclass of:

FmContainer

Instance variables:

none

Class variables:

none

Pool dictionaries:

none

Public instance methods:

setItem: *item* **value:** *anObject*

Set the value of data item *item* in the output container to *anObject* via the FlowMark API. The type of *anObject* must be String, Long, or Float and must match the type of the data item in the FlowMark container. If the FlowMark API returns an error code, an instance of FmError is returned.

getItem: *item*

Returns the value of data item *item* that has been previously set to the container, or, if the data item has not been set, returns the default value for the specified data item. The value is obtained from the internal data collection of the part. Thus, a previously set value is returned only if it has been set using the same part. An instance of FmError is returned if the data item does not exist in the container.

Public class methods:

none

FmContainerItem

Class FmContainerItem is used to describe the data structure members in a FlowMark container.

Subclass of:

Object

Instance variables:

name type cardinality stringType requested

Class variables:

none

Pool dictionaries:

FmConstants

Public instance methods:

= anObject

Returns true if *anObject* has the same name as the receiver.

cardinality

Returns an integer containing the cardinality of the data structure member.

name

Returns the name of the data structure member.

type

Returns an integer containing the type of the data structure member, which is one of the constants ExmpjTypeString, ExmpjTypeLong, or ExmpjTypeFloat from the FmConstants pool dictionary.

stringType

Returns a string describing the type of the data structure member.
Possible values are String, Long, Float, and unknown.

valueClass

Returns the class that is required for values of the data structure member.
Possible values are String, Integer, Float, and Object.

Public class methods:

namePtr: aPtr type: aType size: anInt

Returns an initialized instance of the receiver, with name set to a string obtained from address *aPtr*, type *aType*, and size *anInt*.

FmStartDataItem

Class FmStartDataItem is used to hold input data items that can be passed to new processes.

Subclass of:

Object

Instance variables:

name value displayValue

Class variables:

none

Pool dictionaries:

none

Public instance methods:

= anObject

Returns true if *anObject* has the same name as the receiver.

dataSize

Returns the size of the value. The returned value is 4 for integers, 8 for floating point numbers, and the actual size of the string for strings.

displayValue

Returns a string holding the value of the receiver.

name

Returns the name of the receiver.

name: *aString*

Set the name of the receiver to *aString*.

namePtr

Returns a pointer to the name of the receiver.

type

Returns a string representing the type of the receiver. Possible values are Long, Float, and String.

value

Returns the current value of the receiver.

value: *aValue*

Set the value of the receiver to *aValue*, which must have type String, Integer, or Float.

valuePtr

Returns a pointer to the value of the receiver. If the value is of type Float, the pointer does not point to the actual value, but to the value encoded as C-language double precision number.

Public class methods:

name: *aName* **value:** *aValue*

Returns an initialized instance of the receiver, with name set to *aName* and value set to *aValue*. A value must have type String, Integer, or Float.

FmExmApiBegin

Class FmExmApiBegin is used as a data structure that holds the required data to perform a logon to FlowMark.

Subclass of:

OSStruct

Instance variables:

none

Class variables:

none

Pool dictionaries:

none

Public instance methods:

database

Returns the value of field *database*, which is a pointer.

database: *obj*

Set the value of field *database* to the pointer *obj*.

password

Returns the value of field *password*, which is a pointer.

password: *obj*

Set the value of field *password* to the pointer *obj*.

server

Returns the value of field *server*, which is a pointer.

server: *obj*

Set the value of field *server* to the pointer *obj*.

userid

Returns the value of field *userid*, which is a pointer.

userid: *obj*

Set the value of field *userid* to the pointer *obj*.

Public class methods:

fixedSize

Returns the size of the receiver in bytes.

FmExmApiTypeInfo

Class FmExmApiTypeInfo is used as a data structure to access the information that is returned by FlowMark when the structure of a data container is queried.

Subclass of:

OSStruct

Instance variables:

none

Class variables:

none

Pool dictionaries:

none

Public instance methods:

none

createAttribute: *anInt*

Returns a new instance of class FmContainerItem initialized from the variable item at position *anInt*.

nameAt: *anIndex*

Returns the value of field *name*, which is a pointer, of the variable item at position *anIndex*.

nameAt: *anIndex* **put:** *obj*

Set the value of field *name* of the variable item at position *anIndex* to the pointer *obj*.

number

Returns the value of field *number*, which is an integer.

number: *obj*

Set the value of field *number* to the integer *obj*.

sizeAt: *anIndex*

Returns the value of field *size*, which is an integer, of the variable item at position *anIndex*.

sizeAt: *anIndex* **put:** *obj*

Set the value of field *size* of the variable item at position *anIndex* to the integer *obj*.

typeAt: *anIndex*

Returns the value of field *type*, which is an integer, of the variable item at position *anIndex*.

typeAt: *anIndex* **put:** *obj*

Set the value of field *type* of the variable item at position *anIndex* to the integer *obj*.

Public class methods:

fixedSize

Returns the size of the fixed part of the receiver in bytes.

totalLength: *length*

Returns a new instance of the receiver, with an overall size of *length* bytes, including both fixed and variable components.

variableSize

Returns the size of the variable part of the receiver in bytes.

FmExmApiStructureData

Class FmExmApiStructureData is used as a data structure that holds the input data that is to be passed to new FlowMark processes.

Subclass of:

OSStruct

Instance variables:

none

Class variables:

none

Pool dictionaries:

none

Public instance methods:

dataareaAt: *anIndex*

Returns the value of field *dataarea*, which is a pointer, of the variable item at position *anIndex*.

dataareaAt: *anIndex* **put:** *obj*

Set the value of field *dataarea* of the variable item at position *anIndex* to the pointer *obj*.

nameAt: *anIndex*

Returns the value of field *name*, which is a pointer, of the variable item at position *anIndex*.

nameAt: *anIndex* **put:** *obj*

Set the value of field *name* of the variable item at position *anIndex* to the pointer *obj*.

number

Returns the value of field *number*, which is an integer.

number: *obj*

Set the value of field *number* to the integer *obj*.

sizeAt: *anIndex*

Returns the value of field *size*, which is an integer, of the variable item at position *anIndex*.

sizeAt: *anIndex* **put:** *obj*

Set the value of field *size* of the variable item at position *anIndex* to the integer *obj*.

Public class methods:

fixedSize

Returns the size of the fixed part of the receiver in bytes.

totalLength: *length*

Returns a new instance of the receiver, with an overall size of *length* bytes, including both fixed and variable components.

variableSize

Returns the size of the variable part of the receiver in bytes.

FmMaintainProcess

Class FmMaintainProcess displays a window (shown in Figure 15) that enables you to start new FlowMark processes from the provided sample process template and to perform any of the other process-control functions defined in the FlowMark process API on any of the process instances available in FlowMark Runtime.

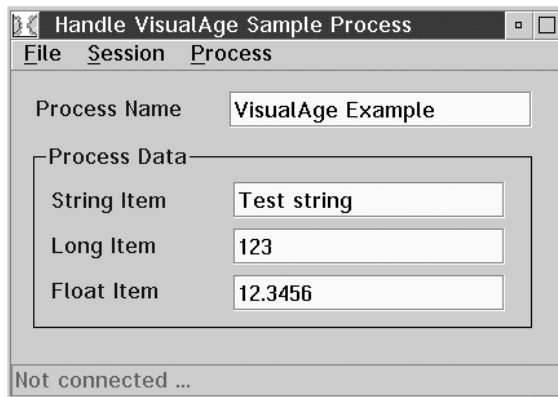


Figure 15. Window Handle VisualAge Sample Process

Use the **Session** menu to start and stop a FlowMark process control session.

Use the **Process** menu to perform process control functions on the process whose name is displayed in the **Process Name** field.

When you select **Start** from **Process**, a new instance is created from process template VisualAge Sample, with the instance name and the input data from the entry fields.

The status area at the bottom of the panel displays the result of the last operation that you performed.

FmDisplayActivity

Part FmDisplayActivity displays a window showing the process name and the activity name of the FlowMark activity it has been invoked from, as well as its FlowMark session identifier and the passed input parameter string.

The window looks like the window shown in Figure 16.

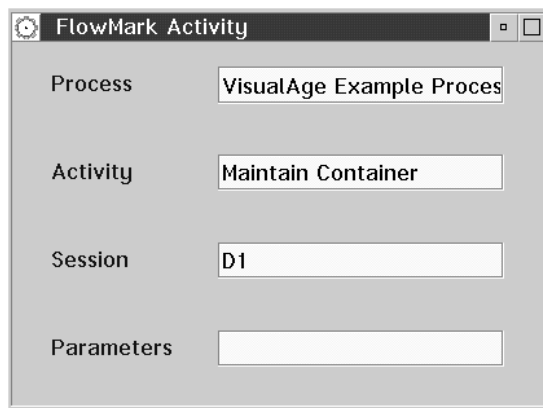


Figure 16. Window FlowMark Activity

FmMaintainContainer

Class FmMaintainContainer displays a window (shown in Figure 17) that enables you to browse the contents of the input container and to modify the contents of the output container of the FlowMark activity that it is invoked from.

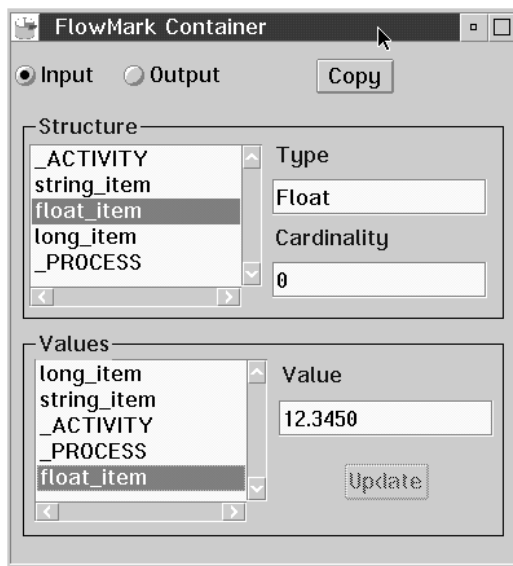


Figure 17. Window FlowMark Container

The window consists of:

- A set of radio buttons labeled **Input** and **Output**, which you can use to switch between input and output mode, and a **Copy** push button used to copy all values of the input container to the output container.
- A group box labeled **Structure**, which displays the structure of the data in the container. The list box shows the names of all data structure members in the container. When you select an element in the list box, the entry fields show the type and cardinality of the data structure member.
- A group box labeled **Values**, which can be used to display the contents of the input container or to update the contents of the output container. The list box shows all data items in the container. In input mode, when you select an element from the list, the value of the data item is displayed in the text field. In output mode, you can select an element, type a value in the text field, and click on the **Update** button to set the value of the data item in the container.

FmMaintainData

Part FmMaintainData displays a window showing the contents of data structure VisualAge Sample. When the window opens, the entry fields display the data from the FlowMark input container. When you close the window with the **OK** button, the contents of the entry fields are written to the FlowMark output container.

The window looks like the one shown in Figure 18.

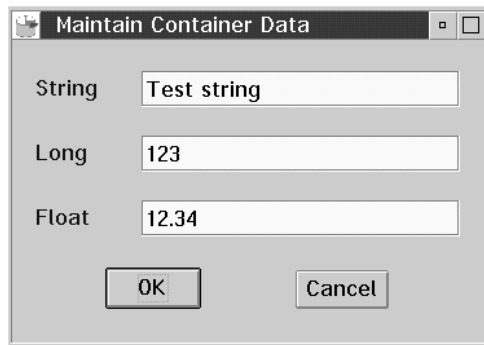


Figure 18. Window Maintain Container Data

Chapter 6. Creating your own service brokers

This section provides information you need to develop your own service brokers and services.

Designing service brokers and services

If you did not install the Toolkit and Samples when the Service Broker Manager was installed, refer to the *Installation and Maintenance* guide for details on how to install them at any other time.

Ensure that the directory where the service broker and service DLLs reside is specified in the LIBPATH statement in your CONFIG.SYS file. This is done automatically by the standard Service Broker Manager installation.

Service brokers and services must be written in C or C++ and compiled with Version 3.0 of the IBM VisualAge for C++ for OS/2 or any other equivalent compiler that can compile 32-bit OS/2 programs.

API functions described in this book are provided as a programming interface for C, C++, or REXX programs. Although other functions may appear in header files, use only the functions described in this book.

C language conventions

Service brokers and service functions receive a single parameter that points to a structure containing actual input or output parameters. The actual size of such a parameter structure is stored in the first field:

```
typedef struct {
    ULONG    Size;          /* I: actual size of structure */
    ...          /* additional fields */
} AnyStructure;
```

The size received should then be compared with the size information for the respective structure within the service broker or the service function. If the sizes do not match, it indicates different versions of Service Broker Manager and service broker, or service broker and service function.

C structure descriptions contain the following abbreviations:

Abbreviation	Description
I	Input (read-only).
O	Output (written by the function).
I/O	Input and output.

Implementing a service broker

This section describes the operation of service brokers and how to implement them. The service broker functions described in this section are called for each service broker instance.

Implementation

Each service broker consists of a DLL that exports the following functions (optional functions are marked with `opt`):

- `Broker_GetDllVersion`
- `Broker_GetVersion`
- `Broker_GetCfgReqs opt`
- `Broker_Init opt`
- `Broker_Exit opt`
- `Broker_Logon opt`
- `Broker_Logoff opt`
- `Broker_SetupCfg opt`

The Service Broker Manager calls these functions to:

- Determine the version of the service broker
- Determine specific requirements of the service broker
- Load and initialize the service broker
- Perform logon and logoff according to the managed application
- Clean up and unload the service broker
- Set up specific configuration notebook pages

Service broker functions are exported through the use of a module definition file. The following is a template for a service broker module definition file:

```
LIBRARY SAMPBROK INITINSTANCE TERMINSTANCE
DESCRIPTION 'Sample Broker DLL'
PROTMODE
DATA MULTIPLE NONSHARED
EXPORTS

    Broker_GetDllVersion
    Broker_GetVersion
    Broker_Init
    Broker_Exit
    Broker_Logon
    Broker_Logoff
```

Necessary definitions are included in the header file EXMP3CBR.H, which requires the standard OS/2 header file OS2.H to be included first. A sample service broker DLL, SAMPBROK.C, is shown in “Sample service broker DLL (SAMPBROK.C)” on page 146.

Description of functions

The following sections describe the functions exported by service broker DLLs.

Broker_GetDllVersion

This mandatory function has the following format:

```
ULONG APIENTRY Broker_GetDllVersion (VOID)
```

This function must return the value of the predefined constant SB_BROKER_DLLVERSION.

Example

```
ULONG APIENTRY Broker_GetDllVersion (VOID)
{
    return SB_BROKER_DLLVERSION;
}
```

Broker_GetVersion

This mandatory function has the following format:

```
VOID APIENTRY Broker_GetVersion (SbBrokerVersion * pVersion)
```

This function must return the internal name and version of the service broker DLL by filling in an empty `SbBrokerVersion` structure that is passed to the function.

The `SbBrokerVersion` structure is defined as follows:

```
typedef struct {  
    ULONG    Size;          /* I: actual size of structure */  
    PSZ     pName;         /* 0: user-defined broker name */  
    ULONG    Version;      /* 0: user-defined broker version */  
} SbBrokerVersion;
```

Important

It is recommended to provide a C header file containing name and version declarations. Implementers of services can then use this information to check the version of the service broker.

Example

```
VOID APIENTRY Broker_GetVersion (SbBrokerVersion * pVersion)  
{  
    pVersion->pName = "SampleBroker";  
    pVersion->Version = 123;  
}
```

Broker_GetCfgReqs

This optional function has the following format:

```
VOID APIENTRY Broker_GetCfgReqs (SbBrokerCfgReqs * pReqs)
```

This function can return configuration requirements that need to be enforced by the Service Broker Manager.

The `SbBrokerCfgReqs` structure is defined as follows:

```
typedef struct {  
    ULONG    Size;          /* I: actual size of structure */  
    BOOL32   Synchronized; /* 0: synchronization required? */  
    ULONG    MinStackSize; /* 0: minimum stack size, 0 or >8192 */  
} SbBrokerCfgReqs;
```


If the service broker must run within a single synchronized thread, `Synchronized` can be set to `TRUE`. Furthermore, if the minimum stack size of threads assigned to the service broker exceeds 8192 bytes (8K), it can be set in the `MinStackSize` output variable.

Example

```
VOID APIENTRY Broker_GetCfgReqs (SbBrokerGetCfgReqs * pReqs)
{
    pReqs->Synchronized = FALSE;
    pReqs->MinStackSize = 32768;    /* 32K */
}
```

Broker_Init

This optional function has the following format:

```
APIRET APIENTRY Broker_Init (SbBrokerInit * pInit)
```

This function is called when a service broker instance is loaded and returns 0 if it was successful, otherwise the loading of the service broker is discontinued. A structure containing initialization information is passed to the function:

```
typedef struct {
    ULONG    Size;           /* I: actual size of structure */
    ULONG    Handle;        /* I: handle of broker instance */
    HMODULE  BrokerDLL;     /* I: module handle of broker DLL */
    PSZ      pBrokerName;   /* I: logical name of broker instance */
    PVOID    pInstance;     /* O: user-defined instance pointer */
} SbBrokerInit;
```

You can use a user-defined instance pointer, `pInstance`, to store any instance-related information. This pointer is passed to all subsequent function calls, but can only be set in the `Broker_Init` function.

If you need to know the module handle of the service broker DLL (for example, to load ± resources), store the value of the `BrokerDLL` field in order to use it during subsequent function calls (for example, `Broker_Logon`).

The handle of the service broker instance, `Handle`, is necessary to issue service broker API functions (see “Using the C language service broker API” on page 110).

If this function takes more than 15 seconds (for example, due to user interaction), the `SbbDisableTimeout` service broker API function must be called (see “Disabling timeout” on page 115).

Example

```
APIRET APIENTRY Broker_Init (SbBrokerInit * pInit)
{
    /* ... user-defined initialization ... */
    return 0;
}
```

Broker_Exit

This optional function has the following format:

```
APIRET APIENTRY Broker_Exit (SbBrokerExit * pExit)
```

This function is called when a service broker instance is unloaded and returns 0 if it was successful, otherwise the unloading of the service broker is discontinued. A read-only structure containing exit information is passed to the function:

```
typedef struct {
    ULONG    Size;           /* I: actual size of structure */
    ULONG    Handle;        /* I: handle of broker instance */
    PVOID    pInstance;     /* I: user-defined instance pointer */
} SbBrokerExit;
```

If this function takes more than 15 seconds (for example, due to user interaction), the SbbDisableTimeout service broker API function must be called (see “Disabling timeout” on page 115).

Example

```
APIRET APIENTRY Broker_Exit (SbBrokerExit * pExit)
{
    /* ... user-defined exit ... */

    return 0;
}
```

Broker_Logon

This optional function has the following format:

```
APIRET APIENTRY Broker_Logon (SbBrokerLogon * pLogon)
```

This function performs a logon to the application that is managed by the service broker, and can fill in optional fields in a `SbBrokerLogon` structure that is passed to the function. It returns 0 if it was successful, otherwise the logon is discontinued.

The `SbBrokerLogon` structure is defined as follows:

```
typedef struct {
    ULONG    Size;           /* I: actual size of structure */
    ULONG    Handle;        /* I: handle of broker instance */
    HAB      Hab;           /* I: anchor block */
    HMQ      Hmq;           /* I: message queue */
    PVOID    pInstance;     /* I: user-defined instance pointer */
    PVOID    pSession;      /* O: user-defined session data */
    ULONG    SessionSize;   /* O: size of session data */
    PVOID    pInfoArea;     /* I: Logon information area */
    ULONG    InfoAreaLength; /* I: Length of area */
} SbBrokerLogon;
```

`pSession` and `SessionSize` refer to a user-defined logon or session buffer, that contains a handle of the connection to the managed application, or similar logon information.

If the buffer is allocated by this function, it must be freed in the `Broker_Logoff` function. `pSession` and `SessionSize` are also accessible from registered services of this service broker instance. `pInfoArea` and `InfoAreaLength` are passed from the requester to the Service Broker Manager if the service broker is started with the API call `SbrStartBrokerWithInfo`. The information in these fields can be used during logon, for example, to suppress the display of a logon panel. If the broker was not started via that API function, `pInfoArea` and `InfoAreaLength` are set to 0.

To accomplish any Presentation Manager (PM) tasks during logon (for example, displaying a logon panel), current handles of anchor block and message queue are provided in the `Hab` and `Hmq` fields. However, the PM message queue is not processed. Therefore, any PM operations that require a standard PM message loop are not allowed.

If this function takes more than 15 seconds (for example, due to user interaction), the `SbbDisableTimeout` service broker API function must be called (see "Disabling timeout" on page 115).

Important

It is recommended to provide a C header file containing type declarations needed for the session buffer. Implementers of services can then use this information to write services running under this particular service broker.

Example

```
typedef struct { ... } MyLogonInfo;      /* Logon structure */

...

APIRET APIENTRY Broker_Logon (SbBrokerLogon * pLogon)
{
    MyLogonInfo * pLogonInfo;

    pLogonInfo = (MyLogonInfo *) malloc (sizeof (MyLogonInfo));

    /* ... logging on ... */
    /* ... set logon information ... */

    pLogon->pSession = (PVOID) pLogonInfo;
    pLogon->SessionSize = sizeof (MyLogonInfo);

    return 0;
}
```

Broker_Logoff

This optional function has the following format:

```
APIRET APIENTRY Broker_Logoff (SbBrokerLogoff * pLogoff)
```

This function performs a logoff from the application that is managed by the service broker. The fields of the SbBrokerLogoff structure contain the information from the SbBrokerLogon structure from the call to Broker_Logon. A session buffer that was allocated by Broker_Logon must be freed with this function.

The SbBrokerLogoff structure is defined as follows:

```
typedef struct {

    ULONG    Size;           /* I: actual size of structure */
    ULONG    Handle;        /* I: handle of broker instance */
    HAB      Hab;           /* I: anchor block */
    HMQ      Hmq;           /* I: message queue */
    PVOID    pInstance;     /* I: user-defined instance pointer */
    PVOID    pSession;      /* I: user-defined session data */
    ULONG    SessionSize;   /* I: size of session data */

} SbBrokerLogoff;
```

If this function takes more than 15 seconds (for example, due to user interaction), the `SbbDisableTimeout` service broker API function must be called (see “Disabling timeout” on page 115).

Example

```
APIRET APIENTRY Broker_Logoff (SbBrokerLogoff * pLogoff)
{
    /* ... logging off ... */
    free (pLogoff->pSession);
    return 0;
}
```

Broker_SetupCfg

This optional function has the following format:

```
APIRET APIENTRY Broker_SetupCfg (SbBrokerCfg * pCfg)
```

This function is called when the Settings notebook of an already loaded service broker instance is opened. You can add user-defined pages to the notebook. The function returns 0 if it was successful. A read-only structure containing the notebook window handle is passed to the function:

```
typedef struct {
    ULONG    Size;           /* I: actual size of structure */
    ULONG    Handle;        /* I: handle of broker instance */
    PVOID    pInstance;     /* I: user-defined instance pointer */
    HWND     Notebook;     /* I: handle of configuration notebook */
} SbBrokerCfg;
```

See the *OS/2 Warp, Version 3 Presentation Manager Programming Guide, Advanced Topics*, for details on notebook programming.

There are three predefined window messages that added notebook pages may handle:

Message	Description
WM_SB_BROK_SET_FOCUS	The notebook page receives the focus.
WM_SB_BROK_COLLECT_DATA	The user pressed the notebook's OK button. The notebook page may save any user-defined configuration values, for example, by using the profile functions of the service broker API (see "Using the C language service broker API"). It must respond by returning either TRUE to signal successful completion, or FALSE in case of errors (for example, if wrong user input was detected). In the latter case, the notebook page then receives the WM_SB_BROK_SET_FOCUS_ERROR window message.
WM_SB_BROK_SET_FOCUS_ERROR	The notebook page receives the focus due to an error in the handling of the WM_SB_BROK_COLLECT_DATA window message.

Example

```
APIRET APIENTRY Broker_SetupCfg (SbBrokerCfg * pCfg)
{
    /* ... add user-defined notebook pages ... */

    return 0;
}
```

Using the C language service broker API

The service broker API provides auxiliary functions that can be utilized by service brokers and services. Service broker API function names start with the prefix Sbb.

The auxiliary API functions are:

- SbbWriteProfile
- SbbReadProfile
- SbbLog
- SbbQueryLogLevel
- SbbDisableTimeout

Service broker API functions may only be called:

- After or within the Broker_Init or Service_Init function
- Before or within the Broker_Exit or Service_Exit function
- Within service functions

A handle of a service broker or service instance is needed by all service broker API functions and is passed to all standard service broker or service functions like `Broker_Init` and `Service_Init`.

Necessary definitions are in the header file `EXMP3CBR.H` that is also included by `EXMP3CSE.H`. The standard OS/2 header file `OS2.H` must be included first. Service brokers and services using the service broker API must be linked with the supplied library `EXMP3KBR.LIB`.

Storing configuration data

Use the `SbbWriteProfile` function to store user-defined configuration data for a service broker or service instance.

```
APIRET APIENTRY SbbWriteProfile (ULONG Handle,
                                PVOID pBuffer,
                                ULONG BufferSize)
```

Parameter	I/O	Description
Handle	I	Handle of service broker or service instance.
pBuffer	I	User-defined configuration data or NULL.
BufferSize	I	Size of user-defined configuration data or 0.

Return Code	Description
SB_BROK_RC_OK (0)	Operation completed successfully.
SB_BROK_RC_MEMORY (1)	Not enough memory.
SB_BROK_RC_INVALID_ARGS (2)	Invalid data or size specified.
SB_BROK_RC_INVALID_HANDLE (3)	Invalid handle specified.

If `pBuffer` is NULL and `BufferSize` is 0, an existing configuration entry is deleted.

Example

```
typedef struct { ... } MyConfigData;    /* configuration data */

...

VOID APIENTRY Broker_Exit (SbBrokerExit * pExit)
{
    MyConfigData config = { ... };      /* set configuration data */
    APIRET      rc;

    rc = SbbWriteProfile (pExit->Handle, &config, sizeof(config));

    if (rc != SB_BROK_RC_OK)
    {
        /* ... error handling ... */
    }
}
```

Retrieving configuration data

Use the SbbReadProfile function to retrieve user-defined configuration data for a service broker or service instance.

```
APIRET APIENTRY SbbReadProfile (ULONG  Handle,
                                PVOID  pBuffer,
                                PULONG  pBufferSize)
```

Parameter	I/O	Description
Handle	I	Handle of service broker or service instance.
pBuffer	O	Address of a buffer where configuration data is written or NULL.
pBufferSize	I/O	On input it contains the size of the supplied buffer or 0. On return the actual size of the stored configuration data is returned. If the actual size exceeds the specified size, error SB_BROK_RC_BUFFER_OVERFLOW is returned.

Return Code	Description
SB_BROK_RC_OK (0)	Operation completed successfully.
SB_BROK_RC_INVALID_ARGS (2)	Invalid data or size specified.
SB_BROK_RC_INVALID_HANDLE (3)	Invalid handle specified.
SB_BROK_RC_BUFFER_OVERFLOW (4)	Supplied buffer is too small.

Example

```
typedef struct { ... } MyConfigData;    /* configuration data */

...

APIRET APIENTRY Service_Init (SbServiceInit * pInit)
{
    MyConfigData config;                /* buffer */
    ULONG        size = sizeof(config); /* size of buffer */
    APIRET      rc;

    rc = SbbReadProfile (pInit->Handle, &config, &size);

    if (rc != SB_BROK_RC_OK || size != sizeof(config))
    {
        /* ... error handling ... */
    }

    /* ... use configuration data ... */
}
```

Logging messages

Use the SbbLog function to log any user-defined messages. These messages are displayed in the Message Monitor of the Service Broker Manager. Furthermore, date and time information and the name of the service broker or service instance are inserted.

```
APIRET APIENTRY SbbLog (ULONG Handle,
                       ULONG Level,
                       PSZ  pText)
```

Parameter	I/O	Description
Handle	I	Handle of service broker or service instance.
Level	I	Logging level of message specified in pText: <ul style="list-style-type: none"> • SB_BROK_LOG_LEVEL_1 for error messages • SB_BROK_LOG_LEVEL_2 for error messages and general status information • SB_BROK_LOG_LEVEL_3 for error messages, status information, and problem determination messages <p>The message is logged only if the current logging level of the Service Broker Manager is greater than or equal to the specified level. The Service Broker Manager logging levels are described in "Defining the detail level for messages" on page 14.</p>
pText	I	User-defined message to be logged without CR or LF characters.

Return Code	Description
SB_BROK_RC_OK (0)	Operation completed successfully.
SB_BROK_RC_INVALID_ARGS (2)	Invalid data or size specified.
SB_BROK_RC_INVALID_HANDLE (3)	Invalid handle specified.

Example

```
APIRET APIENTRY Broker_Logon (SbBrokerLogon * pLogon)
{
    SbbLog (pLogon->Handle, SB_BROK_LOG_LEVEL_2, "Logon in progress ...");

    /* ... user-defined logon ... */

    SbbLog (pLogon->Handle, SB_BROK_LOG_LEVEL_2, "Logged on!");

    return 0;
}
```

Querying current logging level

Use the SbbQueryLogLevel function to determine the current logging level of the Service Broker Manager. In situations where complex logging messages are built, this function can be used to check in advance if logging is appropriate at all.

```
APIRET APIENTRY SbbQueryLogLevel (ULONG Handle,
                                  PULONG pLevel)
```

Parameter	I/O	Description
Handle	I	Handle of service broker or service instance.
pLevel	O	Address of a value where current logging level is returned.

Return Code	Description
SB_BROK_RC_OK (0)	Operation completed successfully.
SB_BROK_RC_INVALID_ARGS (2)	Invalid data or size specified.
SB_BROK_RC_INVALID_HANDLE (3)	Invalid handle specified.

Example

```
APIRET APIENTRY Service_Start (SbBrokerStart * pStart)
{
    ULONG level;

    SbbQueryLogLevel (pStart->Handle, &level);

    if (level >= SB_BROK_LOG_LEVEL_2) {
        /* ... build and log messages ... */
    }

    /* ... user-defined start-up ... */

    return 0;
}
```

Disabling timeout

The SbbDisableTimeout function *must* be used in situations where the following functions take more than 15 seconds (for example, due to user interaction):

- Broker_Init
- Broker_Exit
- Broker_Logon
- Broker_Logoff
- Service_Init
- Service_Exit
- Service_Start
- Service_Stop

The Service Broker Manager disables service brokers and services that do not return within 15 seconds if you did not disable this timeout with the SbbDisableTimeout function.

```
APIRET APIENTRY SbbDisableTimeout (ULONG Handle)
```

Parameter	I/O	Description
Handle	I	Handle of service broker or service instance.

Return Code	Description
SB_BROK_RC_OK (0)	Operation completed successfully.
SB_BROK_RC_MEMORY (1)	Disabling was not possible.
SB_BROK_RC_INVALID_HANDLE (3)	Invalid handle specified.

Example

```
APIRET APIENTRY Broker_Logon (SbBrokerLogon * pLogon)
{
    SbbDisableTimeout (pLogon->Handle);

    /* ... display logon panel ... */

    return 0;
}
```

Implementing a service

This section describes the operation of services and how to implement them. The standard service functions described in this section are called for each service instance.

Implementation

Each service consists of a DLL that exports the following standard functions, plus any user-defined service functions (optional functions are marked with `opt`):

- `Service_GetDllVersion`
- `Service_CheckBroker`
- `Service_GetCfgReqs opt`
- `Service_Init opt`
- `Service_Exit opt`
- `Service_Start opt`
- `Service_Stop opt`
- `Service_SetupCfg opt`

The Service Broker Manager calls these functions to:

- Determine the version of the service
- Check if the service is compatible with a service broker
- Determine specific requirements of the service
- Load and initialize the service
- Start and stop the service
- Clean up and unload the service
- Set up specific configuration notebook pages

A service can additionally export an arbitrary number of service functions performing user-defined operations using the current logon or session provided by the corresponding service broker.

For performance reasons, you can place a single service in a service broker DLL. But this is recommended only for general services that are needed whenever the service broker is used.

Multiple service functions can be executed in parallel, so they should be reentrant. To avoid unserialized access to the C run-time environment, use multithread libraries supplied with your compiler (for example, specify the /Gm+ compiler option to enable multithread support for the IBM VisualAge for C++).

Service functions are exported through the use of a module definition file. A template for a service DLL module definition file is shown next:

```
LIBRARY SAMPSEV INITINSTANCE TERMINSTANCE
DESCRIPTION 'Sample Service DLL'
PROTMODE
DATA MULTIPLE NONSHARED
EXPORTS

    Service_GetDllVersion
    Service_CheckBroker
    Service_Init
    Service_Exit
    Service_Start
    Service_Stop

    AFunction
    AnotherFunction
```

Necessary definitions are included in the header file EXMP3CSE.H which requires the standard OS/2 header file OS2.H to be included first. A sample service DLL, SAMPSRVC.C, is shown in "Sample service DLL (SAMPSEV.C)" on page 148.

Description of functions

The following sections describe the functions exported by service DLLs.

Service_GetDllVersion

This mandatory function has the following format:

```
ULONG APIENTRY Service_GetDllVersion (VOID)
```

This function must return the value of the predefined constant `SB_SERVICE_DLLVERSION`.

Example

```
ULONG APIENTRY Service_GetDllVersion (VOID)
{
    return SB_SERVICE_DLLVERSION;
}
```

Service_CheckBroker

This mandatory function has the following format:

```
APIRET APIENTRY Service_CheckBroker (SbBrokerVersion * pVersion)
```

This function checks if the service is compatible with the service broker. A read-only `SbBrokerVersion` structure is passed to the function containing the internal name and version of the service broker. The function returns 0 if it is compatible with the service broker, otherwise the loading of the service is discontinued (see “`Broker_GetVersion`” on page 103).

The `SbBrokerVersion` structure is defined as follows:

```
typedef struct {
    ULONG    Size;           /* I: actual size of structure */
    PSZ     pName;          /* I: user-defined broker name */
    ULONG    Version;       /* I: user-defined broker version */
} SbBrokerVersion;
```

Example

```
APIRET APIENTRY Service_CheckBroker (SbBrokerVersion * pVersion)
{
    return (strcmp (pVersion->pName, "SampleBroker") != 0) ||
           pVersion->Version < 123);
}
```

Service_GetCfgReqs

This optional function has the following format:

```
VOID APIENTRY Service_GetCfgReqs (SbServiceCfgReqs * pReqs)
```

This function can return configuration requirements that need to be enforced by the Service Broker Manager.

The SbServiceCfgReqs structure is defined as follows:

```
typedef struct {
    ULONG    Size;           /* I: actual size of structure */
    BOOL32   Synchronized; /* 0: synchronization required? */
    ULONG    MinStackSize; /* 0: minimum stack size, 0 or >8192 */
} SbServiceCfgReqs;
```

If the service must run within a single synchronized thread of a synchronized service broker, Synchronized can be set to TRUE. Furthermore, if the minimum stack size of threads assigned to the service or its service broker exceeds 8192 bytes (8K), it can be set in the MinStackSize output variable.

Example

```
VOID APIENTRY Service_GetCfgReqs (SbServiceGetCfgReqs * pReqs)
{
    pReqs->Synchronized = FALSE;
    pReqs->MinStackSize = 32768; /* 32K */
}
```

Service_Init

This optional function has the following format:

```
APIRET APIENTRY Service_Init (SbServiceInit * pInit)
```

This function is called when a service instance is loaded and returns 0 if it was successful, otherwise the loading of the service is discontinued. A structure containing initialization information is passed to the function:

```
typedef struct {
    ULONG    Size;          /* I: actual size of structure      */
    ULONG    Handle;       /* I: handle of service instance    */
    HMODULE  ServiceDLL;   /* I: module handle of service DLL  */
    PSZ     pServiceName; /* I: logical name of service instance */
    HMODULE  BrokerDLL;   /* I: module handle of broker DLL   */
    PSZ     pBrokerName;  /* I: logical name of broker instance */
    PVOID   pInstance;    /* O: user-defined instance pointer */
} SbServiceInit;
```

You can use a user-defined instance pointer, `pInstance`, to store any instance-related information. This pointer is passed to all subsequent service function calls, but can be set in the `Service_Init` function only.

If you need to know the module handle of the service or broker DLL (for example, to load PM resources), store the value of the `ServiceDLL` and `BrokerDLL` fields in order to use it during subsequent service function calls.

The handle of the service instance, `Handle`, is necessary to issue service broker API functions (see “Using the C language service broker API” on page 110).

If this function takes more than 15 seconds (for example, due to user interaction), the `SbbDisableTimeout` service broker API function must be called (see “Disabling timeout” on page 115).

Example

```
APIRET APIENTRY Service_Init (SbServiceInit * pInit)
{
    /* ... user-defined initialization ... */
    return 0;
}
```

Service_Exit

This optional function has the following format:

```
APIRET APIENTRY Service_Exit (SbServiceExit * pExit)
```


This function is called when a service instance is unloaded and returns 0 if it was successful, otherwise the unloading of the service is discontinued. A read-only structure containing exit information is passed to the function:

```
typedef struct {
    ULONG    Size;           /* I: actual size of structure */
    ULONG    Handle;        /* I: handle of service instance */
    PVOID    pInstance;     /* I: user-defined instance pointer */
} SbServiceExit;
```

If this function takes more than 15 seconds (for example, due to user interaction), the SbbDisableTimeout service broker API function must be called (see “Disabling timeout” on page 115).

Example

```
APIRET APIENTRY Service_Exit (SbServiceExit * pExit)
{
    /* ... user-defined exit ... */

    return 0;
}
```

Service_Start

This optional function has the following format:

```
APIRET APIENTRY Service_Start (SbServiceStart * pStart)
```

This function is called when the service is started and returns 0 if it was successful, otherwise the start-up is discontinued.

The SbServiceStart structure is defined as follows:

```
typedef struct {
    ULONG    Size;           /* I: actual size of structure */
    ULONG    Handle;        /* I: handle of service instance */
    PVOID    pInstance;     /* I: user-defined instance pointer */
    PVOID    pSession;      /* I: user-defined session data */
    ULONG    SessionSize;   /* I: size of session data buffer */
} SbServiceStart;
```

SessionSize and pSession refer to the associated service broker's session data.

If this function takes more than 15 seconds (for example, due to user interaction), the SbbDisableTimeout service broker API function must be called (see "Disabling timeout" on page 115).

Example

```
APIRET APIENTRY Service_Start (SbServiceStart * pStart)
{
    /* ... starting ... */
    return 0;
}
```

Service_Stop

This optional function has the following format:

```
APIRET APIENTRY Service_Stop (SbServiceStop * pStop)
```

This function is called when the service is stopped and returns 0 if it was successful, otherwise the shut-down is discontinued.

The SbServiceStop structure is defined as follows:

```
typedef struct {
    ULONG    Size;           /* I: actual size of structure */
    ULONG    Handle;        /* I: handle of service instance */
    PVOID    pInstance;     /* I: user-defined instance pointer */
    PVOID    pSession;      /* I: user-defined session data */
    ULONG    SessionSize;   /* I: size of session data buffer */
} SbServiceStop;
```

If this function takes more than 15 seconds (for example, due to user interaction) the SbbDisableTimeout service broker API function must be called (see "Disabling timeout" on page 115).

Example

```
APIRET APIENTRY Service_Stop (SbServiceStop * pStop)
{
    /* ... stopping ... */
    return 0;
}
```

Service_SetupCfg

This optional function has the following format:

```
APIRET APIENTRY Service_SetupCfg (SbServiceCfg * pCfg)
```

This function is called when the Settings notebook of an already loaded service instance is opened. You can add user-defined pages to the notebook. The function returns 0 if it was successful. A read-only structure containing the notebook window handle is passed to the function:

```
typedef struct {
    ULONG    Size;          /* I: actual size of structure */
    ULONG    Handle;        /* I: handle of service instance */
    PVOID    pInstance;     /* I: user-defined instance pointer */
    HWND     Notebook;      /* I: handle of configuration notebook */
} SbServiceCfg;
```

See the *OS/2 Warp, Version 3 Presentation Manager Programming Guide, Advanced Topics*, for details on notebook programming.

There are three predefined window messages that added notebook pages may handle:

Message	Description
WM_SB_SERV_SET_FOCUS	The notebook page receives the focus.
WM_SB_SERV_COLLECT_DATA	The user pressed the notebook's OK button. The notebook page can save any user-defined configuration values, for example, by using the profile functions of the service broker API (see "Using the C language service broker API" on page 110). It must respond by returning either TRUE to signal successful completion, or FALSE in case of errors (for example, if wrong user input was detected). In the latter case, the notebook page then receives the WM_SB_SERV_SET_FOCUS_ERROR window message.
WM_SB_SERV_SET_FOCUS_ERROR	The notebook page receives the focus due to an error in the handling of the WM_SB_SERV_COLLECT_DATA window message.

Example

```
APIRET APIENTRY Service_SetupCfg (SbServiceCfg * pCfg)
{
    /* ... add user-defined notebook pages ... */

    return 0;
}
```

Service functions

The syntax of a service function is as follows:

```
LONG APIENTRY AFunction (SbFuncInfo * pInfo,
                        PVOID      pData,
                        PULONG     pDataSize,
                        ULONG      MaxOutDataSize)
```

A service DLL can provide several service functions. The length of a service function name is limited to 30 characters, otherwise the function cannot be called by a service requester.

Parameter	I/O	Description
pInfo	I	A read-only structure containing information about the service function's environment (for example, the field pSession points to the session buffer allocated by Broker_Logon).
pData	I/O	Points to a data buffer; as input, it contains input data of size *pDataSize. The service function can copy MaxOutDataSize bytes of its output data to pData.
pDataSize	I/O	As input, it contains the size of input data; as output, it must contain the size of written output data or 0.
MaxOutDataSize	I	Maximum size of output data.

The SbFuncInfo structure is defined as follows:

```
typedef struct {
    ULONG    Size;           /* I: actual size of structure      */
    ULONG    Handle;        /* I: handle of service instance    */
    HAB      Hab;           /* I: anchor block of service thread */
    HMQ      Hmq;           /* I: message queue of service thread */
    PVOID    pInstance;     /* I: user-defined instance pointer  */
    PVOID    pSession;      /* I: user-defined session buffer    */
    ULONG    SessionSize;   /* I: user-defined size of session buffer */
} SbFuncInfo;
```

To accomplish any PM tasks within a service function, handles of anchor block and message queue are provided in the Hab and Hmq fields. Any PM operations that require a standard PM message loop (such as displaying a dialog box or a window in a service function) are not allowed, because they can affect the Service Broker Manager.

A service function can return a user-defined return code that must be greater than or equal to 0, or standard error codes below 0:

Standard Error Code	Description
SB_FUNC_RC_MEMORY (-1)	Not enough memory to complete the operation.
SB_FUNC_RC_OVERFLOW (-2)	Buffer is too small to write output data.
SB_FUNC_RC_INVALID_DATA (-3)	Input data cannot be recognized, or is invalid.

Output data is returned to the caller only if the return code is greater than or equal to 0. Standard error codes cause the function call to fail.

Example

```
#define STRING_RESULT "This is a string result"

LONG APIENTRY AFunction (SbFuncInfo * pInfo,
                        PVOID          pData,
                        PULONG         pDataSize,
                        ULONG          MaxOutDataSize)
{
    unsigned OutputSize = strlen(STRING_RESULT) + 1;
                                /* size of output data */
    if (*pDataSize == 0)         /* invalid input data? */
        return SB_FUNC_RC_INVALID_DATA;

    if (OutputSize > MaxOutDataSize) /* buffer too small? */
        return SB_FUNC_RC_OVERFLOW;

    memcpy(pData, STRING_RESULT, OutputSize); /* copy result string */
    *pDataSize = OutputSize;                 /* set size of output */

    return 456;                               /* user-defined return code */
}
```

Implementing a service requester

This section describes the function and implementation of a service requester.

To invoke a service function, the following must be specified:

- The logical name of a service broker
- The logical name of a service
- The name of a service function
- Any optional parameters for this service function

Using the C language service requester API

Service functions can be executed using the service requester API. Service requester API function names start with the prefix Sbr.

There are these API functions:

- SbrCallService
- SbrCallServiceWithRetry
- SbrStartBroker
- SbrStartBrokerWithInfo
- SbrStopBroker
- SbrStartService
- SbrStopService
- SbrGetErrorMessage

Necessary definitions are included in the header file EXMP3FRE.H, which requires the standard OS/2 header file OS2.H to be included first. Programs using the service requester API must be linked with the supplied library EXMP3KRE.LIB. A sample service requester, SAMPREQ.C, is shown in “Sample C language service requester (SAMPREQ.C)” on page 150.

Calling a service function

Use the SbrCallService function to call a service function. If the specified service broker or service is not active, the Service Broker Manager starts it before calling the service function.

```
APIRET APIENTRY SbrCallService (PSZ   pBroker,  
                                PSZ   pService,  
                                PSZ   pFunction,  
                                PVOID  pInData,  
                                ULONG  InDataSize,  
                                PVOID  pOutData,  
                                PULONG pOutDataSize,  
                                PLONG  pRC,  
                                ULONG  TimeOut)
```

Parameter	I/O	Description
pBroker	I	Logical name of service broker (not case-sensitive, limited to 8 characters).
pService	I	Logical name of service (not case-sensitive, limited to 8 characters).
pFunction	I	Name of service function (case-sensitive, limited to 30 characters).
pInData	I	Address of a buffer that contains input data or NULL.
InDataSize	I	Size of input data or 0.
pOutData	O	Address of a buffer for output data.
pOutDataSize	I/O	As input, it contains the size of the output buffer. As output, it contains the actual size of output data written to pOutData or 0.
pRC	O	User-defined return code of service function.
TimeOut	I	Timeout in milliseconds or SB_REQ_WAIT_INDEFINITE for no timeout. The TimeOut parameter limits the amount of time that a thread blocks on a SbrCallService call. If the time limit is reached before the service function call could be completed, SB_REQ_RC_TIMEOUT is returned.

Return Code	Description
SB_REQ_RC_OK (0)	Operation completed successfully.
SB_REQ_RC_MEMORY (1)	Not enough memory.
SB_REQ_RC_INVALID_ARGS (2)	Invalid arguments specified.
SB_REQ_RC_SERVICE_NOTFOUND (3)	The specified service is not available.
SB_REQ_RC_TIMEOUT (4)	The operation has not been completed in time.
SB_REQ_RC_BUFFER_OVERFLOW (7)	The provided output data buffer is too small and was truncated.
SB_REQ_RC_FUNCTION_NOTFOUND (8)	The specified function does not exist.
SB_REQ_RC_FUNCTION_LOAD (9)	The specified function could not be loaded.
SB_REQ_RC_FUNCTION_TRAP_ACCESS (10)	The specified function trapped due to an access violation.
SB_REQ_RC_FUNCTION_TRAP_INSTR (11)	The specified function trapped due to an instruction that is not permitted.
SB_REQ_RC_FUNCTION_TRAP_FP (12)	The specified function trapped due to a floating point exception.
SB_REQ_RC_FUNCTION_MEMORY (13)	The specified function does not have enough memory.
SB_REQ_RC_FUNCTION_OVERFLOW (14)	The provided output data buffer is too small for the specified function.
SB_REQ_RC_FUNCTION_DATA (15)	Wrong service function parameters specified.

Return Code	Description
SB_REQ_RC_MGR_NOTFOUND (17)	The Service Broker Manager is not running.
SB_REQ_RC_MGR_BUSY (18)	The Service Broker Manager is currently busy.
SB_REQ_RC_MGR_CONNECT (19)	The connection to the Service Broker Manager is broken.
SB_REQ_RC_BROKER_NOTFOUND (20)	The specified service broker is not available.
SB_REQ_RC_BROKER_START (21)	The specified service broker could not be started.
SB_REQ_RC_SERVICE_START (23)	The specified service could not be started.
SB_REQ_RC_MGR_STOP (25)	The Service Broker Manager has been stopped.
SB_REQ_RC_SERVICE_DISABLED (26)	The specified service is disabled and cannot be used.
SB_REQ_RC_BROKER_DISABLED (27)	The specified service broker is disabled and cannot be used.

Example

```
#define BUFFER_SIZE 512

char    buffer    [BUFFER_SIZE];    /* buffer */
PSZ     inputData;    /* input data */
ULONG   outSize;    /* size of output */
LONG    outRC;
APIRET  rc;

outSize = BUFFER_SIZE;    /* size of output buffer */
inputData = "This is input data";    /* sample input data */

rc = SbrCallService ("MYBROK",
                    "MYSERV",
                    "DoSomething",
                    (PVOID) inputData,
                    strlen(inputData) + 1,
                    buffer,
                    &outSize,
                    &outRC,
                    5000);    /* 5 secs timeout */

if (rc != SB_REQ_RC_OK)
{
    /* ... error handling ... */
}
```


Calling a service function with retry

The SbrCallServiceWithRetry function is equivalent to the SbrCallService function, except that, in case of an error, a message box describing the reason is displayed. The user can cancel or retry the operation.

```
APIRET WINAPI SbrCallServiceWithRetry (PSZ  pBroker,
                                       PSZ  pService,
                                       PSZ  pFunction,
                                       PVOID pInData,
                                       ULONG InDataSize,
                                       PVOID pOutData,
                                       PULONG pOutDataSize,
                                       PLONG  pRC,
                                       ULONG TimeOut)
```

This function can only be called by a thread that has already initialized its PM interface (WinInitialize, WinCreateMsgQueue). However, since the calling thread is blocked until the result is available, this function must not be used inside a PM message loop.

This function is used by the standard service requester (see “Using the standard service requester” on page 144) and the FlowMark service requester (see “FlowMark requester” on page 34).

Example

```
...
WinInitialize (...);
WinCreateMsgQueue (...);
...
SbrCallServiceWithRetry (...);
...
WinDestroyMsgQueue (...);
WinTerminate (...);
...
```

Starting a service broker

Use the SbrStartBroker function to start an inactive service broker:

```
APIRET WINAPI SbrStartBroker (PSZ  pBroker,
                              ULONG TimeOut)
```

Parameter	I/O	Description
pBroker	I	Logical name of service broker (not case-sensitive, limited to 8 characters).
TimeOut	I	Timeout in milliseconds or SB_REQ_WAIT_INDEFINITE for no timeout. The TimeOut parameter limits the amount of time a thread blocks on a SbrStartBroker call. If the time limit is reached before the service broker could be started, SB_REQ_RC_TIMEOUT is returned.

Return Code	Description
SB_REQ_RC_OK (0)	Operation completed successfully.
SB_REQ_RC_MEMORY (1)	Not enough memory.
SB_REQ_RC_INVALID_ARGS (2)	Invalid arguments specified.
SB_REQ_RC_TIMEOUT (4)	The operation has not been completed in time.
SB_REQ_RC_MGR_NOTFOUND (17)	The Service Broker Manager is not running.
SB_REQ_RC_MGR_BUSY (18)	The Service Broker Manager is currently busy.
SB_REQ_RC_MGR_CONNECT (19)	The connection to the Service Broker Manager is broken.
SB_REQ_RC_BROKER_NOTFOUND (20)	The specified service broker is not available.
SB_REQ_RC_BROKER_START (21)	The specified service broker could not be started.
SB_REQ_RC_MGR_STOP (25)	The Service Broker Manager has been stopped.
SB_REQ_RC_BROKER_DISABLED (27)	The specified service broker is disabled and cannot be used.

Example

```
APIRET rc;

...

rc = SbrStartBroker ("MYBROK", SB_REQ_WAIT_INDEFINITE);

if (rc != SB_REQ_RC_OK)
{
    /* ... error handling ... */
}
```

Starting a service broker with additional information

Use the SbrStartBroker function to start an inactive service broker and pass logon information to the Service Broker Manager:

```
APIRET APIENTRY SbrStartBrokerWithInfo(PSZ pBroker,  
                                       PVOID pInfoArea,  
                                       ULONG InfoAreaLength,  
                                       ULONG TimeOut)
```

This function basically works like SbrStartBroker but you can pass additional information to the structure SbBrokerLogon that is used in the Broker_Logon function.

If you work with multiple applications that are served by service brokers and need to log on to any of these applications, you usually have a logon panel displayed for the respective application. To suppress the display of logon panels, start the service brokers for the applications with SbrBrokerWithInfo. So you can pass the logon information directly to the service brokers which can then perform an *unattended* logon.

Parameter	I/O	Description
pBroker	I	Logical name of service broker (not case-sensitive, limited to 8 characters).
pInfoArea	I	Information area that is passed to the SbrBrokerLogon function.
InfoAreaLength	I	Length of the information area.
TimeOut	I	Timeout in milliseconds or SB_REQ_WAIT_INDEFINITE for no timeout. The TimeOut parameter limits the amount of time a thread blocks on a SbrStartBroker call. If the time limit is reached before the service broker could be started, SB_REQ_RC_TIMEOUT is returned.

For information about the return codes of this function, refer to “Starting a service broker” on page 129.

Stopping a service broker

Use the SbrStopBroker function to stop an active service broker:

```
APIRET APIENTRY SbrStopBroker (PSZ pBroker,  
                               ULONG TimeOut)
```

Parameter	I/O	Description
pBroker	I	Logical name of service broker (not case-sensitive, limited to 8 characters).
TimeOut	I	Timeout in milliseconds or SB_REQ_WAIT_INDEFINITE for no timeout. The TimeOut parameter limits the amount of time a thread blocks on a SbrStopBroker call. If the time limit is reached before the service broker could be stopped, SB_REQ_RC_TIMEOUT is returned.

Return Code	Description
SB_REQ_RC_OK (0)	Operation completed successfully.
SB_REQ_RC_MEMORY (1)	Not enough memory.
SB_REQ_RC_INVALID_ARGS (2)	Invalid arguments specified.
SB_REQ_RC_TIMEOUT (4)	The operation has not been completed in time.
SB_REQ_RC_MGR_NOTFOUND (17)	The Service Broker Manager is not running.
SB_REQ_RC_MGR_BUSY (18)	The Service Broker Manager is currently busy.
SB_REQ_RC_MGR_CONNECT (19)	The connection to the Service Broker Manager is broken.
SB_REQ_RC_BROKER_NOTFOUND (20)	The specified service broker is not available.
SB_REQ_RC_BROKER_STOP (22)	The specified service broker could not be stopped.
SB_REQ_RC_MGR_STOP (25)	The Service Broker Manager has been stopped.
SB_REQ_RC_BROKER_DISABLED (27)	The specified service broker is disabled and cannot be used.

Example

```

APIRET rc;

...

rc = SbrStopBroker ("MYBROK", SB_REQ_WAIT_INDEFINITE);

if (rc != SB_REQ_RC_OK)
{
    /* ... error handling ... */
}

```

Starting a service

Use the SbrStartService function to start an inactive service:

```
APIRET APIENTRY SbrStartService (PSZ pBroker,  
                                PSZ pService,  
                                ULONG TimeOut)
```

Parameter	I/O	Description
pBroker	I	Logical name of service broker (not case-sensitive, limited to 8 characters).
pService	I	Logical name of service (not case-sensitive, limited to 8 characters).
TimeOut	I	Timeout in milliseconds or SB_REQ_WAIT_INDEFINITE for no timeout. The TimeOut parameter limits the amount of time a thread blocks on a SbrStartService call. If the time limit is reached before the service could be started, SB_REQ_RC_TIMEOUT is returned.

Return Code	Description
SB_REQ_RC_OK (0)	Operation completed successfully.
SB_REQ_RC_MEMORY (1)	Not enough memory.
SB_REQ_RC_INVALID_ARGS (2)	Invalid arguments specified.
SB_REQ_RC_SERVICE_NOTFOUND (3)	The specified service is not available.
SB_REQ_RC_TIMEOUT (4)	The operation has not been completed in time.
SB_REQ_RC_MGR_NOTFOUND (17)	The Service Broker Manager is not running.
SB_REQ_RC_MGR_BUSY (18)	The Service Broker Manager is currently busy.
SB_REQ_RC_MGR_CONNECT (19)	The connection to the Service Broker Manager is broken.
SB_REQ_RC_BROKER_NOTFOUND (20)	The specified service broker is not available.
SB_REQ_RC_SERVICE_START (23)	The specified service could not be started.
SB_REQ_RC_MGR_STOP (25)	The Service Broker Manager has been stopped.
SB_REQ_RC_SERVICE_DISABLED (26)	The specified service is disabled and cannot be used.
SB_REQ_RC_BROKER_DISABLED (27)	The specified service broker is disabled and cannot be used.

Example

```
APIRET rc;

...

rc = SbrStartService ("MYBROK", "MYSERV", SB_REQ_WAIT_INDEFINITE);

if (rc != SB_REQ_RC_OK)
{
    /* ... error handling ... */
}
```

Stopping a service

Use the SbrStopService function to stop an active service:

```
APIRET APIENTRY SbrStopService (PSZ pBroker,
                                PSZ pService,
                                ULONG TimeOut)
```

Parameter	I/O	Description
pBroker	I	Logical name of service broker (not case-sensitive, limited to 8 characters).
pService	I	Logical name of service (not case-sensitive, limited to 8 characters).
TimeOut	I	Timeout in milliseconds or SB_REQ_WAIT_INDEFINITE for no timeout. The TimeOut parameter limits on the amount of time a thread blocks on a SbrStopService call. If the time limit is reached before the service could be stopped, SB_REQ_RC_TIMEOUT is returned.

Return Code	Description
SB_REQ_RC_OK (0)	Operation completed successfully.
SB_REQ_RC_MEMORY (1)	Not enough memory.
SB_REQ_RC_INVALID_ARGS (2)	Invalid arguments specified.
SB_REQ_RC_SERVICE_NOTFOUND (3)	The specified service is not available.
SB_REQ_RC_TIMEOUT (4)	The operation has not been completed in time.
SB_REQ_RC_MGR_NOTFOUND (17)	The Service Broker Manager is not running.
SB_REQ_RC_MGR_BUSY (18)	The Service Broker Manager is currently busy.
SB_REQ_RC_MGR_CONNECT (19)	The connection to the Service Broker Manager is broken.
SB_REQ_RC_BROKER_NOTFOUND (20)	The specified service broker is not available.
SB_REQ_RC_SERVICE_STOP (24)	The specified service could not be stopped.

Return Code	Description
SB_REQ_RC_MGR_STOP (25)	The Service Broker Manager has been stopped.
SB_REQ_RC_SERVICE_DISABLED (26)	The specified service is disabled and cannot be used.
SB_REQ_RC_BROKER_DISABLED (27)	The specified service broker is disabled and cannot be used.

Example

```
APIRET rc;

...

rc = SbrStopService ("MYBROK", "MYSERV", SB_REQ_WAIT_INDEFINITE);

if (rc != SB_REQ_RC_OK)
{
    /* ... error handling ... */
}
```

Retrieving an error message

Use the `SbrGetErrorMessage` function to retrieve the error message for a particular error code:

```
APIRET APIENTRY SbrGetErrorMessage (APIRET Error,
                                     PSZ pBuffer,
                                     ULONG BufferSize)
```

Parameter	I/O	Description
Error	I	Error code of requested error message (see the return codes of service requester API functions).
pBuffer	O	Address of a buffer where error message is written.
BufferSize	I	Size of the buffer including trailing 0.

Return Code	Description
SB_REQ_RC_OK (0)	Operation completed successfully.
SB_REQ_RC_INVALID_ARGS (2)	Invalid arguments specified.
SB_REQ_RC_BUFFER_OVERFLOW (7)	The provided buffer was too small. Truncated error message was copied.
SB_REQ_RC_MESSAGE_NOTFOUND (16)	The error message could not be found.

Example

```
#define BUFFER_SIZE 512

char    buffer    [BUFFER_SIZE];
APIRET  errorCode;
APIRET  rc;

...
errorCode = SbrCallService (...);
...
if (errorCode != SB_REQ_RC_OK)
{
    rc = SbrGetErrorMessage (errorCode, buffer, BUFFER_SIZE);
    ...
}
```

Using the REXX language service requester API

Service functions can be executed using the REXX service requester API. REXX service requester API function names start with the prefix RxSbr.

There are these API functions:

- RxSbrLoadFuncs
- RxSbrDropFuncs
- RxSbrCallService
- RxSbrStartBroker
- RxSbrStopBroker
- RxSbrStartService
- RxSbrStopService
- RxSbrGetErrorMessage

By default, a REXX DLL file, EXMP3KRX.DLL, is automatically installed. To use the REXX API functions, the service requester functions can be loaded by a REXX program at each invocation.

A sample REXX service requester, EXMP3SRX.CMD, is shown in “Sample REXX language service requester (EXMP3SRX.CMD)” on page 152.

Loading API functions

Use the RxSbrLoadFuncs function to load and register all REXX service requester API functions.

```
RC = RxSbrLoadFuncs()
```


Return Code	Description
0	Operation completed successfully.
<> 0	Operation failed.

Example

```
call RxFuncAdd 'RxSbrLoadFuncs', 'SBREQRX', 'RxSbrLoadFuncs'
rv = RxSbrLoadFuncs()
if rv \= 0 then do
  say "Service Requester API functions could not be loaded: rc =" rv
  exit 1
end
```

Unloading API functions

Use the RxSbrDropFuncs function to deregister and unload all REXX service requester API functions.

```
RC = RxSbrDropFuncs()
```

Return Code	Description
0	Operation completed successfully.
<> 0	Operation failed.

Example

```
rv = RxSbrDropFuncs()
if rv \= 0 then do
  say "Service Requester API functions could not be unloaded: rc =" rv
  exit 1
end
```

Calling a service function

Use the RxSbrCallService function in your REXX program to call service functions that handle string arguments as input and output parameters. Service functions expecting or returning binary data must be called via the C language service requester API. If the specified service broker or service is not active, the Service Broker Manager starts it before calling the service function.

```

RC = RxSbrCallService(Broker,
                      Service,
                      Function,
                      Input,
                      'Output',
                      'Result'
                      [,TimeOut])

```

Parameter	I/O	Description
Broker	I	Logical name of service broker (not case-sensitive, limited to 8 characters).
Service	I	Logical name of service (not case-sensitive, limited to 8 characters).
Function	I	Name of service function (case-sensitive, limited to 30 characters).
Input	I	Input string.
<i>Output</i>	O	The name of a REXX variable into which the output string is to be returned (maximum output data size is currently 512 bytes).
<i>Result</i>	O	The name of a REXX variable into which the function's return code is to be returned.
TimeOut	I	Timeout in milliseconds or 0 for no timeout (optional). The TimeOut parameter limits the amount of time the program blocks on a RxSbrCallService call. If the time limit is reached before the service function call could be completed, 4 is returned.

Return Code	Description
0	Operation completed successfully.
1	Not enough memory, or not enough memory to store output variables <i>Output</i> or <i>Result</i> .
2	Invalid arguments specified.
3	The specified service is not available.
4	The operation has not been completed in time.
7	The internal output buffer is too small (currently 512 bytes).
8	The specified function does not exist.
9	The specified function could not be loaded.
10	The specified function trapped due to an access violation.
11	The specified function trapped due to an instruction that is not permitted.
12	The specified function trapped due to a floating point exception.
13	The specified function does not have enough memory.
14	The provided output data buffer is too small for the specified function.
15	Wrong service function parameters specified.

Return Code	Description
17	The Service Broker Manager is not running.
18	The Service Broker Manager is currently busy.
19	The connection to the Service Broker Manager is broken.
20	The specified service broker is not available.
21	The specified service broker could not be started.
23	The specified service could not be started.
25	The Service Broker Manager has been stopped.
26	The specified service is disabled and cannot be used.
27	The specified service broker is disabled and cannot be used.

Example

```
rv = RxSbrCallService('MYBROK',
                    'MYSERV',
                    'DoSomething',
                    'This is input data',
                    'output',
                    'result')

if rv = 0 then do
  say 'Output is:' output
  say 'Result is:' result
end
else
  say 'Service could not be called (error 'rv')'
```

Starting a service broker

Use the RxSbrStartBroker function in your REXX program to start an inactive service broker:

```
RC = RxSbrStartBroker(Broker,
                    [,TimeOut])
```

Parameter	I/O	Description
Broker	I	Logical name of service broker (not case-sensitive, limited to 8 characters).
TimeOut	I	Timeout in milliseconds or 0 for no timeout (optional). The TimeOut parameter limits the amount of time the program blocks on a RxSbrStartBroker call. If the time limit is reached before the service broker could be started, 4 is returned.

Return Code	Description
0	Operation completed successfully.
1	Not enough memory.
2	Invalid arguments specified.
4	The operation has not been completed in time.
17	The Service Broker Manager is not running.
18	The Service Broker Manager is currently busy.
19	The connection to the Service Broker Manager is broken.
20	The specified service broker is not available.
21	The specified service broker could not be started.
25	The Service Broker Manager has been stopped.
27	The specified service broker is disabled and cannot be used.

Example

```
rv = RxSbrStartBroker('MYBROK')
if rv \= 0 then do
  /* ... error handling ... */
end
```

Stopping a service broker

Use the RxSbrStopBroker function in your REXX program to stop an active service broker:

```
RC = RxSbrStopBroker(Broker,
                    [,TimeOut])
```

Parameter	I/O	Description
Broker	I	Logical name of service broker (not case-sensitive, limited to 8 characters).
TimeOut	I	Timeout in milliseconds or 0 for no timeout (optional). The TimeOut parameter limits the amount of time the program blocks on a RxSbrStopBroker call. If the time limit is reached before the service broker could be stopped, 4 is returned.

Return Code	Description
0	Operation completed successfully.
1	Not enough memory.
2	Invalid arguments specified.
4	The operation has not been completed in time.
17	The Service Broker Manager is not running.

Return Code	Description
18	The Service Broker Manager is currently busy.
19	The connection to the Service Broker Manager is broken.
20	The specified service broker is not available.
22	The specified service broker could not be stopped.
25	The Service Broker Manager has been stopped.
27	The specified service broker is disabled and cannot be used.

Example

```
rv = RxSbrStopBroker('MYBROK')
if rv \= 0 then do
  /* ... error handling ... */
end
```

Starting a service

Use the RxSbrStartService function in your REXX program to start an inactive service:

```
RC = RxSbrStartService(Broker,
                       Service,
                       [,TimeOut])
```

Parameter	I/O	Description
Broker	I	Logical name of service broker (not case-sensitive, limited to 8 characters).
Service	I	Logical name of service (not case-sensitive, limited to 8 characters).
TimeOut	I	Timeout in milliseconds or 0 for no timeout (optional). The TimeOut parameter limits the amount of time the program blocks on a RxSbrStartService call. If the time limit is reached before the service could be started, 4 is returned.

Return Code	Description
0	Operation completed successfully.
1	Not enough memory.
2	Invalid arguments specified.
3	The specified service is not available.
4	The operation has not been completed in time.
17	The Service Broker Manager is not running.
18	The Service Broker Manager is currently busy.
19	The connection to the Service Broker Manager is broken.

Return Code	Description
20	The specified service broker is not available.
23	The specified service could not be started.
25	The Service Broker Manager has been stopped.
26	The specified service is disabled and cannot be used.
27	The specified service broker is disabled and cannot be used.

Example

```
rv = RxSbrStartService('MYBROK','MYSERV')
if rv \= 0 then do
  /* ... error handling ... */
end
```

Stopping a service

Use the RxSbrStopService function in your REXX program to stop an active service:

```
RC = RxSbrStopService(Broker,
                      Service,
                      [,TimeOut])
```

Parameter	I/O	Description
Broker	I	Logical name of service broker (not case-sensitive, limited to 8 characters).
Service	I	Logical name of service (not case-sensitive, limited to 8 characters).
TimeOut	I	Timeout in milliseconds or 0 for no timeout (optional). The TimeOut parameter limits the amount of time the program blocks on a RxSbrStopService call. If the time limit is reached before the service could be stopped, 4 is returned.

Return Code	Description
0	Operation completed successfully.
1	Not enough memory.
2	Invalid arguments specified.
3	The specified service is not available.
4	The operation has not been completed in time.
17	The Service Broker Manager is not running.
18	The Service Broker Manager is currently busy.
19	The connection to the Service Broker Manager is broken.
20	The specified service broker is not available.

Return Code	Description
24	The specified service could not be stopped.
25	The Service Broker Manager has been stopped.
26	The specified service is disabled and cannot be used.
27	The specified service broker is disabled and cannot be used.

Example

```
rv = RxSbrStopService('MYBROK', 'MYSERV')
if rv \= 0 then do
  /* ... error handling ... */
end
```

Retrieving an error message

Use the `RxSbrGetErrorMessage` function to retrieve the error message for a particular error code in your REXX program:

```
RC = RxSbrGetErrorMessage(Error, 'Message')
```

Parameter	I/O	Description
Error	I	Error code of requested error message (see the return codes of service requester API functions).
Message	O	The name of a REXX variable into which the specified error message is to be returned (maximum message size is currently 512 bytes).

Return Code	Description
0	Operation completed successfully.
1	Not enough memory to store output variable <i>Message</i> .
2	Invalid arguments specified.
7	The internal buffer is too small (currently 512 bytes). Truncated error message was copied.
16	The error message could not be found.

Example

```
rv = RxSbrCallService(...)
if rv = 0 then do
  ...
end
else do
  rc = RxSbrGetErrorMessage(rv, 'message')
  if rc = 0 then
    say message
  else
    say 'Service could not be called (error 'rv')'
  end
end
```

Using the standard service requester

To test your service broker or service, you can use the program EXMP3FFR.EXE.

Note: Use EXMP3FFR if your service function requires textual parameters only. If the parameters are based on access to the FlowMark container API (for example, by receiving the session ID), you can only test your services by starting the FlowMark service requester as a FlowMark program activity within a process model. See Chapter 2, “Using the Service Broker Manager for OS/2” on page 7 for details.

Enter **exmp3ffr**, followed by at least three parameters, at the OS/2 command prompt:

```
►—exmp3ffr—/q—broker—service—function—params—►
```

Parameter	Description
<i>/q</i>	Quiet; output data and result code, or error message is not displayed (optional).
<i>broker</i>	Logical name of the service broker that provides the specific service (not case-sensitive).
<i>service</i>	Logical name of the service providing the requested function (not case-sensitive).
<i>function</i>	Name of the service function (case-sensitive). This function must be exported by the service DLL.
<i>params</i>	Textual parameters that must be passed to the service function (optional). Refer to the documentation of the used service brokers and services for further details.

EXMP3FFR returns the return code of the service function or 255, if the function failed. The service function's output data and result code are displayed in a window unless /Q was specified.

Example

```
exmp3ffr mybrok myserv DoSomething 'this is a parameter string'
```


Using the standard external controller

You can use the program EXMP3UCT.EXE to control the Service Broker Manager. It allows you to:

- Start a service broker
- Start a service
- Stop a service broker
- Stop a service

The syntax for invoking this program is:

```
►► exmp3uct [ /q ] [ start | stop ] broker service ►►
```

Parameter	Description
/q	Quiet; error messages are not displayed.
broker	The logical name of a service broker. This value is not case-sensitive.
service	The logical name of a service. This value is not case-sensitive.

Starting a service of a service broker that has not been started fails. EXMP3UCT returns 0 to signal successful completion, or 255, if the action failed.

Example

```
exmp3uct start mybrok myserv
```

Debugging service brokers and services

This section describes how to debug service broker and service DLLs using the debugger supplied with the IBM VisualAge for C++ (IPMD.EXE).

Since service broker and service DLLs are loaded by the Service Broker Manager, the only way to debug these DLLs is to debug the Service Broker Manager:

```
►► ipmd exmp3uup ►►
```

Use the **run** command to start execution. If a DLL has been successfully loaded by the Service Broker Manager, its name appears in the **Components** list box of the **Debug Session Control** window, provided that debug information is included in the DLL file (use the /Ti+ compiler option of the IBM VisualAge for C++ to generate debug information). Click on the plus icon to the left of the DLL file name to display the list of contained object files, double-click on the object of your choice, and set appropriate breakpoints. Use the **run** command again to continue execution.

To debug broker or service initialization code (Broker_Init or Service_Init) a load occurrence breakpoint must be set for the particular DLL before starting execution of the Service Broker Manager. To set such a breakpoint, select the **Breakpoints** menu from the **Debug Session Control** window, select **Load occurrence...** and specify the name of your DLL. Use the **run** command to start execution. When the specified DLL is being loaded, the above breakpoint is encountered, and the DLL's name appears. Continue as described above.

Sample files

When you install the Service Broker Manager toolkit, the following files are provided. Use these files as templates for your own service brokers, services and service requesters. By default, these files are installed in subdirectories of the EXM\SBM\SAMPLES\TOOLKIT directory.

Sample service broker DLL (SAMPBROK.C)

```

/*****/
/* */
/* Licensed Materials - Property of IBM */
/* */
/* "Restricted Materials of IBM" */
/* */
/* 5697-216 */
/* */
/* (C) Copyright IBM Corp. 1995, 1996 All Rights Reserved */
/* */
/* US Government Users Restricted Rights - Use, duplication or */
/* disclosure restricted by GSA ADP Schedule Contract with */
/* IBM Corp. */
/* */
/*****/
/* */
/* File: SAMPBROK.C */
/* */
/* Purpose: Sample Broker DLL */
/* */
/*****/

#include <os2.h>
#include <string.h>
#include <stdlib.h>
#include "exp3cbr.h"

/*****/
/* */
/* Standard Broker DLL Interface: */
/* */
/* Mandatory: Broker_GetDllVersion */
/* Broker_GetVersion */
/* */
/* Optional: Broker_Init */
/* Broker_Exit */
/* Broker_Logon */
/* Broker_Logoff */
/* Broker_GetCfgReqs */
/* Broker_SetupCfg */
/* */
/*****/

```

```

/* Broker name & version */
#define BROKER_NAME    "MyBroker"
#define BROKER_VERSION 123

/*****
/* Return DLL version
*****/

ULONG APIENTRY Broker_GetDllVersion (void)
{
    return SB_BROKER_DLLVERSION;
}

/*****
/* Return broker version information
*****/

VOID APIENTRY Broker_GetVersion (PSbBrokerVersion pVer)
{
    pVer->pName    = BROKER_NAME;
    pVer->Version = BROKER_VERSION;
}

/*****
/* Called when broker is loaded:  0 ..... Successful
/*                               != 0 ... Error
*****/

APIRET APIENTRY Broker_Init (PSbBrokerInit pInit)
{
    /* ... user-defined initialization ... */

    SbbLog (pInit->Handle, SB_BROK_LOG_LEVEL_2, "Initialized!");

    return 0;
}

/*****
/* Called when broker is unloaded:  0 ..... Successful
/*                                 != 0 ... Error
*****/

APIRET APIENTRY Broker_Exit (PSbBrokerExit pExit)
{
    /* ... user-defined exit ... */

    SbbLog (pExit->Handle, SB_BROK_LOG_LEVEL_2, "Exit!");

    return 0;
}

/*****
/* Called when broker is started:  0 ..... Successful
/*                                != 0 ... Error
*****/

APIRET APIENTRY Broker_Logon (PSbBrokerLogon pLogon)
{
    char * pSession = (char *) malloc (100);
    strcpy (pSession, "This is a session");

    pLogon->pSession    = pSession;
    pLogon->SessionSize = 100;

    SbbLog (pLogon->Handle, SB_BROK_LOG_LEVEL_2, "Logged on!");
}

```

```

    return 0;
}

/*****
/* Called when broker is stopped:    0 ..... Successful    */
/*                               != 0 ... Error            */
*****/

APIRET APIENTRY Broker_Logoff (PSbBrokerLogoff pLogoff)
{
    free (pLogoff->pSession);

    SbbLog (pLogoff->Handle, SB_BROK_LOG_LEVEL_2, "Logged off!");

    return 0;
}

```

Sample service DLL (SAMPSEV.C)

```

/*****
/* Licensed Materials - Property of IBM
/*
/* "Restricted Materials of IBM"
/*
/* 5697-216
/*
/* (C) Copyright IBM Corp. 1995, 1996 All Rights Reserved
/*
/* US Government Users Restricted Rights - Use, duplication or
/* disclosure restricted by GSA ADP Schedule Contract with
/* IBM Corp.
/*
*****/
/* File:          SAMPSEV.C
/*
/* Purpose:       Sample Service DLL
/*
*****/

#include <os2.h>
#include <string.h>
#include "exp3cse.h"

/*****
/*
/* Standard Service DLL Interface:
/*
/* Mandatory:     Service_GetDllVersion
/*                Service_CheckBroker
/*
/* Optional:      Service_Init
/*                Service_Exit
/*                Service_Start
/*                Service_Stop
/*                Service_GetCfgReqs
/*                Service_SetupCfg
/*
*****/

/* Required broker name & version */

#define BROKER_NAME    "MyBroker"
#define BROKER_VERSION 123

```

```

/*****
/* Return DLL version
*/
/*****

ULONG APIENTRY Service_GetDllVersion (void)
{
    return SB_SERVICE_DLLVERSION;
}

/*****
/* Check if broker is compatible: 0 ..... Successful
/*                               != 0 ... Error
*/
/*****

APIRET APIENTRY Service_CheckBroker (PSbBrokerVersion pVer)
{
    return (strcmp(pVer->pName, BROKER_NAME) != 0) ||
           (pVer->Version < BROKER_VERSION);
}

/*****
/* Called when service is loaded: 0 ..... Successful
/*                               != 0 ... Error
*/
/*****

APIRET APIENTRY Service_Init (PSbServiceInit pInit)
{
    /* ... user-defined initialization ... */

    SbbLog (pInit->Handle, SB_BROK_LOG_LEVEL_2, "Initialized!");

    return 0;
}

/*****
/* Called when service is unloaded: 0 ..... Successful
/*                               != 0 ... Error
*/
/*****

APIRET APIENTRY Service_Exit (PSbServiceExit pExit)
{
    /* ... user-defined exit ... */

    SbbLog (pExit->Handle, SB_BROK_LOG_LEVEL_2, "Exit!");

    return 0;
}

/*****
/* Called when service is started: 0 ..... Successful
/*                               != 0 ... Error
*/
/*****

APIRET APIENTRY Service_Start (PSbServiceStart pStart)
{
    /* ... user-defined start-up ... */

    SbbLog (pStart->Handle, SB_BROK_LOG_LEVEL_2, "Started!");

    return 0;
}

/*****
/* Called when service is stopped: 0 ..... Successful
/*                               != 0 ... Error
*/
/*****

```

```

APIRET APIENTRY Service_Stop (PSbServiceStop pStop)
{
    /* ... user-defined stopping ... */

    SbbLog (pStop->Handle, SB_BROK_LOG_LEVEL_2, "Stopped!");

    return 0;
}

/*****
/*
/* Service Functions
/*
/*
*****/

#define STRING_RESULT "This is a string result"

LONG APIENTRY AFunction (PSbFuncInfo pInfo,
                        PVOID      pData,
                        PULONG     pDataSize,
                        ULONG      MaxOutDataSize)
{
    unsigned OutputSize = strlen(STRING_RESULT) + 1; /* size of output data */

    if (*pDataSize == 0) /* invalid input data? */
        return SB_FUNC_RC_INVALID_DATA;

    if (OutputSize > MaxOutDataSize) /* buffer too small? */
        return SB_FUNC_RC_OVERFLOW;

    memcpy(pData, STRING_RESULT, OutputSize); /* copy result string */
    *pDataSize = OutputSize; /* set size of output */

    SbbLog (pInfo->Handle, SB_BROK_LOG_LEVEL_2, "AFunction completed!");

    return 0;
}

LONG APIENTRY AnotherFunction (PSbFuncInfo pInfo,
                              PVOID      pData,
                              PULONG     pDataSize,
                              ULONG      MaxOutDataSize)
{
    *pDataSize = 0; /* no output */

    SbbLog (pInfo->Handle, SB_BROK_LOG_LEVEL_2, "AnotherFunction completed!");

    return 123;
}

```

Sample C language service requester (SAMPREQ.C)

```

/*****
/*
/* Licensed Materials - Property of IBM
/*
/* "Restricted Materials of IBM"
/*
/* 5697-216
/*
/* (C) Copyright IBM Corp. 1995, 1996 All Rights Reserved
/*
/* US Government Users Restricted Rights - Use, duplication or
/* disclosure restricted by GSA ADP Schedule Contract with
/* IBM Corp.
/*
*****/

```

```

/*****
/*
/* File:      SMPREQ.C
/*
/* Purpose:   Sample Service Requester
/*
*****/

#include <os2.h>
#include <stdio.h>
#include <string.h>
#include "exmp3fre.h"

/*****
/* Constants
*****/

#define EXITCODE_ERROR 255
#define BUFFER_SIZE 512

/*****
/* main
*****/

int main (int argc, char * argv [])
{
    char    buffer [BUFFER_SIZE];          /* buffer */

    char *  broker;                       /* service broker */
    char *  service;                      /* service */
    char *  function;                     /* service function */
    char *  args;                         /* function argument */

    ULONG  inSize;                        /* size of input */
    ULONG  outSize;                       /* size of output */
    LONG   outRC;
    APIRET rc;

    /*****
    /* Process arguments
    *****/

    if (argc != 4 && argc != 5) {
        printf ("Usage:  SMPREQ Broker Service Function [\"...\"]\n");
        return EXITCODE_ERROR;
    }

    broker  = argv[1];
    service = argv[2];
    function = argv[3];

    if (argc == 5 && *argv[4] != 0) {
        args = argv[4];
        inSize = strlen (argv[4]) + 1;
    } else {
        args = 0;
        inSize = 0;
    }

    /*****
    /* Call service function
    *****/

    outSize = BUFFER_SIZE;                /* size of output buffer */

    rc = SbrCallService (broker,          /* name of broker */

```

```

        service,          /* name of service */
        function,        /* name of function */
        args,            /* input data */
        inSize,          /* size of "-" */
        buffer,          /* output buffer */
        &outSize,        /* size of "-" */
        &outRC,          /* return code */
        SB_REQ_WAIT_INDEFINITE); /* no timeout */

/*****
/* Handle result */
*****/

if (rc == SB_REQ_RC_OK) {
    if (outSize != 0)
        printf ("Result = '%s'\n", buffer);
    printf ("Return Code = %u\n", outRC);
    return outRC;
} else {
    if (SbrGetErrorMessage (rc, buffer, BUFFER_SIZE) == SB_REQ_RC_OK)
        printf ("Error %u: %s\n", rc, buffer);
    else
        printf ("Error %u: error message could not be retrieved!\n", rc);
    return EXITCODE_ERROR;
}
}

```

Sample REXX language service requester (EXMP3SRX.CMD)

```

/*****
/*
/* Licensed Materials - Property of IBM
/*
/* "Restricted Materials of IBM"
/*
/* 5697-216
/*
/* (C) Copyright IBM Corp. 1995, 1996 All Rights Reserved
/*
/* US Government Users Restricted Rights - Use, duplication or
/* disclosure restricted by GSA ADP Schedule Contract with
/* IBM Corp.
*****/
/*
/* File:          EXMP3SRX.CMD
/*
/* Purpose:      REXX Requester sample; similar to EXMP3FFR.EXE
/*
/* Usage:       EXMP3SRX broker service function params
*****/

parse arg broker service function
/* Check parameters */

if broker = '' | service = '' | function = '' then do
    say 'Usage: EXMP3SRX broker service function params'
    exit 100
end

/* Load functions */

```



```

call RxFuncAdd 'RxSbrLoadFuncs', 'EXMP3KRX', 'RxSbrLoadFuncs'
rv = RxSbrLoadFuncs()
if rv \= 0 then do
    say "Service Requester API functions could not be loaded: rc =" rv
    exit 1
end

/* Call service:  output data      -> variable 'output' */
/*                function return code -> variable 'result' */

rv = RxSbrCallService(broker,service,function,params,'output','result')
if rv = 0 then do
    say 'Output is: "'output'"'
    say 'Result is: 'result
    exit result
end
else do
    rv2 = RxSbrGetErrorMessage(rv,'message')
    if rv2 = 0 then
        say message
    else
        say 'Service could not be called (error 'rv')'
    exit 101
end

/* Registration error */

RegistrationError:
    say 'Necessary functions could not be loaded!'
    exit 102

```

Chapter 7. The Service Broker Manager for Windows

This chapter gives you some general information about the Service Broker Manager for Windows 3.1. It also explains how to use the Service Broker Manager for Windows and how to implement service brokers and services.

About the Service Broker Manager for Windows

The concept of the Service Broker Manager for Windows, like that of the Service Broker Manager on OS/2, includes the following components:

- Service Broker Manager
The Service Broker Manager controls the operation of service broker sessions. This includes the interaction between service requester and services, between service broker and services, and also the initialization of the service brokers and services.
- Service broker
The service broker establishes and maintains a logon session with the base product (for example, Lotus Notes).
- Services
A service interfaces to the integrated product. A service function receives the user data from the Service Broker Manager and calls the appropriate product APIs to perform the work. The results are returned via the Service Broker Manager to the service requester and then back to the user application.
- Service requester
A service requester is the interface to the user application. The user application calls the service requester APIs to request the product to perform some work. The service requester formats the user data and issues a request to the Service Broker Manager function which forwards the request to the appropriate service function.

However, there are differences between the Service Broker Manager for Windows and the Service Broker Manager on OS/2:

1. The Service Broker Manager for Windows does not supply a general interface with which you can load multiple service brokers and services. When you start Service Broker Manager for Windows, one service broker and one service are loaded. So, if you want to work with more than one base application at the same time, you must start several instances of the Service Broker Manager for Windows.
2. The Service Broker Manager for Windows does not support threads. No function that relates to thread administration is available.
3. The data area exchanged between the service requester and the Service Broker Manager for Windows is limited to 64 KB.

Starting the Service Broker Manager for Windows

You can start the Service Broker Manager for Windows from:

- The FlowMark Runtime folder

To start the Service Broker Manager, click on the Service Broker Manager program item in your FlowMark folder. This starts the program EXMW3USB.EXE. The EXE file is stored in the BIN directory of FlowMark. If the program cannot be loaded, check if the BIN directory is part of your PATH.

When the EXE is started, the Service Broker Manager window with the menu items **Action** and **Log** and the Service Broker Manager:Log window are displayed. To start or stop a service broker or to exit the Service Broker Manager, select **Action**. With **Start Broker**, specify the names of the service broker and service DLLs without the extension .DLL. Note that these DLLs must reside in a directory that is part of your PATH. The window titles change to the name of the broker that you started.

Figure 19 shows the windows for an active service broker.

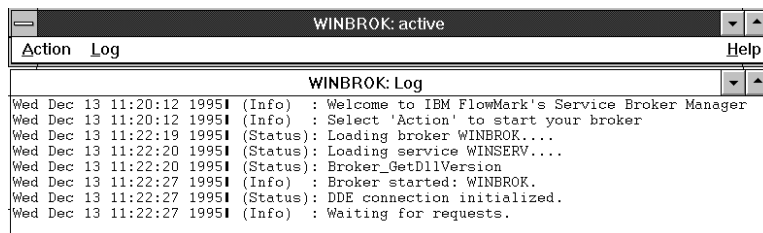


Figure 19. Active service broker

With **Log**, you can select which type of messages are to be recorded:

No Logging

No messages are recorded.

Error messages

Only error messages are recorded.

Status messages

Error and status messages are recorded.

All messages

Error, status, and information messages are recorded.

Log messages are written to your log window and to a log file. The log file is named *broker_name*.LOG, where *broker_name* is the name of the service broker that you started. The log file resides in the LOG directory of the Service Broker Manager.

- The command line

To start the Service Broker Manager from the command line, use the EXMW3USB.EXE for Windows 3.1. The syntax is:

```
▶▶——exmw3usb.exe——broker——service——▶▶
```

The service broker and the service that you specify with the command are started automatically.

If an error occurs during the startup (for example, if the logon to the base product is unsuccessful), the service broker and the service are unloaded.

When you stop the Service Broker Manager, the service broker and the service are unloaded.

Implementing a service broker on Windows

Each service broker consists of a DLL that exports the following functions (optional functions are marked with *opt*):

- Broker_GetDllVersion
- Broker_GetVersion
- Broker_Init *opt*
- Broker_Exit *opt*
- Broker_Logon *opt*
- Broker_Logoff *opt*
- LibMain *opt* for Windows 3.1

The Service Broker Manager calls these functions to:

- Determine the version of the service broker
- Load and initialize the service broker
- Perform logon and logoff according to the managed application
- Clean up and unload the service broker

If your service broker does not contain the mandatory functions, the Service Broker Manager stops processing and unloads the service broker.

Necessary definitions are included in the header file EXMW3CAL.H. Additionally, the EXMW3CAL.H file includes EXMP3FTP.H, which contains the type definitions for this API. Include the broker header file first when you implement a service broker.

So, your service broker program starts with the following statements:

```
//  
// Sample broker dll  
//  
#include <windows.h>           // Windows header  
  
#include <exmw3cal.h>         // broker header file
```

Broker_GetDllVersion function

This mandatory function has the following format:

```
ULONG APIENTRY Broker_GetDllVersion(VOID)
```

This function must return the value of predefined constant SB_BROKER_DLLVERSION.

Example

```
ULONG APIENTRY Broker_GetDllVersion (VOID)  
{  
    return SB_BROKER_DLLVERSION;  
}
```

Broker_GetVersion

This mandatory function has the following format:

```
VOID APIENTRY Broker_GetVersion(SbBrokerVersion FAR * pVer)
```

This function must return the name and version of the broker DLL by filling the structure SbBrokerVersion:

Example

```
VOID APIENTRY Broker_GetVersion (PSbBrokerVersion pVer)  
{  
  
    pVer->pName   = "WINBROK"; // Name of broker DLL  
    pVer->Version = 123;  
  
}
```

Broker_Init

This optional function is called after your service broker has been loaded. It has the following format:

```
APIRET APIENTRY Broker_Init(SbBrokerInit FAR * pInit)
```

A structure containing initialization information is passed to the function:

```
typedef struct
{
    ULONG    Size;           /* I: actual size of structure */
    ULONG    Handle;        /* NOT USED ON WINDOWS */
    HMODULE  BrokerDLL;     /* NOT USED ON WINDOWS */
    PSZ      pBrokerName;   /* I: Name of broker dll */
    PVOID    pInstance;     /* 0: user-defined instance pointer */
} SbBrokerInit;
```

You can use a user-defined instance pointer, `pInstance`, to store any instance-related information. This pointer is passed to all subsequent function calls, but can only be set in the `Broker_Init` function.

Example

```
APIRET APIENTRY Broker_Init (PSbBrokerInit pInit)
{
    SbbLog(0, SB_BROK_LOG_LEVEL_3, "Broker_Init has been called");

    //
    // Initialization starts here
    //

    return 0;
}
```

If the return code of the `Broker_Init` function is not zero, the Service Broker Manager stops processing and unloads the service broker and the service.

Broker_Exit

This optional function is called when the Service Broker Manager is stopped. This is the last function that the Service Broker Manager calls before exiting, so do all cleanup here.

The function has the following format:

```
APIRET APIENTRY Broker_Exit(SbBrokerExit FAR * pExit)
```

A read-only structure containing exit information is passed to the function:

```
typedef struct
{
    ULONG    Size;           /* I: actual size of structure */
    ULONG    Handle;        /* NOT USED ON WINDOWS */
    PVOID    pInstance;    /* I: user-defined instance pointer */
} SbBrokerExit;
```

Example

```
APIRET APIENTRY Broker_Exit (PSbBrokerExit pExit)
{
    SbbLog(0, SB_BROK_LOG_LEVEL_3, "Broker_Exit has been called");

    // Cleanup processing:
    // Close file now
    //
    _lclose(hFile);

    return 0;
}
```

If the return code of the `Broker_Exit` function is not zero, an error message containing the return code is written to the log file. The service broker is unloaded.

Broker_Logon

This function performs a logon to the application that is managed by the service broker. It has the following format:

```
APIRET APIENTRY Broker_Logon (SbBrokerLogon FAR * pLogon)
```

The SbBrokerLogon structure that is passed to the function has the following format:

```
typedef struct
{
    ULONG    Size;           /* I: actual size of structure */
    ULONG    Handle;        /* NOT USED ON WINDOWS */
    HAB      Hab;           /* NOT USED ON WINDOWS */
    HMQ      Hmq;           /* NOT USED ON WINDOWS */
    PVOID    pInstance;     /* I: user-defined instance pointer */
    PVOID    pSession;      /* I/O: user-defined session data */
    ULONG    SessionSize;   /* I/O: data size */
} SbBrokerLogon;
```

pSession and SessionSize refer to a user-defined logon or session buffer that contains a handle of the connection to the managed application, or similar logon information. The buffer must be allocated within the Broker_Logon function and can be freed in the Broker_Logoff function. Services also have access to pSession and pSessionSize.

If the return code of the Broker_Logon function is not zero, the Service Broker Manager stops processing and unloads the service broker and services.

Broker_Logoff

This optional function performs a logoff from the application that is managed by the service broker. It is called when you stop the Service Broker Manager. The format of this function is:

```
APIRET APIENTRY Broker_Logoff(SbBrokerLogoff FAR * pLogoff)
```

A SbBrokerLogoff structure is passed to the function. The fields of the SbBrokerLogoff structure contain the information from the SbBrokerLogon structure from the call to Broker_Logon. If you allocated a session buffer in Broker_Logon, free this buffer in the Broker_Logoff function. The SbBrokerLogoff structure is defined as follows:

```
typedef struct
{
    ULONG    Size;           /* I: actual size of structure */
    ULONG    Handle;        /* NOT USED ON WINDOWS */
    HAB      Hab;           /* NOT USED ON WINDOWS */
    HMQ      Hmq;           /* NOT USED ON WINDOWS */
    PVOID     pInstance;    /* I: user-defined instance pointer */
    PVOID     pSession;     /* I/O: user-defined session data */
    ULONG     SessionSize;  /* I/O: data size */
} SbBrokerLogoff;
```

Example

```
APIRET APIENTRY Broker_Logoff (PSbBrokerLogoff pLogoff)
{
    //
    // Logoff ...
    //

    free (pLogoff->pSession);

    return 0;
}
```

If the return code of the Broker_Logoff function is not zero, an error message containing the return code is written to the log file.

LibMain (for Windows 3.1)

This function is called by the Windows operating system when a DLL is loaded. You can use this function to retrieve the module instance handle.

Example

```
HINSTANCE hInst;                // Global dll instance handle

int FAR PASCAL LibMain (HINSTANCE hInstance, WORD wDataSeg, WORD wHeapSize,
                        LPSTR lpszCmdLine)
{
    if (wHeapSize > 0)
        UnlockData (0);

    hInst = hInstance;          // Set the instance handle

    return 1;
}
```

Building the service broker DLL

To build the service broker DLL, you need a makefile and a module definition file. The module definition file exports the broker functions, so that the Service Broker Manager can load the functions:

Windows 3.1

```
LIBRARY winbrok
EXETYPE WINDOWS
DESCRIPTION 'Broker DLL Windows'
CODE PRELOAD MOVEABLE DISCARDABLE
DATA PRELOAD MOVEABLE SINGLE

HEAPSIZE 8192

EXPORTS
  Broker_GetDllVersion @1
  Broker_GetVersion    @2
  Broker_Init          @3
  Broker_Exit          @4
  Broker_Logon         @5
  Broker_Logoff        @6
```

A makefile to build the DLL looks similar to the following:

```
Makefile for Windows 3.1
# ----
# Makefile for Broker DLL on Windows
# Borland Compiler
# ---

SBRROOT = m:\exmw
BORLANDC = f:\bc45
INCLUDES = -I$(BORLANDC)\INCLUDE -I$(SBRROOT)\INCLUDE

CC = bcc
CFLAGS = -c -v -ml -w-par -P -W -2 $(INCLUDES)
LFLAGS = /c /v /n /Tw /L$(BORLANDC)\LIB;$(SBRROOT)\LIB c0d1
LIBS = import mathws cw1 exmw3cal
ADDFLAGS = /DWIN31

winbrok.dll: winbrok.obj winbrok.def
    tlink $(LFLAGS) winbrok.obj, winbrok.d11, NUL, $(LIBS), winbrok
    rc winbrok.d11
    implib winbrok.lib winbrok.def

winbrok.obj: winbrok.c
    $(CC) $(CFLAGS) $(ADDFLAGS) winbrok.c
```

The libraries (IMPORT.LIB, MATHWS.LIB, and CWL.LIB) are part of the compiler (in this case, the Borland C++ compiler). The library EXMW3CAL.LIB contains the C language service broker API. Currently, only function SbbLog() is supported, which writes messages to the log file and to your main Service Broker Manager window.

Implementing a service on Windows

Each service consists of a DLL that exports the following standard functions, plus any user-defined service functions (optional functions are marked with *opt*):

- Service_GetDllVersion
- Service_CheckBroker
- Service_Init *opt*
- Service_Exit *opt*
- Service_Start *opt*
- Service_Stop *opt*
- LibMain *opt* for Windows 3.1

The Service Broker Manager calls these functions to:

- Determine the version of the service
- Check if the service is compatible with a service broker
- Load and initialize the service
- Start and stop the service
- Clean up and unload the service

A service can additionally export an arbitrary number of service functions performing user-defined operations using the current logon or session provided by the corresponding service broker.

For performance reasons, you can place a single service in a service broker DLL. But this is recommended only for general services that are needed whenever the service broker is used.

Multiple service functions can be executed in parallel, so they should be reentrant.

If your service does not contain the mandatory functions, the Service Broker Manager stops processing and unloads the service broker and the service.

Necessary definitions are included in the header file EXMW3LSR.H. Additionally, EXMW3LSR.H includes EXMP3FTP.H that contains the type definitions used in this API. Include the service header file first when you implement a service. So, your service program starts with the following statements:

```
#include <windows.h>           // Windows header
#include <exmw3lsr.h>         // Service header
```

Service_GetDllVersion

This mandatory function must return the value of the predefined constant SB_SERVICE_DLLVERSION. It has the following format:

```
ULONG APIENTRY Service_GetDllVersion (void)
```

If the returned version does not match the current version of the Service Broker Manager, the Service Broker Manager stops processing. In this case, you must update your Service Broker Manager for Windows.

Example

```
ULONG APIENTRY Service_GetDllVersion(void)
{
    return SB_SERVICE_DLLVERSION;
}
```

Service_CheckBroker

This mandatory function checks if the service is compatible with the service broker. It has the following format:

```
APIRET APIENTRY Service_CheckBroker(SbBrokerVersion FAR * pBroker)
```

A read-only `SbBrokerVersion` structure containing the name and version of the service broker is passed to the function. The function returns 0 if the name and version are correct. Otherwise the service broker and the service are unloaded and the Service Broker Manager stops processing.

Example

```
APIRET APIENTRY Service_CheckBroker(PSbBrokerVersion pVer)
{
    APIRET apiRc = 0;

    //
    // Check the name of the broker:
    //
    if (strcmp(pVer->pName, "WINBROK") != 0)
    {
        apiRc = 1;                // Wrong broker name
    }

    //
    // Check the version:
    //
    if (pVer->Version < 123)
    {
        apiRc = 1;                // Wrong version
    }

    return apiRc;
}
```

Service_Init

This optional function is called after the service has been loaded. It contains all initialization information for the service. The format of this function is:

```
APIRET APIENTRY Service_Init(SbServiceInit FAR * pInit)
```

A structure containing initialization information is passed to the function:

```
typedef struct {  
    ULONG    Size;           /* I: actual size of structure */  
    ULONG    Handle;        /* NOT USED ON WINDOWS */  
    HMODULE  ServiceDLL;    /* NOT USED ON WINDOWS */  
    PSZ     pServiceName;   /* I: Name of service instance */  
    HMODULE  BrokerDLL;     /* NOT USED ON WINDOWS */  
    PSZ     pBrokerName;    /* I: Name of broker instance */  
    PVOID   pInstance;     /* O: user-defined instance pointer */  
} SbServiceInit;
```

You can use a user-defined instance pointer, `pInstance`, to store any instance-related information. This pointer is passed to any subsequent service function calls, but can be set in the `Service_Init` function only. This function returns zero if it was successful. Otherwise the service broker and the service are unloaded and the Service Broker Manager stops.

Example

```
APIRET APIENTRY Service_Init (PSbServiceInit pInit)  
{  
    // Init process ...  
    return 0;  
}
```


Service_Exit

This optional function is called when the service is unloaded. The format of this function is:

```
APIRET APIENTRY Service_Exit(SbServiceExit FAR * pExit)
```

A read-only structure containing exit information is passed to the function:

```
typedef struct {  
    ULONG    Size;           /* I: actual size of structure */  
    ULONG    Handle;        /* NOT USED ON WINDOWS */  
    PVOID    pInstance;     /* I: user-defined instance pointer */  
} SbServiceExit;
```

Example

```
APIRET APIENTRY Service_Exit (PSbServiceExit pExit)  
{  
    // Cleanup processing ...  
    return 0;  
}
```

If the return code of the `Service_Exit` function is not zero, an error message that contains this return code is written to the log file and the service is unloaded.

Service_Start

This optional function is called when the service is started. It has the following format:

```
APIRET APIENTRY Service_Start(SbServiceStart FAR * pStart)
```

The function returns 0 if it was successful. Otherwise the service broker and the service are unloaded.

The SbServiceStart structure is defined as follows:

```
typedef struct
{
    ULONG    Size;           /* I: actual size of structure */
    ULONG    Handle;        /* NOT USED ON WINDOWS */
    PVOID    pInstance;     /* I: user-defined instance pointer */
    PVOID    pSession;      /* I: user-defined session buffer */
    ULONG    SessionSize;   /* I: user-defined size of session buffer */
} SbServiceStart;
```

pSession and SessionSize refer to the service broker's session data.

Example

```
APIRET APIENTRY Service_Start (PSbServiceStart pStart)
{
    SbbLog(0, SB_BROK_LOG_LEVEL_3, "... Service_Start ...");

    // Start processing ...

    return 0;
}
```

Service_Stop

This optional function is called when the service is stopped. It has the following format:

```
APIRET APIENTRY Service_Stop(SbServiceStop FAR * pStop)
```

This function returns 0 if it was successful. If the return code of the Service_Stop function is not zero, an error message that contains the return code is written to the log and the service is unloaded.

The SbServiceStop structure is defined as follows:

```
typedef struct
{
    ULONG    Size;           /* I: actual size of structure */
    ULONG    Handle;        /* NOT USED ON WINDOWS */
    PVOID    pInstance;     /* I: user-defined instance pointer */
    PVOID    pSession;     /* I: user-defined session buffer */
    ULONG    SessionSize;   /* I: user-defined size of session buffer */
} SbServiceStop;
```

Example

```
APIRET WINAPI Service_Stop (PSbServiceStop pStop)
{
    // Stop processing

    return 0;
}
```

LibMain (for Windows 3.1)

This function is called by the Windows operating system when a DLL is loaded. You can use this function to retrieve the module instance handle.

Example

```
HINSTANCE hInst;           // Global dll instance handle

int FAR PASCAL LibMain (HINSTANCE hInstance, WORD wDataSeg, WORD wHeapSize,
                        LPSTR lpszCmdLine)
{
    if (wHeapSize > 0)
        UnlockData (0);

    hInst = hInstance;     // Set the instance handle

    return 1;
}
```

Implementing a service function

The syntax of a service function is as follows:

```
LONG APIENTRY ServiceFunction(SbFuncInfo FAR * pInfo,  
                              PVOID          pData,  
                              UINT FAR *     pDataSize,  
                              UINT           MaxOutDataSize)
```

A service DLL can provide several service functions. The parameters passed to the function are:

Parameter	I/O	Description
pInfo	I	A read-only structure containing information about the service function's environment (for example, the field pSession points to the session buffer allocated by Broker_Logon).
pData	I/O	Points to a data buffer; as input, it contains input data of size *pDataSize. The service function can copy MaxOutDataSize bytes of its output data to pData.
pDataSize	I/O	As input, it contains the size of input data; as output, it must contain the size of written output data or 0.
MaxOutDataSize	I	Maximum size of output data.

The SbFuncInfo structure is defined as follows:

```
typedef struct
{
    ULONG    Size;          /* I: actual size of structure          */
    ULONG    Handle;       /* NOT USED ON WINDOWS                 */
    HAB      Hab;         /* NOT USED ON WINDOWS                 */
    HMQ      Hmq;         /* NOT USED ON WINDOWS                 */
    PVOID    pInstance;   /* I: user-defined instance pointer    */
    PVOID    pSession;    /* I: user-defined session buffer      */
    ULONG    SessionSize; /* I: user-defined size of session buffer */
} SbFuncInfo;
```

A service function can return a user-defined return code that must be greater than or equal to 0. Return codes less than 0 should not be used, because they are reserved for Service Broker Manager internal purposes, except the following pre-defined return codes:

Standard Error Code	Description
SB_FUNC_RC_MEMORY (-1)	Not enough memory to complete the operation
SB_FUNC_RC_OVERFLOW (-2)	Buffer is too small to write output data
SB_FUNC_RC_INVALID_DATA (-3)	Invalid input data

Output data is returned to the caller only if the return code is greater than or equal to 0.

The following is a service function example that just copies a string into the output area:

Example

```
#ifdef __cplusplus
extern "C" {
#endif

//
// Function prototype:
//
LONG APIENTRY DoSomething(PSbFuncInfo  pInfo,
                          PVOID        pData,
                          UINT FAR *   pulDataSize,
                          UINT         MaxOutDataSize);

#ifdef __cplusplus
}
#endif

//
// ... and the function itself:
//
#define STRING_RESULT "This is a string result"

LONG APIENTRY DoSomething (PSbFuncInfo pInfo,
                          PVOID        pData,
                          UINT FAR *   pDataSize,
                          UINT         MaxOutDataSize)
{
```

Example (continued)

```
SbbLog(0, SB_BROK_LOG_LEVEL_3, " ... DoSomething ... ");

//
// What is our output size that we want to send ?
//
UINT OutputSize = strlen(StringResult) + 1;

//
// Invalid output data
//
if (*pDataSize == 0)
    return SB_FUNC_RC_INVALID_DATA;

//
// Buffer too small ?
//
if (OutputSize > MaxOutDataSize)
    return SB_FUNC_RC_OVERFLOW;

//
// Copy the data into the buffer
// and set the size field
memcpy(pData, StringResult, OutputSize);
*pDataSize = OutputSize;

//
// That's it. No problem.
//
return 0;
}
```

Building the service DLL

To build the service DLL, you need a makefile and a module definition file. The module definition file exports the service functions, so that the Service Broker Manager can load the functions:

Module definition file for Windows 3.1

```
LIBRARY winserv
EXETYPE WINDOWS
DESCRIPTION 'Broker Service DLL Windows'
CODE PRELOAD MOVEABLE DISCARDABLE
DATA PRELOAD MOVEABLE SINGLE

HEAPSIZE 8192

EXPORTS

    Service_GetDllVersion @1
    Service_CheckBroker   @2
    Service_Init          @3
    Service_Exit          @4
    Service_Start         @5
    Service_Stop          @6
    DoSomething           @7
```


A makefile to build the DLL looks similar to the following:

```
Makefile for Windows 3.1
# ----
# Makefile for Broker DLL on Windows
# Borland Compiler
# ---

SBRROOT = m:\exmw
BORLANDC = f:\bc45
INCLUDES = -I$(BORLANDC)\INCLUDE -I$(SBRROOT)\INCLUDE

CC = bcc
CFLAGS = -c -v -ml -w-par -P -W -2 $(INCLUDES)
LFLAGS = /c /v /n /Tw /L$(BORLANDC)\LIB;$(SBRROOT)\LIB c0d1
LIBS = import mathws cw1 exmw3cal
ADDFLAGS = /DWIN31

winserv.dll: winserv.obj winserv.def
    tlink $(LFLAGS) winserv.obj, winserv.dll, NUL, $(LIBS), winserv
    brc winserv.dll
    implib winserv.lib winserv.def

winserv.obj: winserv.c
    $(CC) $(CFLAGS) $(ADDFLAGS) winserv.c
```

The libraries (IMPORT.LIB, MATHWS.LIB, and CWL.LIB) are part of the compiler (in this case, the Borland C++ compiler). The library EXMW3CAL.LIB contains the C language service broker API.

Implementing a service requester

Service functions can be executed using the service requester API. Necessary definitions are in the header file EXMP3FRQ.H. This include file contains all prototypes and the return codes of the SbrCallService function. The name of the service requester API library is EXMW3KRQ.LIB for Windows 3.1.

Calling a service function

Use the SbrCallService function to call a service function. The format of SbrCallService is as follows:

```

APIRET APIENTRY SbrCallService (PSZ      pBroker,
                                PSZ      pService,
                                PSZ      pFunction,
                                PVOID    pInData,
                                UINT     InDataSize,
                                PVOID    pOutData,
                                UINT FAR * pOutDataSize,
                                LONG FAR * pRC,
                                ULONG    TimeOut);

```

The parameters are:

Parameter	I/O	Description
pBroker	I	Name of service broker (not case-sensitive, limited to 8 characters).
pService	I	Name of service (not case-sensitive, limited to 8 characters).
pFunction	I	Name of service function (case-sensitive).
pInData	I	Address of a buffer that contains input data or NULL.
InDataSize	I	Size of input data or 0.
pOutData	O	Address of a buffer for output data.
pOutDataSize	I/O	As input, it contains the size of the output buffer. As output, it contains the actual size of output data written to pOutData or 0.
pRC	O	User-defined return code of service function.
TimeOut	I	Currently not used.

Sample service requester for Windows 3.1

The following is an example for coding a service requester. The sample service requester WINREQ.C for Windows 3.1 is provided with the product. It is located in the SBM\SAMPLES\TOOLKIT subdirectory of your FlowMark directory. This directory also contains a sample makefile and a sample module definition file for a service requester.

Example

```
//
// Includes:
//
#include <windows.h>
#include <string.h>

#include <exmp3frq.h>           // necessary !
#include "winreq.h"

VOID CallServiceBroker()
{
    VOID FAR * pInAreaPtr;
    VOID FAR * pOutAreaPtr;

    // Set inarea and outarea stuff
    pInAreaPtr = (VOID FAR *) &szInarea[0];
    pOutAreaPtr = (VOID FAR *) &szOutarea[0];

    dInareaLen = lstrlen(szInarea);
    dOutareaLen = AREALEN;

    lOutRc = 0;

    //
    // Call ServiceBrokerManager via requester interface
    //
    apiRc = SbrCallService(szBroker,
                           szService,
                           szFunction,
                           pInAreaPtr,
                           dInareaLen,
                           pOutAreaPtr,
                           &dOutareaLen,
                           &lOutRc,
                           0);

    if (apiRc != 0)
    {
        MessageBox(NULL, "SBM error detected. See <request.log> for more info",
                   szBroker, MB_ICONEXCLAMATION|MB_OK);
    }
    return;
}
```

Using the standard service requester on Windows 3.1

Use the program EXMW3FRC.EXE, which is located in the EXMWIN\BIN directory, to pass textual parameters to the Service Broker Manager for Windows and to the services. To run the program, do one of the following:

- Set the Program Icon Properties under Windows 3.1 as follows:

Description	FlowMark Requester Command Line
Command Line	<i>x:EXMWIN\BIN\EXMW3FRC.EXE parameters</i>
Working Directory	<i>x:EXMWIN\BIN</i>

Where *x* is the drive where this directory is located. Then, double-click on the program icon.

- On the Windows 3.1 Run command line, enter:

exmw3frc *parameters*

Where *parameters* are:

Parameter	Description
<i>/q</i>	Quiet; output data and result code, or error message is not displayed (optional).
<i>broker</i>	Logical name of the service broker that provides the specific service (not case-sensitive).
<i>service</i>	Logical name of the service providing the requested function (not case-sensitive).
<i>function</i>	Name of the service function (case-sensitive). This function must be exported by the service DLL.
<i>params</i>	Textual parameters that must be passed to the service function (optional). Refer to the documentation of the used service brokers and services for further details.

Using the FlowMark requester

You can use the program EXMW3FRQ.EXE, which is located in the EXMWIN\BIN directory, to prepare and test the FlowMark requester functions. The program opens a dialog that provides entry fields for all required parameters. You can set default values by using the push buttons labeled with the names of the FlowMark service functions.

To run the program, do the following:

1. Set the Program Icon Properties under Windows 3.1 as follows:

Description	FlowMark Requester Test
Command Line	<i>x:EXMWIN\BIN\EXMW3FRQ.EXE</i>
Working Directory	<i>x:EXMWIN\BIN</i>

Where *x* is the drive where this directory is located.

2. Double-click on the program icon.

Chapter 8. Building block for MQSeries support

Note: The programs in this building block are designed for general purpose use and are available for OS/2 and AIX.

You can use this building block with FlowMark to start, suspend, resume, and terminate a FlowMark process on another FlowMark system by using MQSeries. The started process can also return data, such as a subprocess. These functions are currently supported between FlowMark and FlowMark for MVS/ESA Application Integration Feature (AIF) systems.

MQSeries enables communication between applications running on the same platform or on different platforms. MQSeries uses queues as its communications vehicle. The applications use the queues to send and receive messages.

Restrictions

- The parent process name, the FlowMark database, and the FlowMark server cannot be determined by FlowMark API calls. The database and server names are extracted from environment variables, if available, and the parent-process name is replaced by FMMQI_UNKNOWN.
- The codepage support is currently restricted to one of the following:
 - Any codepage if the same codepage is used on all FlowMark systems
 - Codepage 037 (US-EBCDIC) and 001 (US-ASCII) when different codepages are used on the FlowMark systems.

MQSeries definitions

A local queue manager must already be installed.

With MQSeries, you can use various communications protocols. Therefore, the standard and tuning settings can vary from installation to installation. The following descriptions show sample minimum definitions in a TCP/IP environment.

For more complex definitions, refer to the MQSeries documentation, especially to the *MQSeries Distributed Queuing Guide* and the *MQSeries Command Reference*.

Customizing the MQSeries definitions

Edit the sample file EXMP2ABB.MQI. Modify the definitions according to your network installation parameters. In the following definitions, it is assumed that the queue manager name is the same as the TCP/IP host name.

For a connection between *two FlowMark systems on OS/2*, change at least the names of the channels and the transmission queue and the attributes of the remote queue on each system:

- Channels
 - FMMQI_LOCAL_OS2NAME
 - FMMQI_OS2NAME_LOCAL

Replace the string LOCAL with the name of the respective local queue manager.

- Transmission queue
 - OS2NAME
- Remote queue
 - FMMQI_START_OS2

Note: The easiest way to do this in a TCP/IP environment is to set *OS2NAME* in all these definitions to the queue manager name of the target OS/2 machine.

For a connection between *two FlowMark systems on AIX*, change at least the names of the channels and the transmission queue and the attributes of the remote queue on each system:

- Channels
 - FMMQI_LOCAL_AIXNAME
 - FMMQI_AIXNAME_LOCAL

Replace the string LOCAL with the name of the respective local queue manager.

- Transmission queue
 - AIXNAME
- Remote queue
 - FMMQI_START_AIX

Note: The easiest way to do this in a TCP/IP environment is to set *AIXNAME* in all these definitions to the queue manager name of the target AIX machine.

For a connection between a *FlowMark system on OS/2* and a *FlowMark system on AIX*, change at least the names of the channels and the transmission queue and the attributes of the remote queue on each system:

- Channels
 - On AIX:
 - FMMQI_LOCAL_OS2NAME
 - FMMQI_OS2NAME_LOCAL
 - On OS/2:
 - FMMQI_LOCAL_AIXNAME
 - FMMQI_AIXNAME_LOCAL

Replace the string LOCAL with the name of the respective local queue manager.

- Transmission queue
 - On AIX: OS2NAME
 - On OS/2: AIXNAME
- Remote queue
 - On AIX: FMMQI_START_OS2
 - On OS/2: FMMQI_START_AIX

Note: The easiest way to do this in a TCP/IP environment is to set *AIXNAME* in these definitions to the queue manager name of the AIX machine and *OS2NAME* to the queue manager name of the OS/2 machine.

For a connection between a *FlowMark system on OS/2 or AIX* and a *FlowMark for MVS/ESA system*, change at least the names of the channels and the transmission queue and the attributes of the remote queue on each system:

- Channels
 - On AIX or OS/2:
 - FMMQI_LOCAL_MVSNAME
 - FMMQI_MVSNAME_LOCAL
 - On MVS for a connection to AIX:
 - FMMQI_LOCAL_AIXNAME
 - FMMQI_AIXNAME_LOCAL
 - On MVS for a connection to OS/2:
 - FMMQI_LOCAL_OS2NAME
 - FMMQI_OS2NAME_LOCAL

Replace the string LOCAL with the name of the respective local queue manager.

- Transmission queue
 - On AIX or OS/2: *MVSNAME*
 - On MVS for a connection to AIX: *AIXNAME*
 - On MVS for a connection to OS/2: *OS2NAME*
- Remote queue
 - On AIX or OS/2: *FMMQI_START_MVS*
 - On MVS for a connection to AIX: *FMMQI_START_AIX*
 - On MVS for a connection to OS/2: *FMMQI_START_OS2*

Note: The easiest way to do this in a TCP/IP environment is to set *MVSNAME* in these definitions to the queue manager name of the MVS machine, *AIXNAME* to the queue manager name of the AIX machine and *OS2NAME* to the queue manager name of the OS/2 machine.

Make sure that the local queue manager is active and apply the updated definitions by entering the command:

```
runmqsc < exmp2abb.mqi > exmp2abb.dat
```

Check the contents of EXMP2ABB.DAT.

For communication between different queue managers, you must make some network definitions that enable the queue managers to communicate via channels. These definitions are different for each communications protocol and each platform. For details about setting up MQSeries communication on OS/2 and AIX systems, refer to the *MQSeries Distributed Queuing Guide*.

Example for a connection between three systems A, B, and C

System A: OS/2, queue manager name = HOSTNAME = ALPHA

- Channels


```
FMMQI_ALPHA_BETA
  CHLTYPE(SDR) CONNAME('BETA') XMITQ('BETA')
FMMQI_ALPHA_GAMMA
  CHLTYPE(SDR) CONNAME('GAMMA') XMITQ('GAMMA')
FMMQI_BETA_ALPHA
  CHLTYPE(RCVR)
FMMQI_GAMMA_ALPHA
  CHLTYPE(RCVR)
```
- Transmission queues


```
BETA
GAMMA
```
- Remote queues


```
FMMQI_START_AIX
  RQMNAME('BETA')
FMMQI_START_MVS
  RQMNAME('GAMMA')
```


System B: AIX, queue manager name = HOSTNAME = BETA

- Channels

```
FMMQI_BETA_ALPHA
  CHLTYPE(SDR) CONNAME('ALPHA') XMITQ('ALPHA')
FMMQI_BETA_GAMMA
  CHLTYPE(SDR) CONNAME('GAMMA') XMITQ('GAMMA')
FMMQI_ALPHA_BETA
  CHLTYPE(RCVR)
FMMQI_GAMMA_BETA
  CHLTYPE(RCVR)
```
- Transmission queues

```
ALPHA
GAMMA
```
- Remote queues

```
FMMQI_START_OS2
  RQMNAME('ALPHA')
FMMQI_START_MVS
  RQMNAME('GAMMA')
```

System C: MVS, queue manager name = HOSTNAME = GAMMA

- Channels

```
FMMQI_GAMMA_ALPHA
  CHLTYPE(SDR) CONNAME('ALPHA') XMITQ('ALPHA')
FMMQI_GAMMA_BETA
  CHLTYPE(SDR) CONNAME('BETA') XMITQ('BETA')
FMMQI_ALPHA_GAMMA
  CHLTYPE(RCVR)
FMMQI_BETA_GAMMA
  CHLTYPE(RCVR)
```
- Transmission queues

```
ALPHA
BETA
```
- Remote queues

```
FMMQI_START_OS2
  RQMNAME('ALPHA')
FMMQI_START_AIX
  RQMNAME('BETA')
```

Setting up FlowMark

This section describes how to set up FlowMark for the use of the MQSeries building blocks.

Preparing the sample processes

Start the Buildtime client, import the FDL file EXMP2ABB.FDL, and translate all sample processes.

Using the sample processes

Open the sample processes as diagrams to look at the samples.

On both the local and the remote system:

1. Start the queue manager
2. Start the listener program (if necessary)
3. Start the channels by entering the following commands:

```
runmqsc  
START CHANNEL(FMMQI_LOCAL_qmremote)
```

Start the daemon programs EXMP2ASV.EXE and EXMP2ARM.EXE with the appropriate parameters from the command line.

Start the Runtime client and start the process of your choice:

- OS/2 Runtime client

FMMQI_OS2_LOCAL

Start a subprocess on the same OS/2 machine

FMMQI_OS2_OS2

Start a subprocess on a remote OS/2 machine

FMMQI_OS2_AIX

Start a subprocess on a remote AIX machine

FMMQI_OS2_MVS

Start a subprocess on a remote MVS machine

FMMQI_CONTROL_OS2_LOCAL

Control (start, suspend, resume, terminate, or restart) a subprocess on the same OS/2 machine

FMMQI_CONTROL_OS2_OS2

Control (start, suspend, resume, terminate, or restart) a subprocess on a remote OS/2 machine

FMMQI_CONTROL_OS2_AIX

Control (start, suspend, resume, terminate, or restart) a subprocess on a remote AIX machine

FMMQI_CONTROL_OS2_MVS

Control (start, suspend, resume, terminate, or restart) a subprocess on a remote MVS machine

- AIX Runtime client

FMMQI_AIX_LOCAL

Start a subprocess on the same AIX machine

FMMQI_AIX_AIX

Start a subprocess on a remote AIX machine

FMMQI_AIX_OS2

Start a subprocess on a remote OS/2 machine

FMMQI_AIX_MVS

Start a subprocess on a remote MVS machine

FMMQI_CONTROL_AIX_LOCAL

Control (start, suspend, resume, terminate, or restart) a subprocess on the same AIX machine

FMMQI_CONTROL_AIX_AIX

Control (start, suspend, resume, terminate, or restart) a subprocess on a remote AIX machine

FMMQI_CONTROL_AIX_OS2

Control (start, suspend, resume, terminate, or restart) a subprocess on a remote OS/2 machine

FMMQI_CONTROL_AIX_MVS

Control (start, suspend, resume, terminate, or restart) a subprocess on a remote MVS machine

Sample scenarios

This section describes several scenarios:

- “Starting FlowMark for MVS/ESA from FlowMark on OS/2 or AIX”
- “Controlling FlowMark for MVS/ESA from FlowMark on OS/2 or AIX”
- “Starting FlowMark on OS/2 or AIX from FlowMark for MVS/ESA”
- “Controlling FlowMark on OS/2 or AIX from FlowMark for MVS/ESA”
- “Starting FlowMark on OS/2 or AIX from FlowMark on OS/2 or AIX”
- “Controlling FlowMark on OS/2 or AIX from FlowMark on OS/2 or AIX”

Starting FlowMark for MVS/ESA from FlowMark on OS/2 or AIX

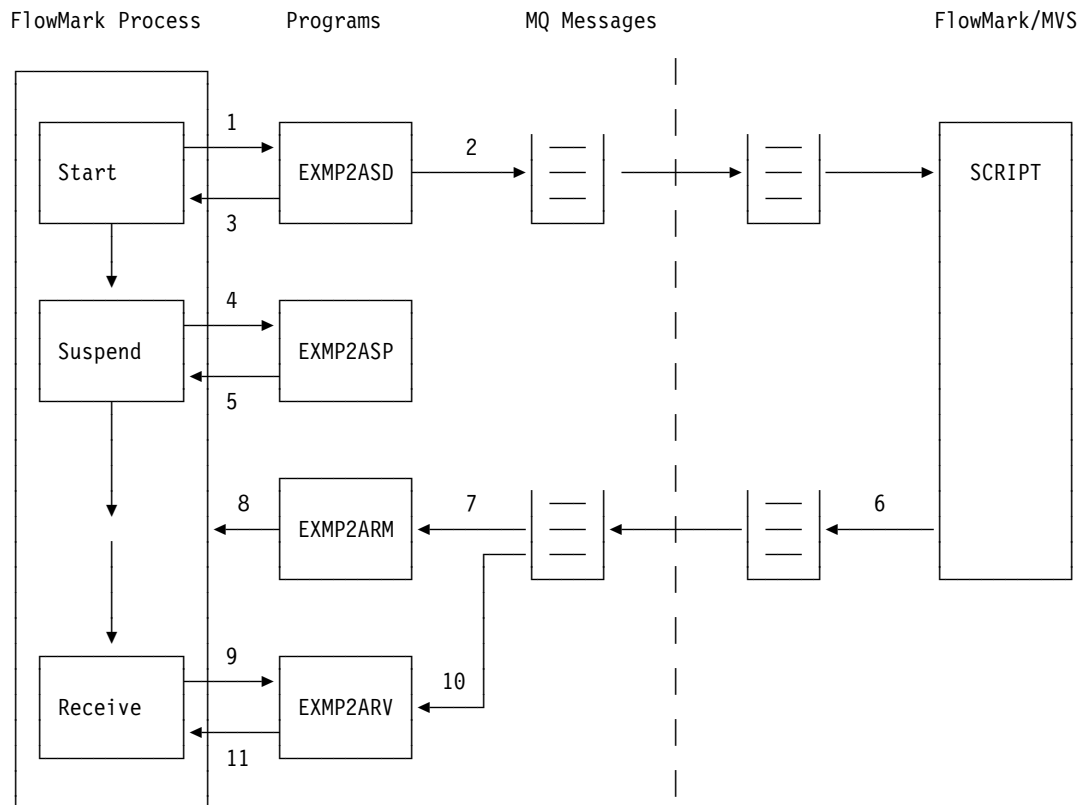


Figure 20. Process and data flow: Start MVS from LAN. Overview about the process and data flow when starting FlowMark for MVS/ESA scripts from a FlowMark on OS/2 or AIX process.

The FlowMark for MVS/ESA script is started by a FlowMark on OS/2 or AIX process consisting of the activities:

- Start
- Suspend
- Receive

The **Start** activity invokes **EXMP2ASD** (1) and passes data to it through command-line parameters and the FlowMark input container.

EXMP2ASD uses MQ calls to create an MQ message. In addition, it encodes the data (including all input container data and specific process-relevant data, for example, instance name of the starting process) within the application data and passes the message to the queue specified with the command-line data (2). The connected remote queue is serviced by FlowMark for MVS/ESA, so a script is started. This script can invoke other scripts to handle the actual request. It is recommended that this first script controls all logging activities.

EXMP2ASD sets specific members within the output container to return (3) process-relevant information (for example, FMMQI_CONTROL.InstanceName).

The **Suspend** activity invokes **EXMP2ASP** (4), which suspends the current process (5).

When the FlowMark for MVS/ESA script returns, it writes a message to a queue (6) that is continuously browsed (using blocking mechanisms) by the daemon program **EXMP2ARM** (7). The message contains the original message ID as correlation ID within the MQ header and additional data (including container data).

EXMP2ARM retrieves the instance name from the message and resumes this specific instance (8).

The **Receive** activity invokes **EXMP2ARV** (9) and, via command-line parameters, passes data (including the instance name of the remote process) to it.

EXMP2ARV uses native MQ calls to read the MQ message (10). It generates the correlation ID from the remote instance name returned in step (3) to get the appropriate message.

EXMP2ARV sets specific members within the output container to return process-relevant information and returns all container data from the message to the output container (11), if there are matching item names.

Controlling FlowMark for MVS/ESA from FlowMark on OS/2 or AIX

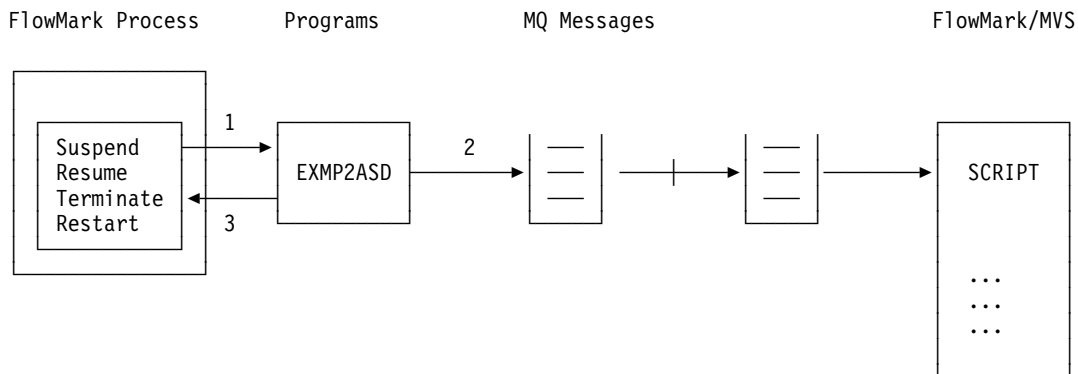


Figure 21. Process and data flow: Control MVS from LAN. Overview about the process and data flow when controlling FlowMark for MVS/ESA scripts from a FlowMark on OS/2 or AIX process.

Assumption: A FlowMark for MVS/ESA process has been started by a FlowMark on OS/2 or AIX process as described in “Starting FlowMark for MVS/ESA from FlowMark on OS/2 or AIX” on page 188 and is currently running.

The **Suspend, Resume, Terminate, or Restart** activity invokes **EXMP2ASD** (1) and, via command-line parameters, passes data (including remote instance name and requested action) to it.

EXMP2ASD uses MQ calls to create an MQ message. In addition, it encodes the data within the application data and passes the message to the queue specified with the command-line data (2). The connected remote queue is serviced by FlowMark for MVS/ESA, so a script is started. This script handles the requested action (suspend, resume, terminate, restart).

EXMP2ASD sets specific members within the output container to return (3) process-relevant information (for example, FMMQI_CONTROL.ReasonCode).

Starting FlowMark on OS/2 or AIX from FlowMark for MVS/ESA

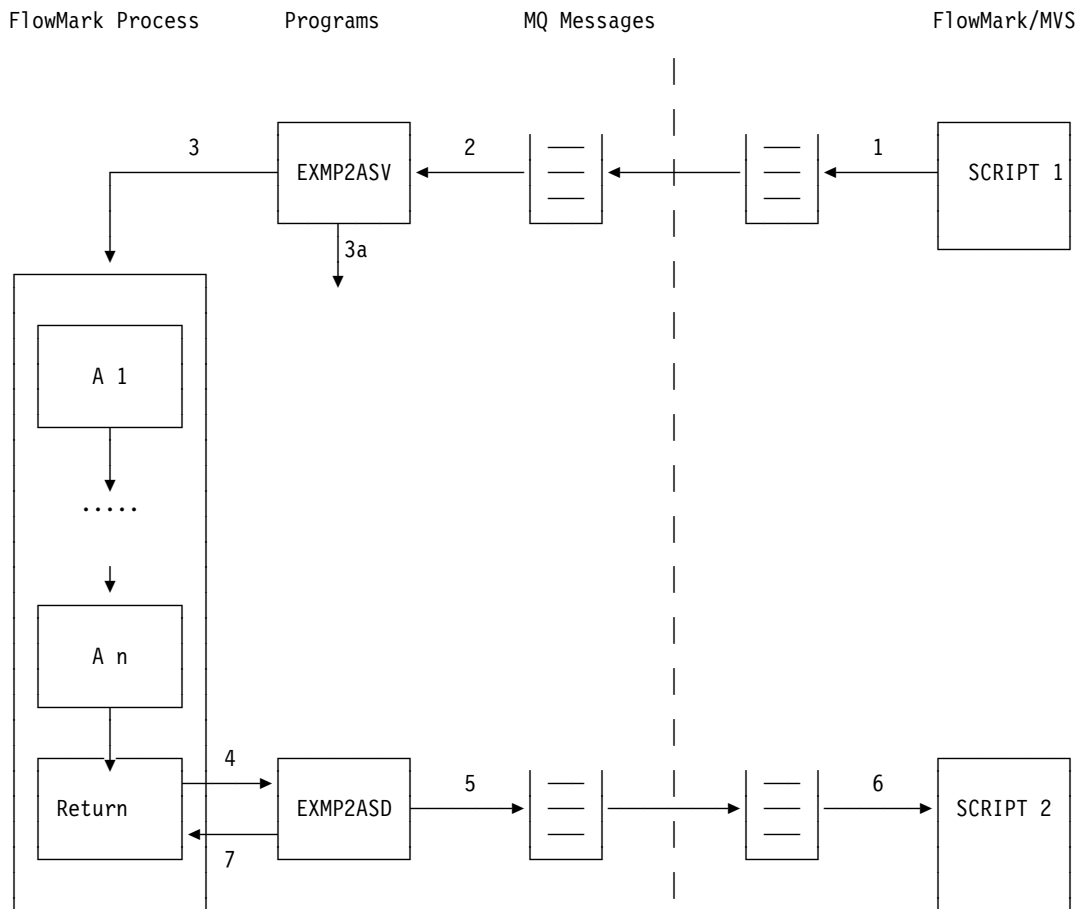


Figure 22. Process and data flow: Start LAN from MVS. Overview about the process and data flow when starting FlowMark on OS/2 or AIX processes from a FlowMark for MVS/ESA process.

SCRIPT 1 uses FlowMark for MVS/ESA API calls to put an MQ message (1) with encoded data (including all input container data and specific process-relevant data, for example, FMMQI_CONTROL.InstanceName) to the queue that is serviced by **EXMP2ASV**.

The connected remote queue is continuously read (using blocking mechanisms) by the daemon program **EXMP2ASV** (2). **EXMP2ASV** decodes the encoded data from the message and starts (3) the FlowMark on OS/2 or AIX process, which is specified within the data. The container data of the message is passed to the process as initial data and some additional data items are automatically provided (for example, FMMQI_CONTROL.InstanceName). These additional data items are described in “EXMP2ASV return data” on page 203.

EXMP2ASV invokes an exit function (3a) after the process start and passes all relevant data (such as success, instance names) to it. This can be used to provide logging and alert possibilities.

The FlowMark on OS/2 or AIX process can consist of several activities **A1** to **An**. The last activity is the **Return** activity, which invokes **EXMP2ASD** with a reply request (4).

EXMP2ASD uses the data passed via its command line and from the input container. It uses MQ calls to create an MQ message. In addition, it encodes the data (including all input container data and specific process-relevant data, for example, instance name of the starting process) within the application data and passes the message to the queue specified with the command-line data (5).

The connected remote queue is serviced by FlowMark for MVS/ESA (6), so the script **SCRIPT2** is started. This script can invoke other scripts to handle the actual return. It is recommended that this script controls all logging activities.

EXMP2ASD sets specific members within the output container to return (7) process-relevant information (for example, FMMQI_CONTROL.ReasonCode).

Controlling FlowMark on OS/2 or AIX from FlowMark for MVS/ESA

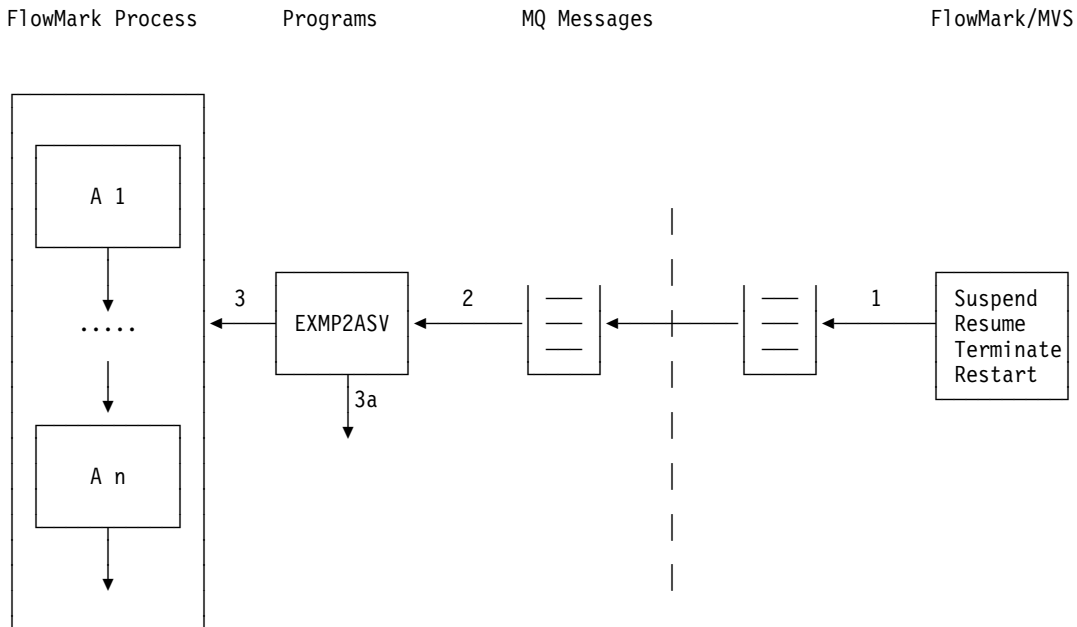


Figure 23. Process and data flow: Control LAN from MVS. Overview about the process and data flow when controlling FlowMark on OS/2 or AIX processes from a FlowMark for MVS/ESA process.

Assumption: A FlowMark on OS/2 or AIX process has been started by a FlowMark for MVS/ESA process as described in “Starting FlowMark on OS/2 or AIX from FlowMark for MVS/ESA” on page 191 and is currently running.

The **Suspend**, **Resume**, **Terminate**, or **Restart** script uses FlowMark for MVS/ESA API calls to put an MQ message (1) with encoded data (including remote instance name and requested action).

The connected remote queue is continuously read (using blocking mechanisms) by the daemon program **EXMP2ASV** (2). **EXMP2ASV** decodes the encoded data from the message and performs the requested action (3) on the specified FlowMark on OS/2 or AIX process instance.

EXMP2ASV invokes an exit function (3a) after performing the requested action and passes all relevant data (such as success, instance names) to it. This can be used to provide logging and alert possibilities.

Starting FlowMark on OS/2 or AIX from FlowMark on OS/2 or AIX

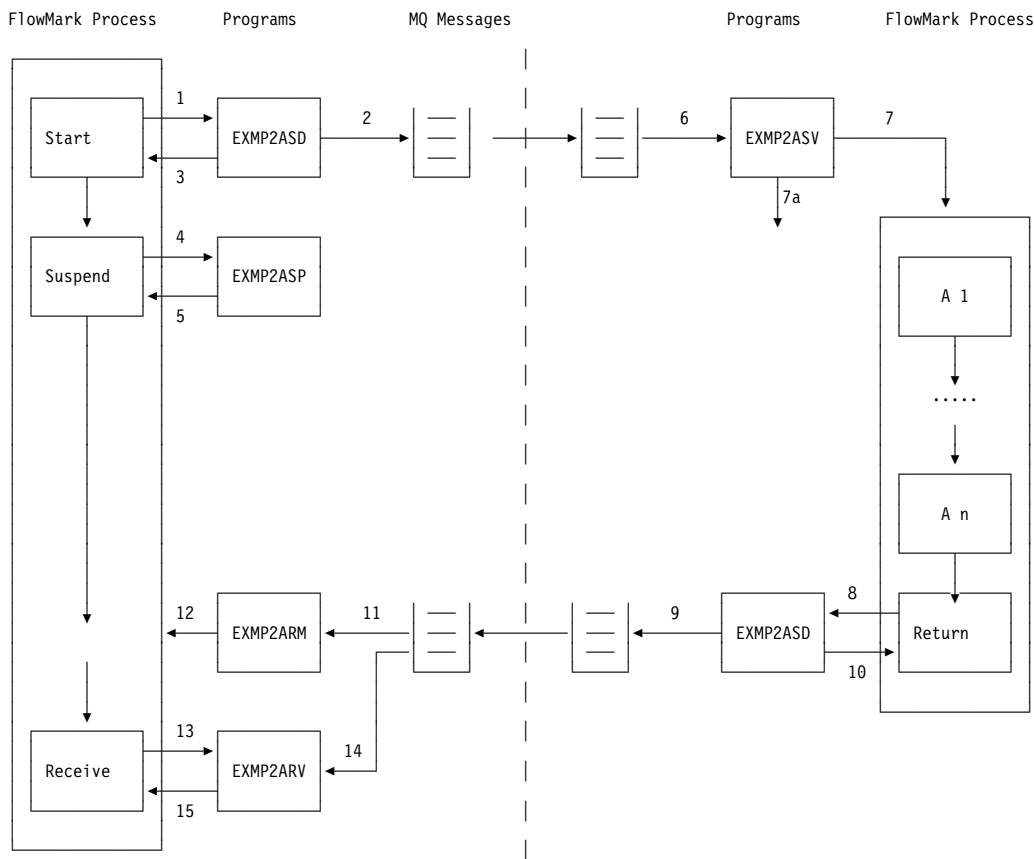


Figure 24. Process and data flow: Start LAN from LAN. Overview about the process and data flow when starting FlowMark on OS/2 or AIX processes from a FlowMark on OS/2 or AIX process.

The remote FlowMark on OS/2 or AIX process is started by a local FlowMark on OS/2 or AIX process consisting of the activities:

- Start
- Suspend
- Receive

The **Start** activity invokes **EXMP2ASD** (1) and passes data to it through command-line parameters and the FlowMark input container.

EXMP2ASD uses MQ calls to create an MQ message. In addition, it encodes the data (including all input container data and specific process-relevant data, for example, instance name of the starting process) within the application data and passes the message to the queue specified within the command-line data (2).

EXMP2ASD sets specific members within the output container to return (3) process-relevant information (for example, FMMQI_CONTROL.InstanceName).

The **Suspend** activity invokes **EXMP2ASP** (4), which suspends the current process (5).

The connected remote queue is continuously read (using blocking mechanisms) by the daemon program **EXMP2ASV** (6). **EXMP2ASV** decodes the encoded data from the message and starts (7) the FlowMark on OS/2 or AIX process that is specified within the data. The container data of the message is passed to the process as initial data and some additional data items are automatically provided (for example, FMMQI_CONTROL.InstanceName). These additional data items are described in “EXMP2ASV return data” on page 203.

EXMP2ASV invokes an exit function (7a) after the process start and passes all relevant data (such as success, instance names) to it. This can be used to provide logging and alert possibilities.

The remote FlowMark on OS/2 or AIX process can consist of several activities **A1** to **An**. The last activity is the **Return** activity, which invokes **EXMP2ASD** with a reply request (8).

EXMP2ASD uses the data passed via its command line and from the input container. It uses MQ calls to create an MQ message. In addition, it encodes the data (including all input container data and specific process-relevant data, for example, instance name of the starting process) within the application data and passes the message to the queue specified with the command-line data (9).

EXMP2ASD sets specific members within the output container to return (10) process-relevant information (for example, FMMQI_CONTROL.ReasonCode).

The reply queue is continuously browsed (using blocking mechanisms) by the daemon program **EXMP2ARM** (11). The message contains the original message ID as correlation ID within the MQ header and data (including container data). The data is encoded.

EXMP2ARM retrieves the instance name from the message and resumes this specific instance (12).

The **Receive** activity invokes **EXMP2ARV** (13) and, via command-line parameters, passes data (including the instance name of the remote process) to it.

EXMP2ARV uses MQ calls to read the MQ message (14). It generates the correlation ID from the remote instance name returned in step (3) to get the appropriate message.

EXMP2ARV sets specific members within the output container to return process-relevant information and returns all container data from the message to the output container (15), if there are matching item names.

Controlling FlowMark on OS/2 or AIX from FlowMark on OS/2 or AIX

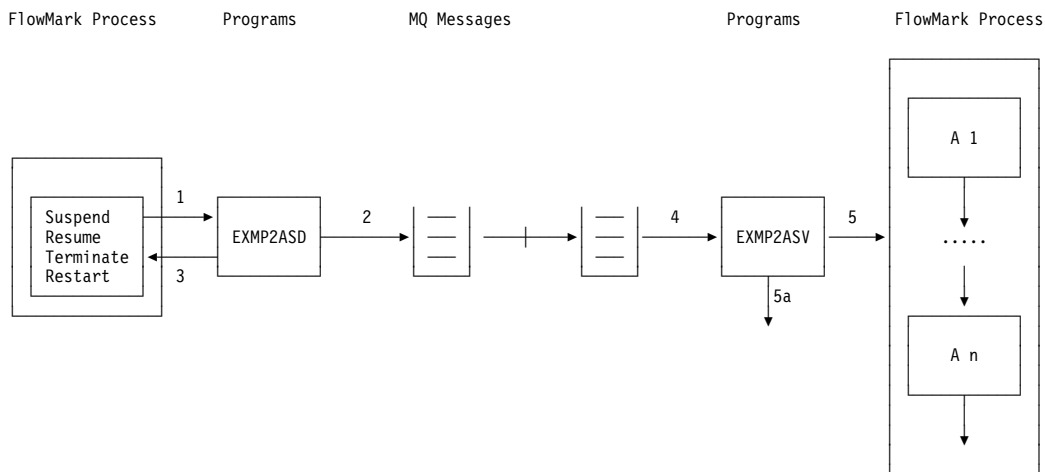


Figure 25. Process and data flow: Control LAN from LAN. Overview about the process and data flow when controlling FlowMark on OS/2 or AIX processes from a FlowMark on OS/2 or AIX process.

Assumption: A remote FlowMark on OS/2 or AIX process has been started by a local FlowMark on OS/2 or AIX process as described in “Starting FlowMark on OS/2 or AIX from FlowMark on OS/2 or AIX” on page 194 and is currently running.

The **Suspend, Resume, Terminate** or **Restart** activity invokes **EXMP2ASD** (1) and passes data (including remote instance name and requested action) to it via command-line parameters.

EXMP2ASD uses MQ calls to create an MQ message. In addition, it encodes the data within the application data and passes the message to the queue specified with the command-line data (2).

EXMP2ASD sets specific members within the output container to return (3) process-relevant information (for example, FMMQI_CONTROL.ReasonCode).

The connected remote queue is continuously read (using blocking mechanisms) by the daemon program **EXMP2ASV** (4). **EXMP2ASV** decodes the encoded data from the message and performs the requested action (5) on the specified FlowMark on OS/2 or AIX process instance.

EXMP2ASV invokes an exit function (5a) after performing the requested action and passes all relevant data (such as success, instance names) to it. This can be used to provide logging and alert possibilities.

General considerations

This section contains information about:

- “Unique process-instance ID”
- “Process tracking and alert events”
- “Predefined data structures”
- “Sample FDL”
- “Sample MQSeries definitions”

Unique process-instance ID

A unique process-instance ID consists of a 3-digit random number, the MQSeries message ID, and a static eye-catcher.

The random number is created by the starting process and passed to the other environment within the encoded process-control data of the message. The purpose of the random number at the beginning of the unique instance ID is to ease update operations on DB2 tables by distributing the key.

The message ID is converted to a hexadecimal representation to ensure that blanks or X'0' values are converted to displayable characters.

The eye-catcher serves for identification purposes within process-instance lists.

The resulting length of the instance name is 54 characters.

For example (in ASCII):

Random number	5
Message ID	ABCDEFGHIJKLMNQRSTUWXYZ
Eye-catcher	FMQ
Resulting unique instance name	005414243444546474849A4B4C4D4E4F505152535455565758FMQ

Process tracking and alert events

Process tracking and alert event messages are supported by calling exit functions within EXMP2ASV and EXMP2ARM. These exit functions are implemented by DLL functions within EXMP2AEX.DLL. The default implementation supplied by IBM just writes error messages to STDOUT (standard output). If customers want to implement process tracking or event tracking, they have to change these exits.

Predefined data structures

There is one predefined data structure that is implicitly used by building block for FlowMark MVS/ESA: FMMQI_CONTROL

Figure 26. Predefined FMMQI_CONTROL data structure

Name	Type	Meaning												
ReasonCode	Long	Depends on return code: <table border="1"> <thead> <tr> <th>RC</th> <th>Reason code</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> </tr> <tr> <td>4</td> <td>FlowMark on OS/2 or AIX API return code</td> </tr> <tr> <td>8</td> <td>FlowMark for MVS/ESA MQ reason code</td> </tr> <tr> <td>12</td> <td>FlowMark on OS/2 or AIX API return code</td> </tr> <tr> <td>16</td> <td>Internal</td> </tr> </tbody> </table>	RC	Reason code	0	0	4	FlowMark on OS/2 or AIX API return code	8	FlowMark for MVS/ESA MQ reason code	12	FlowMark on OS/2 or AIX API return code	16	Internal
RC	Reason code													
0	0													
4	FlowMark on OS/2 or AIX API return code													
8	FlowMark for MVS/ESA MQ reason code													
12	FlowMark on OS/2 or AIX API return code													
16	Internal													
ParentProcessName	String	Name of the parent process (to be used by send reply)												
InstanceName	String	Complete unique instance name												
ReplyQueue	String	Name of the reply queue												
ReplyQueueManager	String	Name of the reply queue manager												

Sample FDL

The sample FDL (file EXMP2ABB.FDL in the BIN subdirectory) supports all scenarios described in "Sample scenarios" on page 187.

Sample MQSeries definitions

The sample MQSeries definitions (file EXMP2ABB.MQI in the BIN subdirectory) support all scenarios described in "Sample scenarios" on page 187.

EXMP2ASD

EXMP2ASD starts a new process within a remote environment, controls the execution (suspend, resume, terminate, restart) of a remote process, or returns the output of a child process to its remote parent process (reply).

The command syntax is:

```

▶▶ exmp2asd [q_mgr] | action | output_q

```

action:

```

| start process_name reply_q |-----|
| suspend current instance |-----|
| resume all |-----|
| terminate |-----|
| restart |-----|
| reply parent_process parent_instance correl_name output_q_mgr |-----|

```

The command parameters are:

q_mgr The name of the local queue manager the program connects to. If an asterisk (*) is specified, the default queue manager is used.

<i>process_name</i>	The name of the process to be started.
<i>reply_q</i>	The name of the reply queue to be used by the subprocess.
<i>output_q_</i>	The name of the output queue the message is put into.
current	Suspend or resume just the <i>current</i> instance.
all	Suspend or resume the current instance and <i>all</i> of its subprocesses.
<i>instance</i>	The instance name of the process to be manipulated.
<i>parent_instance</i>	The name of the parent process instance to which the reply is sent.
<i>parent_process</i>	The name of the parent process to which the reply is sent. If the target is not FlowMark for MVS/ESA, this value is currently not used. If the parent is FlowMark on OS/2 or AIX, the value of FMMQI_CONTROL.ParentProcessName is always FMMQI_UNKNOWN in the current implementation.
<i>correl_name</i>	The correlation name to be used for the reply (the MQ correlation ID is built from this name). If an asterisk (*) is specified, the current instance name is used.
<i>output_q_mgr</i>	The name of the queue manager where the output queue is defined. If an asterisk (*) is specified, the local default queue manager is used.

EXMP2ASD return data

The return data in the following output container members can be:

_rc

Return code

Value	Meaning
0	Successful completion
4	FlowMark API error, no message sent
8	Communications error, no message sent
12	FlowMark API error, message already sent
16	Invocation error
32	Internal error

FMMQI_CONTROL.ReasonCode

Reason code (see "Predefined data structures" on page 197)

FMMQI_CONTROL.InstanceName

Unique instance name of the started process

EXMP2ASD message handling

The message is built from the encoded data including all input container items without any members starting with _ (FlowMark system variables) or FMMQI_CONTROL (FMMQI control variables).

EXMP2ASP

EXMP2ASP suspends its own or the current process, respectively.

The command syntax is:

▶—exmp2asp—————▶

There are no invocation parameters.

EXMP2ASP return data

The return data in the following output container members can be:

_rc

Return code

Value	Meaning
0	Successful completion
4	FlowMark API error, process not suspended
16	Invocation error
32	Internal error

FMMQI_CONTROL.ReasonCode

Reason code (see “Predefined data structures” on page 197)

EXMP2ARM

EXMP2ARM is a daemon program that browses its input queue for reply messages and suspends the FlowMark on OS/2 or AIX processes associated with the messages read.

The command syntax is:

▶—exmp2arm—q_mgr—input_q—userid—password—database—server—xxx—————▶

The command parameters are:

q_mgr The name of the local queue manager the program connects to. If an asterisk (*) is specified, the default queue manager is used.

input_q The name of the input queue (local).

<i>userid</i>	The FlowMark user ID. If an asterisk (*) is specified, the current value of the environment variable FMMQI_USERID is used.
<i>password</i>	The password associated with the FlowMark user ID. If an asterisk (*) is specified, the current value of the environment variable FMMQI_PASSWORD is used.
<i>database</i>	The name of the FlowMark database.
<i>server</i>	The name of the FlowMark server.
<i>xxx</i>	Additional arguments that are passed to the user-exit DLL EXMP2AEX.DLL. The meaning of these parameters is defined by the user's exit code.

EXMP2ARM return data

The return data can be:

Return code	Meaning
0	Successful completion
12	FlowMark API error
16	Invocation error
20	Exit error
32	Internal error

EXMP2ARM message handling

The message is retrieved by an MQSeries MQGET API call (with BROWSE_NEXT option).

The name of the process instance to be resumed is built from the encoded Parent_Process_Instance.

EXMP2ARV

EXMP2ARV reads a message from its input queue and writes the encoded container data within the message to its output container.

The command syntax is:

```

▶▶ exmp2arv q_mgr input_q process_instance ▶▶

```

*

The command parameters are:

<i>q_mgr</i>	The name of the local queue manager the program connects to. If an asterisk (*) is specified, the default queue manager is used.
<i>input_q</i>	The name of the input queue (local).

process_instance The unique process-instance name of the process that has sent the reply message.

EXMP2ARV return data

The return data in the following output container members can be:

_rc	Return code
Value	Meaning
0	Successful completion
8	Communications error, no message read
12	FlowMark API error, message already read
16	Invocation error
32	Internal error

FMMQUI_CONTROL.ReasonCode

Reason code (see “Predefined data structures” on page 197)

EXMP2ARV message handling

The message is read by an MQGET call with the parameters:

- *qname=input_q*
- *correlid=built from process_instance*

EXMP2ASV

EXMP2ASV starts a new process within a local environment (passing the container data of the message) or controls the execution (suspend, resume, terminate, restart) of a local process.

The command syntax is:

```

▶—exmp2asv—q_mgr—input_q—userid—password—database—server—xxx—▶

```

The command parameters are:

<i>q_mgr</i>	The name of the local queue manager the program connects to. If an asterisk (*) is specified, the default queue manager is used.
<i>input_q</i>	The name of the input queue (local).
<i>userid</i>	The FlowMark user ID. If an * is specified, the current value of the environment variable FMMQUI_USERID is used.
<i>password</i>	The password associated with the FlowMark user ID. If an asterisk (*) is specified, the current value of the environment variable FMMQUI_PASSWORD is used.

<i>database</i>	The name of the FlowMark database.
<i>server</i>	The name of the FlowMark server.
<i>xxx</i>	Additional arguments that are passed to the user-exit DLL EXMP2AEX.DLL. The meaning of these parameters is defined by the user's exit code.

EXMP2ASV return data

The return data can be one of the following return codes:

Return code	Meaning
0	Successful completion
4	FlowMark API error during startup, no messages received
8	Communications error during startup, no messages received
16	Invocation error
20	Exit error
32	Internal error

The return data in one of the following container members can be:

FMMQUI_CONTROL.CorreName

The correlation name to be used when invoking EXMP2ASD with the REPLY action

FMMQUI_CONTROL.ParentProcessName

The name of the parent process to be used when invoking EXMP2ASD with the REPLY action

FMMQUI_CONTROL.InstanceName

The unique instance name of the parent process to be used when invoking EXMP2ASD with the REPLY action

FMMQUI_CONTROL.ReplyQueue

The name of the reply queue

FMMQUI_CONTROL.ReplyQueueManager

The name of the reply queue manager

EXMP2ASV message handling

The message is read by an MQGET with the parameter:

- *qname=input_q*

Chapter 9. Building block for AS/400 support

The programs in this building block are designed for general purpose use and are available for OS/2 and AIX.

You can use this building block with FlowMark:

- To access AS/400 applications from FlowMark processes
- To control (that is, to start, suspend, resume, terminate, and restart) FlowMark processes from AS/400 applications

The package contains the following:

- Executables for OS/2 and AIX (EXMP24SD.EXE, EXMP24RC.EXE, and EXMP24SV.EXE)
- Executables for AS/400 (EXMP24EU.SAV) which have to be transferred to the AS/400
- Sample code (EXMP24BB.FDL and EXMP24BB.MQI)

MQSeries definitions for AS/400 support

The following describes the setup for OS/2, but the equivalent setup applies to AIX.

A local queue manager must already be installed on each of the machines

With MQSeries, you can use various communications protocols. Therefore, the standard and tuning settings can vary from installation to installation. The following descriptions show sample minimum definitions in an APPC environment.

For more complex definitions, refer to the MQSeries documentation, especially to the *MQSeries Distributed Queuing Guide* and the *MQSeries Command Reference*.

Customizing the MQSeries definitions for AS/400 access

Figure 27 shows the relationship of the MQSeries elements.

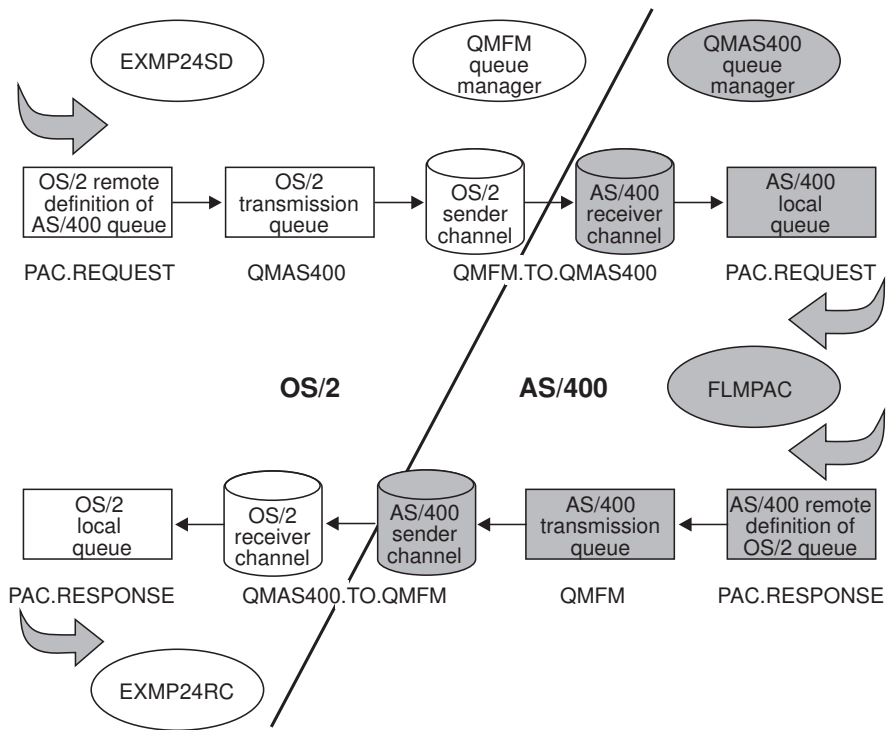


Figure 27. MQSeries topology for the sample process

On OS/2, edit the sample file EXMP24BB.MQI. Modify the definitions according to your MQSeries definitions and your network installation parameters.

Note: In the examples, statements where values in the definitions must match are marked with numbers (such as **3**) to the right of the statement.

- Local queues

```
DEFINE QLOCAL('PAC.RESPONSE') + 1
  REPLACE +
  DESCR('QMFM local queue') +
  PUT(ENABLED) +
  DEFPRTY(5) +
  DEFPSIST(YES) +
  GET(ENABLED)
```

```
DEFINE QLOCAL('CTL.REQUEST') + 2
  REPLACE +
  DESCR('QMFM local queue') +
  PUT(ENABLED) +
  DEFPRTY(5) +
  DEFPSIST(YES) +
  GET(ENABLED)
```

- Remote queues

```
DEFINE QREMOTE('PAC.REQUEST') +
  REPLACE +
  DESCR('AS/400 local queue') +
  PUT(ENABLED) +
  DEFPRTY(5) +
  DEFPSIST(YES) +
  XMITQ('QMAS400') + 3
  RNAME('PAC.REQUEST') + 4
  RQMNAME('QMAS400')
```

```
DEFINE QREMOTE('CTL.RESPONSE') +
  REPLACE +
  DESCR('AS/400 local queue') +
  PUT(ENABLED) +
  DEFPRTY(5) +
  DEFPSIST(YES) +
  XMITQ('QMAS400') + 3
  RNAME('CTL.RESPONSE') + 5
  RQMNAME('QMAS400')
```

- Transmission queue

```
DEFINE QLOCAL('QMAS400') + 3
  REPLACE +
  DESCR('transmission queue to the AS/400') +
  USAGE(XMITQ)
```

- Sender channel

```

DEFINE CHANNEL(QMFM.TO.QMAS400) +
  CHLTYPE(SDR) +
  TRPTYPE(LU62) +
  REPLACE +
  DESCR('sender channel to the AS/400') +
  BATCHSZ(5) +
  DISCINT(0) +
  TPNAME(AS400) +
  MODENAME(LU62) +
  SHORTTMR(60) +
  SHORTRTY(10) +
  LONGTMR(6000) +
  LONGRTY(10) +
  SEQWRAP(999999999) +
  MAXMSGL(10000) +
  CONVERT(NO) +
  XMITQ('QMAS400') +
  CONNAME('400PLU')

```

- Receiver Channel

```

DEFINE CHANNEL(QMAS400.TO.QMFM) +
  CHLTYPE(RCVR) +
  TRPTYPE(LU62) +
  REPLACE +
  DESCR('receiver channel from the AS/400') +
  BATCHSZ(5) +
  PUTAUT(DEF) +
  SEQWRAP(999999999) +
  MAXMSGL(10000)

```

The NDF file on OS/2 must contain statements similar to the following:

```

DEFINE_LOCAL_CP  FQ_CP_NAME(xxxxxxx.xxx)
                  CP_ALIAS(xxx)
                  ...
DEFINE_PARTNER_LU  FQ_PARTNER_LU_NAME(xxxxxxx.AS400)
                   PARTNER_LU_ALIAS(400PLU)
                   ...
DEFINE_TP         TP_NAME(AS400)
                  FILESPEC(x:\MQM\BIN\AMQCRS6A.EXE)
                  PARM_STRING(/M QMFM /N AS400)
                  ...

```


On the AS/400, make sure that the object authority for the libraries QMQM and QMQMDATA is defined as *USE *PUBLIC. Then, edit the sample file MQSETUP in QZWM/QCLSRC and define the following:

1. Local, remote, and transmission queues

Considerations:

- a. The name of this local queue must be used in the definition of the remote queue on OS/2.
- b. Set the message length (MAXMSGLEN) in the definition of the transmission queue at least to the value set for the remote queue plus 4000.

```

CRTMQMQ QNAME (PAC.REQUEST)           4
        QTYPE (*LCL)
        DFTPTY (5)
        MAXMSGLEN (5000)
CRTMQMQ QNAME (CTL.RESPONSE)          5
        QTYPE (*LCL)
        DFTPTY (5)
        MAXMSGLEN (5000)
CRTMQMQ QNAME (PAC.RESPONSE)
        QTYPE (*RMT)
        DFTPTY (5)
        RMTQNAME (PAC.RESPONSE)       1
        RMTQMNAME ('QMFM')
        TMQNAME ('QMFM')              12

CRTMQMQ QNAME (CTL.REQUEST)
        QTYPE (*RMT)
        DFTPTY (5)
        RMTQNAME ('CTL.REQUEST')      2
        RMTQMNAME ('QMFM')
        TMQNAME ('QMFM')              12

CRTMQMQ QNAME (QMFM)                   12
        QTYPE (*LCL)
        MAXMSGLEN (10000)
        USAGE (*TMQ)

```

2. Dedicated channels to the FlowMark server or client on OS/2

Considerations:

- a. Use the same name for the sender channel on one system as for the receiver channel on the other system.
- b. The value for the message-length parameter (MAXMSGLEN) must be at least 5000. Values below 5000 result in a FlowMark Program Access-internal error when FlowMark Program Access writes to the MQSeries message queue.

CRTMQMCHL CHLNAME(QMAS400.TO.QMFM)	9
CHLTYPE(*SDR)	
TRPTYPE(*LU62)	
CONNAME(FLMCSI)	13
TMQNAME(QMFM)	12
MAXMSGLEN(10000)	
CRTMQMCHL CHLNAME(QMFM.TO.QMAS400)	6
CHLTYPE(*RCVR)	
TRPTYPE(*LU62)	
MAXMSGLEN(10000)	

3. Routing entry in the communications subsystem

Considerations:

- The compare value (CMPVAL) must match the value specified for TPNAME in the channel definitions (DEFINE CHANNEL) on OS/2.
- The routing entry must be included before the entry for PGMEVOKE, if there is any.

ADDRTGE SBSD(QSYS/QCMN)	
SEQNBR(5)	
CMPVAL(AS400 37)	7
PGM(QMQM/AMQCRC6A)	
CLS(QSYS/QBATCH)	

4. Communication side information

Considerations:

- The communication side information (CSI) is used only for sending messages from OS/400 to OS/2.
- The remote location name (RMTLOCNAME) must match the LU name in the OS/2 NDF file.
- The transaction program name (TNSPGM) must match the TP name in NDF file (DEFINE_TP TP_NAME(*name*)).

CRTCSI CSI(QSYS/FLMCSI)	13
RMTLOCNAME(xxx)	10
TNSPGM(AS400)	11

5. Communication entry

ADDCMNE SBSD(QSYS/QCMN)
DEV(*ALL)
DFTUSR(QPGMR)
MODE(*ANY)

Customizing the MQSeries definitions for controlling FlowMark processes from the AS/400

Figure 28 shows the relationship of the MQSeries elements.

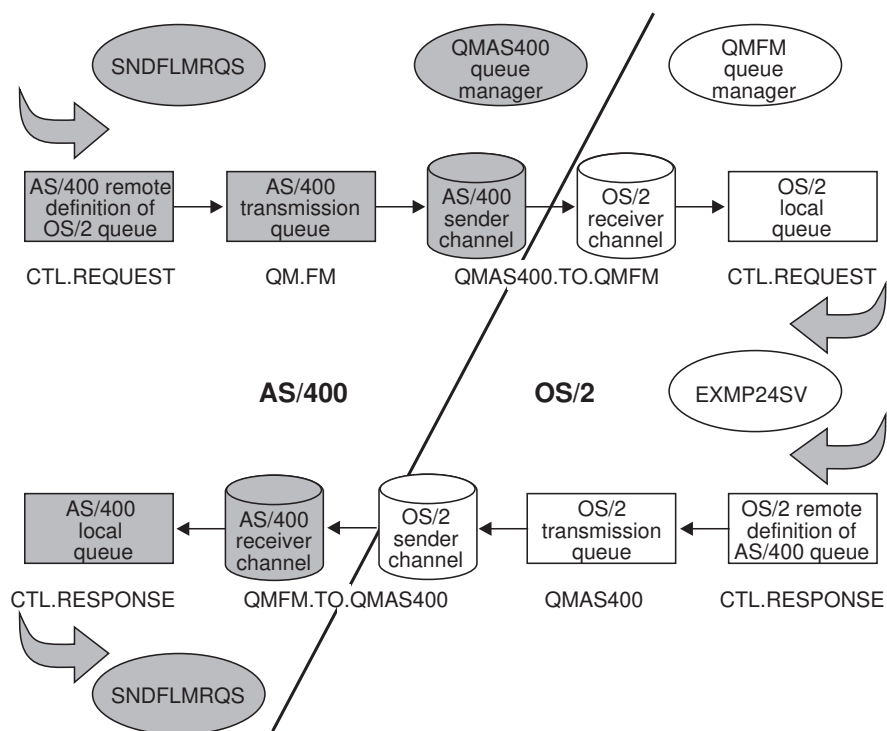


Figure 28. MQSeries topology for controlling FlowMark processes from the AS/400

Accessing AS/400 applications from FlowMark processes

The following executables are provided with FlowMark to enable access to AS/400 applications:

- EXMP24SD.EXE** Sends an MQSeries message to a *FlowMark Program Access* on the AS/400, which calls the appropriate AS/400 application
- EXMP24RC.EXE** Receives the message that is returned from the *FlowMark Program Access* when the AS/400 application has completed

You send the following information to the *FlowMark Program Access*:

- The name of the AS/400 application program
- The name of the AS/400 library where the program resides
- The AS/400 user ID
- Up to 1 KB of data, such as parameters

Before you can work with the building block for AS/400 support, do the following:

1. Set up the MQSeries environment for the FlowMark Program Execution client on OS/2 or AIX and for the AS/400. How to do this is shown in “Customizing the MQSeries definitions for AS/400 access” on page 206.
2. Transfer the file EXMP24EU.SAV in binary format to the AS/400. Then restore the library QZWM and add it to the library list. How to do this is described in the *Installation and Maintenance* manual.
3. Define a FlowMark subprocess with the programs EXMP24SD and EXMP24RC. An example for such a subprocess is shown in Figure 30 on page 214. The FDL for this example is provided with the product (EXMP24BB.FDL).

To work with the building block for AS/400 support, do the following:

1. Start a FlowMark Runtime server or a FlowMark Runtime client. This starts the FlowMark Program Execution client on the OS/2 or AIX workstation.
2. Sign on to the AS/400.
3. Start the message queue manager on the AS/400.
4. Start the message queue manager on the OS/2 or AIX workstation.
5. On the OS/2 or AIX workstation, start the channel to the AS/400.
6. On the AS/400, start the channel to the OS/2 or AIX workstation.
7. Start FlowMark Program Access (in batch or interactive mode) on the AS/400. See “FlowMark Program Access” on page 217 for information on using FlowMark Program Access.

Sample scenario for AS/400 access

This section describes a sample scenario for accessing AS/400 applications from a FlowMark process.

The file EXMP24BB.FDL contains a sample process that shows how EXMP24SD.EXE and EXMP24RC.EXE are used to access an AS/400 application from a FlowMark process.

Note: Include the directory where the files EXMP24SD.EXE and EXMP24RC.EXE reside (for example, MQMBB) in the following statements in your CONFIG.SYS:

- LIBPATH=.;D:\MQMBB;...
- SET PATH=.;D:\MQMBB;...
- SET HELPNDX=D:\MQMBB;

Do the following:

1. Import the FDL in Buildtime
2. If the Program Execution client and the building block programs reside in a directory other than \EXM\BIN, change the path in the settings notebook of the following programs accordingly:
 - MQI_400SetData (file name EXMSSCNT.CMD)
 - MQI_400SendForReply (file name EXMP24SD.EXE)
 - MQI_400Receive (file name EXMP24RC.EXE)
 - MQI_400ShowData (file name EXMSSCNT.CMD)
3. Translate MQI_* processes:
 - a. In the FlowMark Buildtime processes folder, select the processes.
 - b. Select **Translate** from **Selected**.
 - c. In the FlowMark Runtime client, start MQI_400SampleProcess.

The current building block for AS/400 support requires all the parameters within the FlowMark data container that are defined in the sample FDL.

The following parameters are used:

- Send
All parameters except RC and MsgID are used. A MsgID is assigned by MQSeries.
- Receive
Only RC and ParmS are updated in the output data container and MsgID is set to the actual message ID. All other parameters remain unchanged.

The sample process and the sample parameter settings are shown in Figure 29 on page 214 and Figure 30 on page 214. You can use the sample AS/400 programs BATCHAPPL and INTERAPPL (provided in library QZWM) with the sample process.

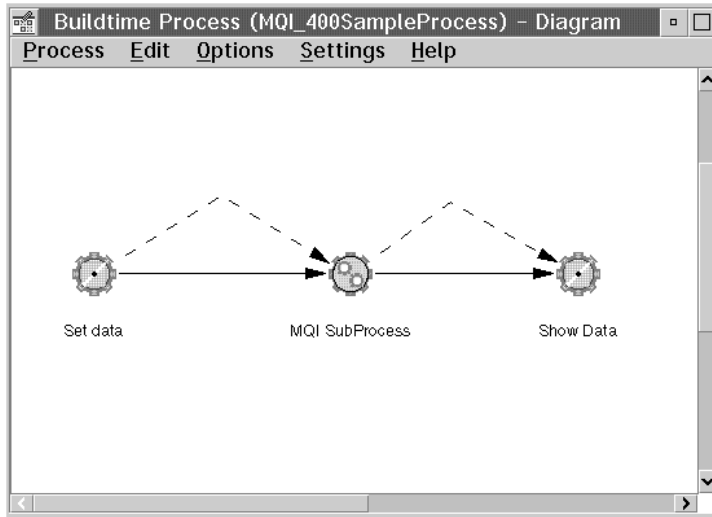


Figure 29. Sample process MQI_400SampleProcess

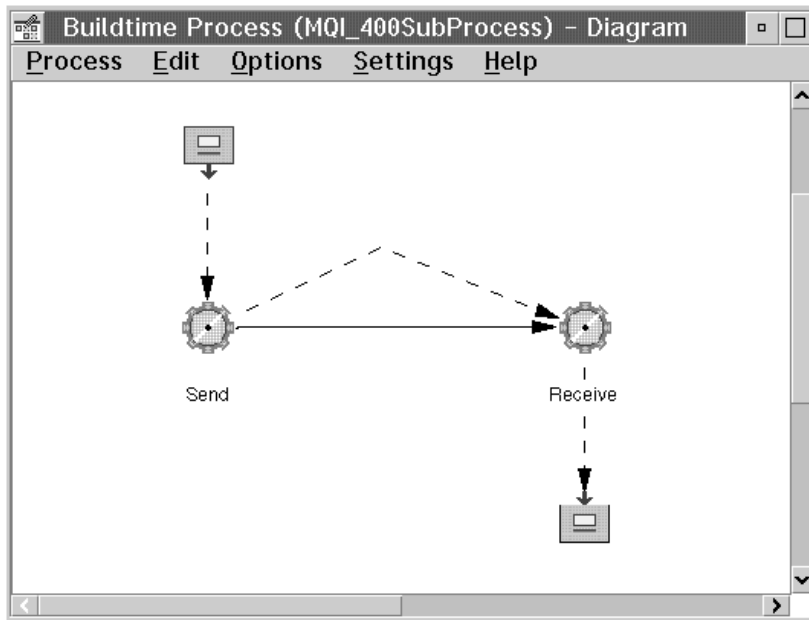


Figure 30. Sample subprocess MQI_400SubProcess

General considerations

This section contains information about:

- “Predefined data structures”
- “Sample FDL”
- “Sample MQSeries definitions”

Predefined data structures

There is one predefined data structure that is implicitly used by EXMP24SD and EXMP24RC: MQI_400Parms.

Figure 31. Predefined MQI_400Parms data structure

Name	Type	Meaning
QMgrName	String	Name of a queue manager
OutQueueName	String	Name of an output queue
InQueueName	String	Name of an input queue
MsgID	String	Message identifier for the communication
CorrelID	String	Correlation identifier to be used for MQSeries (equivalent to the AS/400 user ID)
Lib	String	Name of an AS/400 library
Pgm	String	Name of an AS/400 program
Rc	String	Return code provided by the AS/400 program
Parms	String	Parameters required for the AS/400 program

Sample FDL

The sample FDL (file EXMP24BB.FDL in the BIN subdirectory) supports the sample scenario.

Sample MQSeries definitions

The sample MQSeries definitions (file EXMP24BB.MQI in the BIN subdirectory) supports the sample scenario.

EXMP24SD

EXMP24SD sends a message to the AS/400 to start a defined application.

The command syntax is:

```
►►—exmp24sd—q_mgr:out_q—q_mgr:in_q—correl_id—————►►
```

The command parameters are:

q_mgr:out_q The name of the local queue manager to which the program connects and the name of the output queue, that is, the remote definition of the AS/400 local queue.

<i>q_mgr:in_q</i>	The name of the local queue manager to which the program connects and the name of the input queue.
<i>correl_id</i>	The correlation ID, that is, the AS/400 user ID.

All other parameters are taken from the input container of the respective activity.

EXMP24SD return data

The return data in the following output container members can be:

_rc	Return code
	Value Meaning
	0 Successful completion
	4 FlowMark error, no message sent
	8 MQSeries error, no message sent
	32 Internal error

EXMP24SD message handling

Error messages are written to the file MQBB400.TRC in the current directory.

EXMP24RC

EXMP24RC receives a message from the AS/400 to indicate that a defined application has been processed.

The command syntax is:

▶—*exmp24rc—q_mgr:in_q—msg_id*—————▶

The command parameters are:

<i>q_mgr:in_q</i>	The name of the local queue manager to which the program connects and the name of the input queue.
<i>msg_id</i>	The message ID that was set by EXMP24SD.

EXMP24RC return data

The return data in the following output container members can be:

_rc	Return code
	Value Meaning
	0 Successful completion
	4 FlowMark error, no message sent
	8 MQSeries error, no message sent
	32 Internal error
	99 The defined AS/400 program could not be found

parms The parameters as changed by the AS/400 program (up to 1 KB of data)

EXMP24RC message handling

Error messages are written to the file MQBB400.TRC in the current directory.

FlowMark Program Access

The FlowMark Program Access program is available for batch and for interactive applications. The commands to start and stop the FlowMark Program Access are:

- Start FlowMark Program Access (**STRFLMPAC**)

```
Start FlowMark Program Access (STRFLMPAC)

Type choices, press Enter.

Program type . . . . . _____ *BATCH, *INTERACTIVE
Input queue . . . . . _____
Wait time . . . . . *UNLIMITED 0-99999, *UNLIMITED

F3=Exit  F4=Prompt  F5=Refresh  F12=Cancel  F13=How to use this display
F24=More keys

Bottom
```

The parameters are:

- Program type** Indicates if the FlowMark Program Access is to be used for batch or interactive AS/400 applications.
- If *BATCH is selected, the appropriate job is started in the batch subsystem.
- If *INTERACTIVE is selected, the screen is locked until processing of the interactive application is complete.
- Input queue** The name of the MQSeries queue from which FlowMark Program Access reads the messages sent by FlowMark.
- Wait time** The amount of time, in seconds, for how long the FlowMark Program Access waits for a message to arrive. If this parameter is set to *UNLIMITED, FlowMark Program Access must be stopped explicitly with the **ENDFLMPAC** command.

FlowMark Program Access logs in the history log which applications were called.

- End FlowMark Program Access (**ENDFLMPAC**)

```

                                End FlowMark Program Access (ENDFLMPAC)

Type choices, press Enter.

User ID . . . . . _____ Name
Input queue . . . . . _____

                                Bottom
F3=Exit  F4=Prompt  F5=Refresh  F12=Cancel  F13=How to use this display
F24=More keys

```

FlowMark Program Access is stopped when the current application that was started by FlowMark Program Access is completed. ENDFLMPAC sends a high priority message with stop values to the message queue. This message is read before any other start messages, and FlowMark Program Access terminates.

The parameters are:

- User ID** By default, the ID of the user currently logged on. To terminate FlowMark Program Access for a different user, change this value.
- Input queue** By default, the queue name that was used to start FlowMark Program Access. To terminate FlowMark Program Access for a different queue, change the value.

Controlling FlowMark processes from AS/400 applications

FlowMark process instances on OS/2 or AIX can be controlled from the AS/400 with the command SNDFLMRQS on the AS/400 and the program EXMP24SV.EXE on OS/2 or AIX. That is, process instances can be started, terminated, suspended, resumed, and restarted.

The command SNDFLMRQS sends requests for FlowMark control actions to an MQSeries message queue. The EXMP24SV daemon monitors a local MQSeries queue for messages requesting a FlowMark process-instance action that have been sent by SNDFLMRQS and performs the requested actions. If a process instance is started, the input container is initialized with the data provided in the message.

If the MQSeries message type set with the SNDFLMRQS command is *RQST, EXMP24SV sends a reply message to the answer queue specified by SNDFLMRQS.

Before you can work with this command, you must do the following:

1. Set up the MQSeries environment for the FlowMark Runtime server and the AS/400 as described in “Customizing the MQSeries definitions for controlling FlowMark processes from the AS/400” on page 211.

2. Transfer the file EXMP24EU.SAV to the AS/400. Then restore the library QZWM and add it to the library list. How to do this is described in the *Installation and Maintenance* manual.

To work with the programs, do the following:

1. Start the EXMP24SV daemon.
2. Sign on to the AS/400.
3. Start the message queue manager on the AS/400.
4. Start the message queue manager on the OS/2 or AIX workstation.
5. On the OS/2 or AIX workstation, start the channel to the AS/400.
6. On the AS/400, start the channel to the OS/2 or AIX workstation.
7. Submit the command SNDFLMRQS.

EXMP24SV

EXMP24SV receives a message from the AS/400 with the request to control a defined process instance. The control actions are: start, suspend, resume, terminate, and restart.

The command syntax is:

```
▶▶—exmp24sv—q_mgr:in_q—fm_user_id—fm_password—fm_database—fm_server————▶▶
```

The command parameters are:

<i>q_mgr:in_q</i>	The name of the local queue manager to which the program connects and the name of the input queue
<i>fm_user_id</i>	The FlowMark user ID
<i>fm_password</i>	The FlowMark password for <i>fm_user_id</i>
<i>fm_database</i>	The name of the FlowMark database
<i>fm_server</i>	The name of the FlowMark server

EXMP24SV return data

The return data in the following output container members can be:

_rc	Return code	
	Value	Meaning
	0	Successful completion
	4	FlowMark error
	8	MQSeries error
	32	Internal error

EXMP24SV message handling

Error messages are written to the file MQBB400S.TRC in the current directory.

Send FlowMark request (SNDFLMRQS)

This AS/400 command is used to control FlowMark processes.

```
Send FlowMark Request (SNDFLMRQS)

Type choices, press Enter.

FlowMark Process name . . . . . _____
FlowMark Process instance name _____
FlowMark Action . . . . . _____ *START, *SUSPEND, *RESUME...
FlowMark API mode . . . . . _____ *CURRENT, *ALL, *QUERY
MQSeries Message type . . . . . _____ *DTGRM, *RQST
MQSeries Queue manager . . . . . _____
MQSeries Output queue . . . . . _____
FlowMark Container variables . . _____
      + for more values
MQSeries Input queue . . . . . _____

Bottom
F3=Exit  F4=Prompt  F5=Refresh  F12=Cancel  F13=How to use this display
F24=More keys
```

The parameters are:

FlowMark Process name

The name of FlowMark process to be controlled. This parameter is required for the FlowMark action *START.

FlowMark process instance name

The name of FlowMark process instance to be controlled.

FlowMark Action

The action that is to be performed with the process instance.

Possible values are:

- *START
- *SUSPEND
- *RESUME
- *TERMINATE
- *RESTART

If you select *START, you can also define container variables as a controlled parameter.

FlowMark API mode

Additional FlowMark API parameters. Possible values are:

- ***CURRENT**
- ***ALL**
- ***QUERY**

For a detailed description of the values refer to the sections about the respective API functions in the *Programming Guide*.

FlowMark Container variables

If *START is selected as FlowMark action, you can define container variables here. Do this in the following format:

Name:Type=Value

Where:

Name	The name of the FlowMark container variable.
Type	The type of the FlowMark container variable. Possible values are: S string L long F float
Value	The value of the FlowMark container variable. If the value of a string variable contains blanks, enclose the value in double quotes (").

For example:

Name:S=Tony
Age:L=24
Amount:F=12.34
Song:S="O happy day"

Note: The FlowMark container variables must be specified on FlowMark process level.

MQSeries Message type

The type of the MQSeries message. Possible values are:

- *DTGRM** If no reply message is required.
- *RQST** To request a reply message containing the FlowMark return code.

If *RQST is selected, you must also define an MQSeries input queue as a controlled parameter.

MQSeries Queue manager

The name of the AS/400 queue manager.

MQSeries Output queue

The name of the AS/400 output queue.

MQSeries Input queue

The name of the MQSeries input queue for the response messages.

This parameter is optional. It is necessary only if *RQST is specified as message type.

If the MQSeries message type is set to *RQST, SNDFLMRQS logs the actions to the history log.

Appendix A. Ways to integrate FlowMark and Lotus Notes

This appendix gives an overview of those FlowMark components that you can use to integrate FlowMark and Lotus Notes. It also provides some hints and tips when to use which component.

Overview

The following sections describe in short the functions offered by:

- The FlowMark service broker
- The Lotus Notes service broker
- The Runtime client for Lotus Notes

FlowMark service broker

The FlowMark service broker can be invoked from the Lotus Notes environment by starting EXMP3FFR.EXE with command-line parameters. It allows you to manipulate FlowMark process instances and process activities: You can start, suspend, resume, terminate, and restart process instances and start, restart, and finish process activities.

However, this service broker does *not* provide any functions to retrieve information about existing process templates, process instances, or process activities, like:

- Which process templates are available
- Which process instances are available
- What is the current status of a specific process instance
- Which process activities are currently on my worklist
- What is the actual status of a process activity on my worklist

So, the FlowMark service broker is suitable if there is a need to start process instances with start data from a predefined process template. For all other functions, there has to be additional communication between the FlowMark and the Lotus Notes environment.

Lotus Notes service broker

The Lotus Notes service broker is invoked from FlowMark as a manual or automatic activity. This service broker allows you to open and close Lotus Notes databases and to create, decrypt, delete, encrypt, read, sign, unsign, and update Lotus Notes documents. You can also select a document using a view with selection criteria.

The Lotus Notes service broker manipulates Lotus Notes documents. The Lotus Notes service broker offers no functions to invoke Lotus Notes scripts or to manipulate the Lotus Notes user interface (for example, display a specific view or form).

The different actions in the Lotus Notes environment must be implemented as FlowMark activities. So, the Lotus Notes service broker is suitable if there are only a few actions to be performed within the Lotus Notes environment, like:

- Open a database, create and sign a document, close the database.
- Open a database, search for and read a document, close the database.

Runtime client for Lotus Notes

The Runtime client for Lotus Notes is invoked as a Lotus Notes application from the Lotus Notes environment. The Runtime client for Lotus Notes allows you to manipulate FlowMark process templates, process instances, and process activities (that is, work items), including all available functions and the retrieval of all associated attributes (for example, the status). The functions are available from a Lotus Notes user interface as well as from API functions accessible by other Lotus Notes applications.

The Runtime client for Lotus Notes is a Lotus Notes application that can be combined with customized Lotus Notes scripts. Therefore, it has access to all Lotus Notes API functions within these scripts. This includes the possibility to implement FlowMark activities by Lotus Notes scripts, which can invoke other Lotus Notes applications displaying specific forms and views.

The Runtime client for Lotus Notes cannot be combined with a native FlowMark Runtime client and it is not possible to implement automatic FlowMark activities as Lotus Notes scripts.

So, the Runtime client for Lotus Notes is suitable if one or both of the following are true:

- Lotus Notes is the corporate desktop environment.
- The parts of a FlowMark activity that are to be performed within the Lotus Notes environment are rather complex.

Tips for selecting the appropriate component

The following provides scenarios to help you select the component most suitable for the integration of FlowMark and Lotus Notes in a specific part of a workflow.

Scenario 1: Two organizations, one using FlowMark, the other Lotus Notes

Organization A uses FlowMark and organization B uses Lotus Notes. All members of organization A use the operating system (for example, workplace shell) as the desktop environment, the members of organization B use Lotus Notes as the desktop environment. In this case, the service brokers are probably the best solution: The Lotus Notes service broker for organization A and the FlowMark service broker for organization B.

If the FlowMark process inside organization A requires data available from organization B, the Lotus Notes service broker can be used to retrieve this data. This service broker can also create or update Lotus Notes documents from automatic activities inside the FlowMark process.

A Lotus Notes application can start FlowMark processes from predefined process templates and can pass start data to the process. It can also manipulate such a process instance afterwards.

In this scenario, the integration of FlowMark and Lotus Notes is primarily focused on the interchange of data (start data on process start and data inside a Lotus Notes document).

Scenario 2: Two organizations using FlowMark and Lotus Notes

Organization A uses FlowMark and organization B uses Lotus Notes and FlowMark. In this case, the Lotus Notes service broker is probably the best solution for A and the Runtime client for Lotus Notes is probably the best solution for B.

If the FlowMark process inside organization A requires data available from organization B, the Lotus Notes service broker can be used to retrieve this data. It can also create or update Lotus Notes documents from automatic activities inside the FlowMark process.

Any Lotus Notes application of organization B can use the Runtime client for Lotus Notes to manipulate FlowMark objects. In addition, the members of organization B can use the Runtime client for Lotus Notes to manipulate the FlowMark objects.

In this scenario, the integration of FlowMark and Lotus Notes is focused on the interchange of data (A and B) and on the integration of the FlowMark client user interface with the Lotus Notes desktop (B).

Scenario 3: One organization using FlowMark and Lotus Notes

There is one organization where Lotus Notes and FlowMark are used. In this case, the Runtime client for Lotus Notes is probably the best solution.

If a FlowMark process needs to access data in a Lotus Notes database, it can use the Runtime client for Lotus Notes in combination with an activity implemented as Lotus Notes application to retrieve or manipulate the data.

Any Lotus Notes application can use the Runtime client for Lotus Notes to manipulate FlowMark objects. In addition, the users can manipulate the FlowMark objects via the Runtime client for Lotus Notes.

In this scenario, the integration of FlowMark and Lotus Notes is focused on the integration of the FlowMark client user interface with the Lotus Notes desktop.

Appendix B. Migrating from a previous version of the Service Broker Manager

This appendix describes what you have to consider when you migrate from a previous version (that is, shipped with Version 2.2 of FlowMark or before) of the Service Broker Manager and the provided service brokers on OS/2 or Windows 3.1.

Migrating the Service Broker Manager on OS/2

You have to delete the following files:

- SBM.INI (Service Broker Manager)
- SBBFM.INI (FlowMark Broker)
- SBBLNTS.INI (Lotus Notes Broker)

You have to adapt your FDL:

1. Export your database to an FDL file.
2. Edit the FDL file and change the names of the Service Broker Manager components according to Figure 32 (for example, replace SBREQLN with EXMP3LRQ).

The following files have been renamed:

Figure 32. Service Broker Manager on OS/2: Renaming

	Old name	New name
Library files	SBBROKER.LIB SBREQ.LIB SBSERVIC.LIB	EXMP3KBR.LIB EXMP3KRE.LIB EXMP3KSV.LIB
Executables	SBPREDIT.EXE SBREQ.EXE SBCNTRL.EXE SBREQFM.EXE SBBFMSB.EXE SBM.EXE SBMADMIN.EXE SBMUSER.EXE	EXMP3CPR.EXE EXMP3FFR.EXE EXMP3UCT.EXE EXMP3FRR.EXE EXMP3FMS.EXE EXMP3UUP.EXE EXMP3UAD.EXE EXMP3UUS.EXE
Standard service broker	SBBSTD.DLL	EXMP3CST.DLL
FlowMark service broker	SBBFM.DLL	EXMP3FBR.DLL
FlowMark service	SBSFM.DLL	EXMP3FFM.DLL
FlowMark service requester	SBREQFM.DLL	EXMP3FRQ.DLL
Lotus Notes service broker	SBBLNTS.DLL	EXMP3LBR.DLL
Lotus Notes service	SBSLNTS.DLL	EXMP3LSE.DLL
Lotus Notes service requester	SBREQLN.DLL	EXMP3LRQ.DLL

If you developed your own service broker, change the name of the following include files and recompile your service broker:

Old name	New name
SBBFM.H	EXMP3FBR.H
SBBROKER.H	EXMP3CBR.H
SBREQ.H	EXMP3FRE.H
SBSERVIC.H	EXMP3CSE.H

Migrating the Service Broker Manager on Windows 3.1

You have to adapt your FDL:

1. Export your database to an FDL file.
2. Edit the FDL file and change the names of the Service Broker Manager components according to Figure 33 (for example, replace SBWRQLN with EXMW3LRQ).

The following files have been renamed:

Figure 33. Service Broker Manager on Windows 3.1: Renaming

	Old name	New name
Library files	SBWBROK.LIB SBWREQ.LIB	EXMW3CAL.LIB EXMW3KRQ.LIB
Executables	SBWREQC.EXE SBWRFM.EXE SBMWIN.EXE	EXMW3FRC.EXE EXMW3FRU.EXE EXMW3USB.EXE
FlowMark service broker	SBWBFM.DLL	EXMW3FBR.DLL
FlowMark service	SBWSFM.DLL	EXMW3FSE.DLL
FlowMark service requester	SBWRFM.DLL	EXMW3FRU.DLL
Lotus Notes service broker	SBWBRLN.DLL	EXMW3LBR.DLL
Lotus Notes service	SBWSVLN.DLL	EXMW3LSE.DLL
Lotus Notes service requester	SBWRQLN.DLL	EXMW3LRQ.DLL

If you developed your own service broker, change the name of the following include files and recompile your service broker:

Old name	New name
SBWFM.H	EXMW3FBR.H
SBWBROK.H	EXMW3CAL.H
SBWREQ.H	EXMW3FRQ.H
SBWSERV.H	EXMW3LSR.H

Glossary

This glossary defines terms and abbreviations used in this publication. If you do not find the term you are looking for, refer to the *IBM Dictionary of Computing*, New York: McGraw-Hill, 1994.

Also consult the OS/2 glossary in the **Information** object on the OS/2 desktop.

This glossary includes terms and definitions from the *Information Technology Vocabulary*, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) after the definition.

A

activity. One of the steps that make up a process. See *program activity* and *process activity*.

activity block. Synonym for *block*.

activity bundle. Synonym for *bundle*.

activity information member. A predefined data structure member associated with the operating characteristics of an activity. They are defined in the activity settings notebook. See also *fixed member* and *process information member*.

activity instance. The processing of an activity as part of a process instance.

activity status. The status of an activity in a process instance that has been started. The status can be one of the following: ready, running, suspended, pending, force-finished, or finished.

animation. A facility for dynamically verifying workflow models. Animating a workflow model lets the user simulate the flow of work through its activities.

animation control panel. A graphical object containing buttons that are used to control the animation of a workflow model.

animation session. A record of the interactions between a modeler who is animating a model and the

animation facility. An animation session can be saved and replayed.

animation work area. An OS/2 work-area folder that contains the following animation objects: control panel, activities, connectors, work-list manager, trace, and setup.

animator action. An action performed by the person animating a workflow model to simulate the action of a program or person.

API. Application Programming Interface.

application program interface. An interface provided by the FlowMark workflow manager that enables programs to start and control process instances, and to work with data containers.

audit trail. A facility for recording events that occur when process instances are run. If the audit flag is set in the model of a process, events in each instance of it are recorded in a flat file for later analysis.

authorization. The attributes of a user's staff definition that determine whether that user can model processes, define staff, start processes, or access the worklists of other users.

B

bend point. A point at which a connector starts, ends, or changes direction.

block. A modeling construct that enables the grouping of related activities in a lower-level diagram, and the modeling of loops and bundles. Synonymous with *activity block*. See also *bundle*.

Buildtime. The FlowMark component, sometimes called Buildtime client, used to define processes.

bundle. A type of block that supports multiple instances of a single program or process activity at run time. The number of instances of the activity is determined at run time by a special program activity called the planning activity. Synonymous with *activity bundle*. See also *block*, *bundle activity*, *pattern activity*, and *planning activity*.

bundle activity. One of the multiple instances of the pattern activity created for an activity bundle at run time. The number of instances is determined by input to the planning activity.

bundle-planning tool. An executable program (EXMPOBCL.EXE for OS/2, exmpobcl for AIX and HP-UX, EXMCOBCL.EXE for Windows, and EXMBOBCL.EXE for Windows NT), which is supplied with the FlowMark product. See also *planning activity*.

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*.

condition. An expression that determines the flow of control through a process instance. See *start condition*, *exit condition*, and *transition condition*.


connector. An arrow drawn between two nodes in a process diagram to signify the flow of control or data between them. See *control connector* and *default connector*.

container. Synonym for *data container*.

control connector. The graphical representation of the flow of control from an activity or block to another, shown as  in a process diagram. See also *default connector* and *transition condition*.

coordinator. A predefined role that is automatically assigned to the person designated to coordinate a role.

D

data connector. The graphical representation of the flow of data, shown as  in a process diagram.


data container. Storage for the input and output data of an activity, block, or process. See *input container* and *output container*.

data item. The element defined by a data structure for which values can be assigned.

data mapping. The specification data transfer from source to target data containers.

data structure member. One of the variables of which a data structure is composed.

database client. A FlowMark component that communicates via TCP/IP or NetBIOS with the Database server. The FlowMark server, Buildtime client, and the Database server itself are database clients.

default connector. The graphical representation of a special kind of control connector, shown as  in a process diagram. Control flows along this connector if no other control path is valid. See also *transition condition*.

Delivery server. A server that is used in a process environment with distributed processes. There is one Delivery server for each database. The Delivery server ensures the communication between the FlowMark Runtime servers.

Delivery server node. The node address (APPC or TCP/IP) of a computer on which the Delivery server for a specific FlowMark Runtime server is running.

dynamic staff assignment. A method of assigning staff to an activity by specifying criteria such as role, organization, or level. The users to receive the activity on their worklists are determined when the activity becomes ready. See also *level*, *organization*, *process administrator*, and *role*.

E

end activity. An activity that has no outgoing control connector.

exit condition. A logical expression that specifies whether control exits from an activity or block.

Export. A FlowMark utility program that (1) in Buildtime converts the definitions of staff, servers, programs, processes, and data structures into an external format called FDL, and (2) in Runtime converts templates, instances, and work items into an external format called FRL. Contrast with *Import*.

F

FDL. FlowMark definition language.

fixed member. A predefined data structure member that provides information about the current activity. The value of a fixed member is set by the FlowMark workflow manager.

FlowMark definition language (FDL). An external format for defining staff, programs, data structures, and workflow models in a flat file. The definitions in the FDL file can then be imported into a FlowMark database. See also *Export* and *Import*.

FlowMark runtime language (FRL). An external format for templates, instances, and work items in a flat file. See also *Export* and *Import*.

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.

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.

FRL. FlowMark runtime language.

I

Import. A FlowMark utility program that (1) in Buildtime takes definitions of staff, server, programs, processes, and data structures in the FlowMark definition language (FDL), and places them in a FlowMark database, and (2) in Runtime takes information about templates, instances, and work items in the FlowMark runtime language (FRL), and places them in a FlowMark database. Contrast with *Export*. See also *FlowMark definition language* and *FlowMark runtime language*.

input container. Storage for data used as input to an activity, process, or block. See also *source* and *data mapping*.

integration building block. Reusable sample code modules that enable the interaction of FlowMark with another software product. These modules can be changed or expanded to meet customers's needs.

L

level. A number from 0 through 9 that is assigned to each person in a FlowMark database. The person who defines staff can assign a meaning to these numbers, such as rank or experience. Level is one of the criteria that can be used to dynamically assign activities to people.

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 true and zero is equivalent to false.) See also *exit condition* and *transition condition*.

M

manager. A predefined role that is automatically assigned to the person designated to head an organization.

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 symbols that can be joined by connectors in a process diagram. Nodes include activities, blocks, sources, and sinks. (2) In a network, a point at which one or more functional units connect channels or data circuits. (l)

notification. A FlowMark facility that can notify a designated person when a process or activity is not completed within the specified time.

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, block, 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 organization*.

pattern activity. The single program or process activity in a bundle from which multiple instances, called bundle activities, are created at runtime.

person (pl. people). A member of staff in the enterprise who has been defined in the FlowMark database.

planning activity. A special program activity that creates, at run time, the required number of bundle activities for a specific bundle. The planning activity must use a program that refers, in its registration, to the bundle-planning tool supplied with the FlowMark product. See also *program activity* and *program registration*.

platform. The operating system environment in which a program runs. FlowMark is a distributed cross-platform (OS/2, AIX, and Windows) application.

process. A set of activities that must be completed to accomplish a given task. See also *subprocess*.

process activity. An activity to which a separate process is assigned. Starting this activity creates an

instance of the referenced process and starts it. Contrast with *program activity*.

process administrator. The person responsible for the smooth execution of a process instance. This person can be specified in the workflow model. Otherwise, the person who starts the process instance is the process administrator.

process category. An attribute that a process modeler can specify for a process. Only users authorized for this category can start and control instances of the process as a top-level process.

process diagram. A graphical representation of a process that shows all its nodes and connectors.

process information member. A predefined data structure member associated with the operating characteristics of a process. They are defined on the **Staff** page in the process settings notebook. See also *fixed member* and *activity information member*.

process instance. An executable copy of a process template in FlowMark Runtime.

process management. The FlowMark Runtime tasks associated with process instances. These consist of creating, starting, suspending, resuming, terminating, restarting, and deleting process instances.

process model. Synonym for *workflow model*.

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. The status can be one of the following: ready, pending, running, suspended, terminated, or finished. Process templates, which are also displayed in the process list, always have a status of translated.

process template. The translated form of a workflow model in FlowMark Runtime. See also *process instance*.

program. A computer-based application that supports the work to be done in an activity. Program activities reference executable programs using the logical names associated with the programs in FlowMark program registrations. See also *program registration*.

program activity. An activity to which a registered program is assigned. Starting this activity invokes the program. Contrast with *process activity*.

program registration. Identification of a program to a FlowMark database so that it can be assigned to a program activity in a workflow model.

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.

Runtime. The FlowMark component, sometimes called Runtime client, used to execute process instances.

S

server (server definition). A name for an external FlowMark server. One can use this name upon defining subprocesses. Such subprocesses are run on the external server the name is referring to.

server definition. In the FlowMark database, you can define Runtime servers that can be used for remote execution of subprocesses. In a subprocess, the server is referenced by the name defined in the database.

service. A service interfaces to the integrated product. A service function receives the user data from the Service Broker Manager and calls the appropriate product APIs to perform the work. The results are returned via the Service Broker Manager to the service requester and then back to the user application.

service broker. A FlowMark component that establishes and maintains a logon session with the product that is integrated. Logon session data is passed to the service via the Service Broker Manager. The services use this logon session data when invoking the product APIs.

Service Broker Manager. A FlowMark component that controls the operation of service broker sessions. This includes the interaction between service requester and services, between service broker and services, and also the initialization of the service brokers and services.

service requester. A service requester is the interface to the user application. The user application calls the service requester APIs to request the product to perform some work. The service requester formats the user data

and issues a request to the Service Broker Manager function which forwards the request to the appropriate service function.

service thread. One or more service threads are started for each service. Each thread receives information from the service requester to call the respective service function. When the function has been called, the thread returns information to the service requester.

sink. The symbol that represents the output container of a process or block.

source. The symbol that represents the input container of a process or block.

staff. The people and their roles, organizations, and levels as defined in a FlowMark database.

start activity. An activity that has no incoming control connector. A start activity becomes ready when the process or block that contains it starts. There can be more than one start activity in a process or block.

start condition. The setting that determines when an activity with incoming control connectors can start.

subprocess. A process instance that is started by a process activity.

substitute. The person to whom an activity is automatically transferred when the person assigned that activity is flagged as absent.

support tool. A program that end users can start from their worklists in FlowMark Runtime to help complete a program or process 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 administrator. (1) A predefined role that conveys all authorizations and that can be assigned to exactly one person in a FlowMark database. (2) The person at a computer installation who designs, controls, and manages the use of the computer system.

T

top-level process. A process that is started from a user's process list or from an application program.

transition condition. A logical expression associated with a control connector. If specified, it must be true for control to flow along the associated control connector. See also *control connector* and *default connector*.

translate. To convert a Buildtime workflow model into a Runtime process template.

U

user ID. An alphanumeric string that uniquely identifies a FlowMark user.

W

workflow. The sequence of activities performed in accordance with the business processes of an enterprise.

workflow model. A complete representation of a process. It consists of the process diagram and settings and the definitions of staff, programs, and data structures associated with the activities of the process. Synonymous with *process model*.

work item. Representation of work to be processed in the context of a workflow process activity in a workflow process instance.

worklist. A list of work items assigned to a user and retrieved from a workflow management system.

worklist view. A subset of work items assigned to a user and retrieved from a workflow management system. The worklist view is determined by filter criteria.

Bibliography

To order any of the following publications, contact your IBM representative or IBM branch office.

FlowMark publications

This section lists the publications included in the FlowMark library.

- *IBM FlowMark: Modeling Workflow*, form number SH19-8241, explains the basic concepts of workflow modeling and describes how to use FlowMark to build and automate a workflow model.
- *IBM FlowMark: Managing Your Workflow*, form number SH19-8243, describes how to use the FlowMark Runtime clients.
- *IBM FlowMark: Programming Guide*, form number SH19-8240, explains the FlowMark application program interfaces (APIs).
- *IBM FlowMark: Installation and Maintenance*, form number SH12-6260, contains information and procedures for installing, administrating, and maintaining FlowMark.
- *IBM FlowMark: Diagnosis Guide*, form number SH19-8239, contains information to correct problems encountered when installing and using FlowMark. The procedure for reporting unresolved errors is included.
- *IBM FlowMark: Application Integration Guide*, form number SH12-6267, describes how to use the service broker concept and FlowMark building blocks to integrate other applications with FlowMark.

An online book, *Online Overview*, is part of the library. It provides an interactive introduction to FlowMark and helps the user to become familiar with:

- The organization of FlowMark components
- FlowMark windows and menus
- FlowMark example process models
- The basics of building and running FlowMark process models

Related publications

- *IBM AIX Version 4.1 for RISC System/6000: Installation Guide*, SC23-2550
- *IBM AIX Version 4.1: System Management Guide; Communications and Networks*, SC23-2526
- *IBM REXX/6000: Reference*, SC24-5708
- *IBM Application Support Facility Version 3: Administration Guide*, SH12-5936
- *OS/2 Warp, Version 3 Control Program Programming Guide*, G25H-7101
- *OS/2 Warp, Version 3 Control Program Programming Reference*, G25H-7102
- *OS/2 Warp, Version 3 Presentation Manager Programming Guide, Advanced Topics*, G25H-7104
- *User's Guide to OS/2 Warp Version 3*, S83G-8300¹
- *Warp Connect with Windows Up and Running*, S25H-7876¹
- *Warp Connect without Windows Up and Running*, S25H-7925¹
- *TCP/IP Version 2.1.1 for DOS: Installation and Administration*, SC31-7047
- *TCP/IP Version 2 for OS/2: Installation and Administration*, SC31-6075
- *MQSeries Distributed Queuing Guide*, SC33-1139
- *MQSeries Command Reference*, SC33-1369
- *AS/400 Advanced Series, Integrated Services for File Server I/O Processor Version 3*, SC41-3123
- *Lotus Notes Release 4: Administrator's Guide*
- *Lotus Notes Release 4: Application Developer's Guide*
- *Lotus Notes Release 4: Database Manager's Guide*

¹ These manuals are not available in print. They are part of the respective Online Product Libraries on the IBM Online Library OS/2 Collection, SK2T-2176.

Index

A

- accessing the container 60, 61
- accessing the FlowMark data 60
- accessing the FlowMark data. 61
- activity, change status 39
- alert event 197
- API
 - service broker 110
 - service requester 126
- AS/400
 - applications, accessing 211
 - sample scenario 213
 - support 205
- automatic refresh 12

B

- bibliography 237
- book conventions xii
- broker
 - broker 16
 - service 26
- broker DLL name 17
- broker page 17
- broker setup page 18
- Broker_Exit function
 - OS/2 106
 - Windows 160
- Broker_GetCfgReqs function 104
- Broker_GetDllVersion function
 - OS/2 103
 - Windows 158
- Broker_GetVersion function
 - OS/2 103
 - Windows 158
- Broker_Init function
 - OS/2 105
 - Windows 159
- Broker_Logoff function
 - OS/2 108
 - Windows 162
- Broker_Logon function
 - OS/2 106
 - Windows 161
- Broker_SetupCfg function 109

- buffer size 28
- building block
 - AS/400 support 205
 - MQSeries support 181
- building the service broker DLL 164

C

- C header files
 - EXMP3CBR.H 102, 110
 - EXMP3CSE.H 117
 - EXMP3FRE.H 126
 - EXMW3CAL.H 157
- C language conventions 101
- C-language service requester API
 - return codes
 - call service function 127
 - retrieve error message 135
 - start broker function 130
 - start service function 133
 - stop broker function 132
 - stop service function 134
- calling a service function
 - C 126, 129, 177
 - REXX 137
- change font 15
- change status of an activity 39
- ChangeActivityState 39
- communication entry 210
- communication side information 210
- compatibility check
 - OS/2 103
 - broker 118
 - broker DLL 103
 - service DLL 118
 - Windows
 - broker 158
 - broker DLL 158
 - service DLL 166
- compiler 101
- concepts of integration 1
- configuration notebook
 - service 123
 - service broker 109
- configuration requirements
 - service 119

- configuration requirements (*continued*)
 - service broker 104
- considerations for VisualAge applications 59
- control remote
 - FlowMark for MVS/ESA from FlowMark on OS/2 or AIX 190
 - FlowMark on OS/2 or AIX from FlowMark for MVS/ESA 193
 - FlowMark on OS/2 or AIX from FlowMark on OS/2 or AIX 196
- controlling the Service Broker Manager 145
- conventions
 - syntax xiii
 - typographical xii
- Create function 45
- create the VisualAge runtime image 53
- CreateEncryptable function 46
- Current Activity
 - actions 68
 - events 72
 - general 68
- customizing MQSeries definitions 182, 206, 211

D

- data structures, predefined 197, 215
- DBCclose function 46
- DBOpen function 46
- debugging
 - service 145
 - service broker 145
- Decrypt function 47
- dedicated channels 209
- definitions, MQSeries 181, 205
- delete broker 20
- Delete function 47
- delete service 29
- detail level for messages 14
- details view 10
- display broker details 16
- display broker services 22
- display choices 10
- display service details 26
- DLL name
 - broker 17
 - service 27

E

- Encrypt function 47
- end FlowMark Program Access 217
- ENDFLMPAC 217
- error messages, retrieving
 - C 135
 - REXX 143
- exit function
 - service 120, 169
 - service broker 106, 160
- EXMP24BB.FDL 205
- EXMP24BB.MQI 205, 206
- EXMP24EU.SAV 205
- EXMP24RC 205, 211, 216
- EXMP24SD 205, 211, 215
- EXMP24SV 205, 219
- EXMP2ABB.DAT 184
- EXMP2ABB.MQI 182
- EXMP2ARM 200
- EXMP2ARV 201
- EXMP2ASD 198
- EXMP2ASP 200
- EXMP2ASV 202
- EXMP3CBR.H 228
- EXMP3CPR.EXE 227
- EXMP3CSE.H 228
- EXMP3CST.DLL 227
- EXMP3FBF.DLL 227
- EXMP3FBF.H 228
- EXMP3FBR.DLL 33
- EXMP3FFM.DLL 33, 227
- EXMP3FFR.EXE 56, 227
- EXMP3FMS.EXE 227
- EXMP3FRE.H 228
- EXMP3FRQ.DLL 33, 56, 57, 227
- EXMP3FRR.EXE 227
- EXMP3KBR.LIB 227
- EXMP3KRE.LIB 227
- EXMP3KSV.LIB 227
- EXMP3LBR.DLL 43, 227
- EXMP3LRQ.DLL 43, 227
- EXMP3LSE.DLL 43, 227
- EXMP3SFM.CMD 42
- EXMP3SRX.FDL 42
- EXMP3UAD.EXE 227
- EXMP3UCT.EXE 145, 227
- EXMP3UUP.EXE 227
- EXMP3UUS.EXE 227
- EXMP3VBR.DLL 55, 56

- EXMP3VND.EXE 57
- EXMP3VSM.FDL 54
- EXMP3VSM.IMG 53
- EXMP3VSM.INI 54
- EXMW3CAL.H 228
- EXMW3CAL.LIB 228
- EXMW3FBR.DLL 33, 228
- EXMW3FBR.H 228
- EXMW3FRC.EXE 228
- EXMW3FRQ.DLL 33
- EXMW3FRQ.H 228
- EXMW3FRU.DLL 228
- EXMW3FRU.EXE 228
- EXMW3FSE.DLL 33, 228
- EXMW3KRQ.LIB 228
- EXMW3LBR.DLL 43, 228
- EXMW3LRQ.DLL 43, 228
- EXMW3LSE.DLL 43, 228
- EXMW3LSR.H 228
- EXMW3USB.EXE 156, 228
- exported functions
 - service 116—125
 - service broker 102—110, 157

F

- FlowMark Activity
 - actions 80
 - attributes 80
 - events 81
 - general 79
 - settings 81
- FlowMark C container API 60
- FlowMark C process API 62
- FlowMark example process 54
- FlowMark Process
 - actions 83
 - attributes 82
 - events 85
 - general 82
 - settings 86
- FlowMark Program Access 217
- FlowMark Session
 - actions 78
 - events 79
 - general 78
- FlowMark type mapping 50
- FmApiBase 51
- FmApiLibrary 86

- FmArchivalCode 51
- FmBrokerSamples 53
- FmContainer 89
- FmContainerItem 92
- FmDisplayActivity 55, 98
- FmEditApiBase 51
- FmEditServiceBroker 51
- FmError 88
- FmExmApiBegin 94
- FmExmApiStructureData 96
- FmExmApiTypeInfo 95
- FmInputContainer 90
- FmMaintainContainer 55, 99
- FmMaintainData 55, 100
- FmMaintainProcess 55, 97
- FmOutputContainer 91
- FMRequest 57
- FmSamplesArchivalCode 53
- FmServiceBroker 51, 58
- FmStartDataItem 93
- Function Data
 - actions 65
 - events 65
 - general 64

I

- implementation
 - service 116, 165
 - service broker 2, 102, 157
 - service function 172
 - service requester 125, 177
- implementing application classes as visual parts 58
- indicate the end of the function 58
- initialization function
 - service 119, 168
 - service broker 105, 159
- initialize container parts 60
- initialize Current Activity part 60, 61
- Input Container
 - actions 74
 - attributes 73
 - events 74
 - general 73
 - settings 75
- installing the VisualAge programming examples 53
- instance
 - service 116
 - service broker 3

InvokeClass 55

L

LibMain function 163, 171

library files

EXMP3KBR.LIB 110

EXMP3KRE.LIB 126

load broker 20

load service 30

local queue 209

logoff function for service brokers 108, 162

logon function for service brokers 106, 161

logon procedure 8

Lotus Notes 43

broker 44

service functions

Create 45

CreateEncryptable 46

DBCclose 46

DBOpen 46

Decrypt 47

Delete 47

Encrypt 47

Read 47

SearchDoc 48

Sign 48

SignedOrEncrypted 49

Unsigned 49

UpdateEncryptable 50

M

manage

broker services 25

brokers 15

manual refresh 12

message

change font 15

detail level 14

monitor 13

message handling

EXMP24RC 217

EXMP24SD 216

EXMP24SV 220

EXMP2ARM 201

EXMP2ARV 202

EXMP2ASD 200

EXMP2ASV 203

message ID 197

migrating

OS/2 227

FlowMark service broker 227

Lotus Notes Service Broker Manager 227

Service Broker Manager 227

Windows 3.1 228

FlowMark service broker 228

Lotus Notes Service Broker Manager 228

Service Broker Manager 228

module definition file

service 117

service broker 102, 157

monitor Service Broker Manager 13

MQSeries

customizing definitions 182, 206, 211

definitions 181, 205

support 181

O

Output Container

actions 76

attributes 76

events 77

general 75

settings 77

P

predefined data structures 197, 215

process instance

restart 39

resume 37

start 36

suspend 37

terminate 38

process tracking 197

Process VisualAge Sample 54

process-instance ID 197

Program VisualAge Display Activity 54

Program VisualAge Maintain Container 54

Program VisualAge Maintain Data 54

programming examples, VisualAge broker 53

programming requirements 101

provided parts, VisualAge 64

Q

query data from the FlowMark input container 61
query the structure of the FlowMark container 62
queue
 local 209
 remote 209
 transmission 209

R

Read function 47
refresh
 automatic 12
 choices 12, 19
 interval 13, 24
 manual 12
remote queue 209
restart process instance 39
RestartProcess 39
resume suspended process instance 37
ResumeProcess 37
return codes
 RxSbrCallService function 138
 RxSbrDropFuncs function 137
 RxSbrGetErrorMessage function 143
 RxSbrLoadFuncs function 137
 RxSbrStartBroker function 139
 RxSbrStartService function 141
 RxSbrStopBroker function 140
 RxSbrStopService function 142
 SbrCallService function 127
 SbrGetErrorMessage function 135
 SbrStartBroker function 130
 SbrStartService function 133
 SbrStopBroker function 132
 SbrStopService function 134
 service function 125
return data
 EXMP24RC 216
 EXMP24SD 216
 EXMP24SV 219
 EXMP2ARM 201
 EXMP2ARV 202
 EXMP2ASD 199
 EXMP2ASP 200
 EXMP2ASV 203
REXX language service requester API
 return codes
 call service function 138
 load API functions 137

REXX language service requester API (continued)

 return codes (continued)
 retrieve error message function 143
 start broker function 139
 start service function 141
 stop broker function 140
 stop service function 142
 unload API functions 137

 routing entry 210
 RxSbrCallService function 137
 RxSbrDropFuncs function 137
 RxSbrGetErrorMessage function 143
 RxSbrLoadFuncs function 136
 RxSbrStartBroker function 139
 RxSbrStartService function 141
 RxSbrStopBroker function 140
 RxSbrStopService function 142

S

sample files
 EXMP3SRX.CMD 152
 SAMPBROK.C 102, 146
 SAMPREQ.C 126, 150
 SAMPSEV.C 148
 SAMPSRVC.C 117
sample process
 FMMQI_AIX_AIX 187
 FMMQI_AIX_LOCAL 187
 FMMQI_AIX_MVS 187
 FMMQI_AIX_OS2 187
 FMMQI_CONTROL_AIX_AIX 187
 FMMQI_CONTROL_AIX_LOCAL 187
 FMMQI_CONTROL_AIX_MVS 187
 FMMQI_CONTROL_AIX_OS2 187
 FMMQI_CONTROL_OS2_AIX 186
 FMMQI_CONTROL_OS2_LOCAL 186
 FMMQI_CONTROL_OS2_MVS 186
 FMMQI_CONTROL_OS2_OS2 186
 FMMQI_OS2_AIX 186
 FMMQI_OS2_LOCAL 186
 FMMQI_OS2_MVS 186
 FMMQI_OS2_OS2 186
sample scenario 213
SbbDisableTimeout function 115
SBBFM.DLL 227
SBBFM.H 228
SBBFMSB.EXE 227
SBBLNTS.DLL 227

- SbbLog function 113
- SbbQueryLogLevel function 114
- SbbReadProfile function 112
- SBBROKER.H 228
- SBBROKER.LIB 227
- SBBSTD.DLL 227
- SbbWriteProfile function 111
- SBCNTRL.EXE 227
- SBM.EXE 227
- SBMADMIN.EXE 227
- SBMUSER.EXE 227
- SBMWIN.EXE 228
- SBPREDIT.EXE 227
- SbrCallService function 126, 177
- SbrCallServiceWithRetry function 129
- SBREQ.EXE 227
- SBREQ.H 228
- SBREQ.LIB 227
- SBREQFM.DLL 227
- SBREQFM.EXE 227
- SBREQLN.DLL 227
- SbrGetErrorMessage function 135
- SbrStartBroker function 129
- SbrStartBrokerWithInfo function 131
- SbrStartService function 133
- SbrStopBroker function 131
- SbrStopService function 134
- SBSERVIC.H 228
- SBSERVIC.LIB 227
- SBSFM.DLL 227
- SBSLNTS.DLL 227
- SBWBFM.DLL 228
- SBWBRLN.DLL 228
- SBWBROK.H 228
- SBWBROK.LIB 228
- SBWFM.H 228
- SBWREQ.H 228
- SBWREQ.LIB 228
- SBWREQC.EXE 228
- SBWRFM.DLL 228
- SBWRFM.EXE 228
- SBWSERV.H 228
- SBWSFM.DLL 228
- SBWSVLN.DLL 228
- SearchDoc function 48
- separately accessible function parts 58
- service
 - configuration requirements 119
 - controlling 145
 - debugging 145
- service (*continued*)
 - description 116
 - exported functions 116—125
 - implementation 116, 165
 - instance 116
 - module definition file 117
 - sample 148
 - testing 144, 180
- service broker
 - configuration data 110
 - configuration requirements 104
 - controlling 145
 - debugging 145
 - demo 42
 - description 2
 - exported functions 102—110, 157
 - functions 103
 - implementation 102, 157
 - instance 3
 - module definition file 102, 157
 - sample 146
 - testing 144, 180
 - version information 103, 118, 158, 167
- service broker API
 - description 110
 - SbbDisableTimeout 115
 - SbbLog 113
 - SbbQueryLogLevel 114
 - SbbReadProfile 112
 - SbbWriteProfile 111
- service broker concept
 - design 3
 - overview 1
- service broker library
 - FlowMark 33
 - Lotus Notes 43
- Service Broker Manager
 - differences between Windows and OS/2 155
 - OS/2 7
 - Windows 155
- service functions
 - calling 125, 126, 137, 177
 - calling with retry 129
 - implementation 172
 - OS/2 124
 - return codes 125
 - Windows 172
- service registration
 - service page 27

- service requester
 - API 126
 - error messages 135, 143
 - implementation 177
 - implementing 125
 - samples 150, 152
 - starting
 - service 133
 - service broker 129
 - service broker with additional information 131
 - stopping
 - service 134
 - service broker 131
- service requester API
 - C 126
 - description 126
 - REXX 136
 - RxSbrCallService 137
 - RxSbrDropFuncs 137
 - RxSbrGetErrorMessage 143
 - RxSbrLoadFuncs 136
 - RxSbrStartBroker 139
 - RxSbrStartService 141
 - RxSbrStopBroker 140
 - RxSbrStopService 142
 - SbrCallService 126, 177
 - SbrCallServiceWithRetry 129
 - SbrGetErrorMessage 135
 - SbrStartBroker 129
 - SbrStartBrokerWithInfo 131
 - SbrStartService 133
 - SbrStopBroker 131
 - SbrStopService 134
- Service_CheckBroker function
 - OS/2 118
 - Windows 167
- Service_Exit function
 - OS/2 120
 - Windows 169
- Service_GetCfgReqs function 119
- Service_GetDllVersion function
 - OS/2 118
 - Windows 166
- Service_Init function
 - OS/2 119
 - Windows 168
- Service_SetupCfg function 123
- Service_Start function
 - OS/2 121
 - Windows 169

- Service_Stop function
 - OS/2 122
 - Windows 170
- Sign function 48
- SignedOrEncrypted function 49
- SNDFLMRQS 220
- stack size 28
- standard
 - broker 29
 - external controller 145
 - service requester 144, 180
- start
 - broker 21, 156
 - function for services 121, 169
 - process instance 36
 - service 30, 156
 - Service Broker Manager 8, 156
 - services dynamically 31
- start external process-control session 62
- start FlowMark Program Access 217
- start remote
 - FlowMark for MVS/ESA from FlowMark on OS/2 or AIX 188
 - FlowMark on OS/2 or AIX from FlowMark for MVS/ESA 191
 - FlowMark on OS/2 or AIX from FlowMark on OS/2 or AIX 194
- start service broker at application startup 58
- StartProcess 36
- startup status 19, 27
- status, change 39
- stop
 - broker 21, 156
 - function for services 122, 170
 - service 31, 156
- STRFLMPAC 217
- suspend running process instance 37
- SuspendProcess 37
- synchronization of resources 3
- synchronizing requests 28
- syntax conventions xiii

T

- terminate running process instance 38
- TerminateProcess 38
- testing service brokers and services 144, 180
- tracking of process 197
- transmission queue 209

tree view 10
typographical conventions xii

U

unique process-instance ID 197
unload broker 20
unload service 30
Unsigned function 49
update data in the FlowMark output container 62
UpdateEncryptable function 50
using a FlowMark Session part 62
using the FlowMark requester 180

V

version numbers, returning 103, 118, 158, 166
VisualAge 51
VisualAge applications for FlowMark
 considerations 59
 FlowMark definitions 57
 registering 56
 requirements 58
 Service Broker Manager definitions 56
 testing 63
VisualAge example code 55
VisualAge parts 64
 FlowMark C language API 52
 service broker 51
 support parts 52
 used in the composition editor 52

W

Windows 155

Your comments, please ...

**IBM FlowMark
Application Integration Guide
Version 2 Release 3
Publication No. SH12-6267-01**

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

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.

If you mail this form to us, be sure to print your name and address below if you would like a reply.

You can also send us your comments using:

- A fax machine. The number is: +49-7031-166609.
- Internet. The address is: gadlid@sdfvm1.vnet.ibm.com.
- IBMLink. The address is: SDFVM1(GADLID).
- IBM Mail Exchange. The address is: DEIBM3P3 at IBMMAIL.

Please include the title and publication number (as shown above) in your reply.

Name

Address

Company or Organization

Phone No.

Your comments, please ...
SH12-6267-01



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, Dept. 0446
Postfach 1380
71003 Boeblingen
Germany

Fold and Tape

Please do not staple

Fold and Tape

SH12-6267-01

Cut or Fold
Along Line



Part Number: 83H1985
Program Number: 5697-216

Distributed electronically for customer printing