Store Integration Framework
Data Integration Facility

Programming Guide

August 7, 2003

GA27-4309-01

Store Integration Framework
Data Integration Facility

Programming Guide

August 7, 2003

GA27-4309-01

Contents

Figures

Chapter 1. Overview

The Data Integration Facility (DIF) of the IBM Store Integration Framework consists of a programmer's API and a runtime process. These tools enable the following kinds of messaging:

- Intra-store "point-to-point" communication between applications
  - Applications can exchange arbitrary data
  - One application can query another and receive a response
- Once-only assured delivery of data from the store to the enterprise
  - Send real-time sales and inventory data
  - Adjust the rate of data flow to match network performance and capacity
- Respond to queries from remote systems, such as enterprise requests for store-based data
- Enable legacy terminals and other in-store devices to communicate using XML, SOAP, or Websphere MQ-based messaging

DIF "opens up the store" by providing Java and non-Java applications (including CBASIC applications) easy-to-use interfaces for:

- Web services via HTTP
- Message-oriented-middleware (Websphere MQ, MQe, and other JMS-enabled systems)
- Other applications that use the DI facility

Finally, DIF provides out-of-the-box support for transforming 4690 sales data into standards-based XML. For example, transaction data (tlogs) can be converted from native binary format to IXRetail-standard POSLog XML documents. Transactions can "trickle" in real-time from the store to host systems.

Technologies

The Data Integration Facility of the Store Integration Framework uses many open source technologies and standards as well as IBM software. The following table serves as a quick reference for some of these tools:

Technology	Use	More information
Java	Platform for DIF tools	`http://java.sun.com`
Websphere MQSeries Everyplace (MQe)	Infrastructure for non-persistent and persistent messaging	`http://www-3.ibm.com/software/integration/appconn/wmqe/`
Java Message Service (JMS)	Standard Java API for messaging; used to support MQSeries-based message transfer	`http://java.sun.com/products/jms`
SOAP	Standard Message Definition	`http://www.w3.org/TR/SOAP`
SOAP-SAAJ	Standard Java API for SOAP messaging	`http://java.sun.com/xml/saaj/index.html`
IXRetail	Standard XML Definitions for Retail	`http://www.ixretail.org`

Technology	Use	More information
XSLT	Standard data mapping technology	http://www.w3.org/TR/xslt
Apache Xerces, Xalan	Java Tools for XML parsing and processing	http://xml.apache.org
Apache Commons Logging, Log4J	Standard Java API for logging	http://www.jakarta.org

Data Integration Facility Runtime

The DIF Runtime is a Java-based background application that serves as a messaging hub for the store. It allows non-Java applications, such as CBASIC or C applications, to communicate with message systems like Websphere MQ or with web servers using SOAP messages. Examples include:

- Integrating 4690 sales support/checkout support with Websphere MQ or MQSeries Everyplace for trickling transaction data in real-time
- Allowing a CBASIC terminal application to send requests to the DIF runtime over a 4690 pipe, which the DIF runtime converts to XML or SOAP for communication with an application server

The DIF Runtime also allows external applications to communicate with the store. For example, host systems can send queries to the store to retrieve store-based data, such as data in 4690 keyed files.

The DIF Runtime is extendable because it can execute any number of Services and Actors. Services and Actors are pluggable pieces of Java code that perform specific tasks. The Data Integration Facility provides default Services and Actors for typical store-based messaging, including support for 4690 pipe messaging, TCP/IP-based socket messaging, JMS/MQ/MQe messaging, SOAP messaging, and transaction log transformation into XML.

Users may write custom services and actors to extend the capabilities of the DIF runtime.

Service
A *Service* is a pluggable component that represents an “execution thread” within the DIF runtime. A Service is always active, waiting to perform work. A Service usually acts as an entry point for messages that require processing by DIF. For example, the DIF package contains services for receiving messages sent over 4690 pipes or TCP/IP sockets. Services also exist for retrieving messages from messaging systems such as Websphere MQ. In each of these cases, the Service encapsulates one or more “execution threads” that perform work.

Actor An *Actor* is a pluggable component that processes messages; Actors are akin to “functions”. Every Actor is capable of producing an output message for any given input message. One or more Actors can be linked together to perform multiple tasks on a single message. Typically, a Service is associated with one or more Actors that perform work on a message. This way, a Service is responsible for managing the execution thread of a message, but Actors are plugged in as necessary to perform customized work. This allows DIF customers to modify existing Actors for their own use, or they may build completely new Actors.

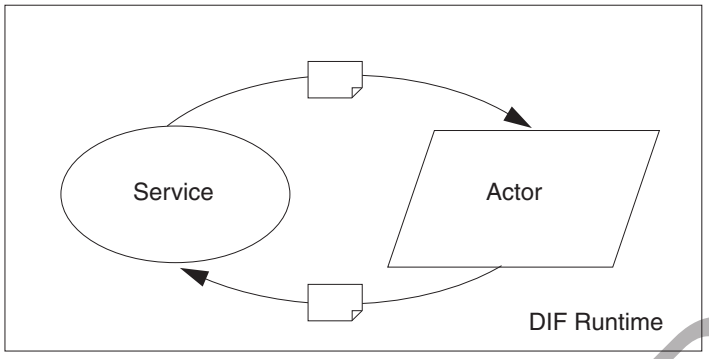


Figure 1-1. Flow of message processing

Multiple services can be started at once within the DIF Runtime. For services that interact with external applications via sockets or pipes, the diagram adds one or more *clients*:

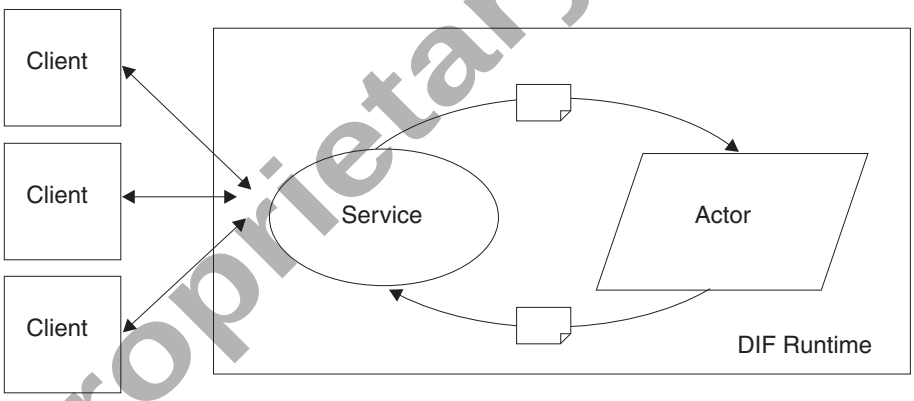


Figure 1-2. Flow of message processing with clients

Finally, in order for Services to pass a message to more than one Actor for processing, a special Actor called the *Director* must be used. The Director can "chain" together multiple Actors to build a more complicated result. The Director allows Actors to specialize on a particular task so that the Actor can be reused in other scenarios.

The Director changes the diagram as follows:

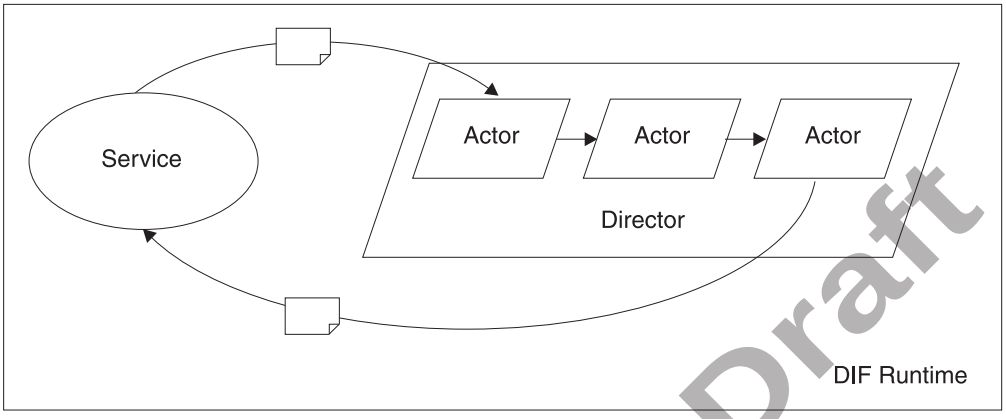


Figure 1-3. Flow of message processing with Director

The Director routes inbound messages to one or more Actors in a sequence. The routing is based on a property of the message called the *message profile Id.* The profile of the message determines the sequence of Actors that will process it.

The Director is responsible for handing the message to each Actor in the sequence. When the first Actor returns its response, the response is fed as input to the next Actor, and so on. The response from the final Actor is returned back to the client that sent the request. Customers can add their own Actors to perform customized processing on a message.

Since the Director allows multiple Actors to be combined in a message flow, powerful processing can be performed on inbound messages prior to delivery to messaging systems or web services.

Message formats

The DIF Runtime and API expect messages that are either raw byte streams or SOAP messages. In either case, a message always has a header that contains zero or more property-value pairs. This allows contextual information to be passed along with the payload of the message. If SOAP messaging is used, the headers are MIME headers implemented by the javax.xml.soap.MimeHeaders class; otherwise, for byte stream messages, the headers are implemented with the DIF MessageHeaders class.

On output, the standard DIF Actors can write pure byte streams or SOAP messages. SOAP messages may be written over HTTP or may be embedded within JMS or Websphere MQe messages.

Configuration of the DIF runtime

The DIF Runtime is configured through a plain-text properties file that contains a series of property/value pairs. These properties determine:

- which Services are executed
- which Actor is associated with each Service
- the sequence of Actors required to process a given message profile id
- connection parameters for remote JMS queue managers
- local parsing and transformation files for 4690 transaction processing
- target directories for error details and problematic messages

Each Actor and Service may have a collection of options that can be configured. For the Actors and Services that are packaged with DIF, these options are detailed in Appendix A, “Standard Actors & Services”.

Chapter 2. Configuration and operation

System requirements

The Java 2 Standard Runtime Environment (J2SE 1.3.x or later) is required. Memory requirements vary by platform and depend on the Services and Actors configured for execution.

- IBM 4690OS
  - 16MB free memory required for DIF Runtime (no conversion)
  - Minimum CPU: *to be determined*
  - Disk space: *to be determined*

Configuring the Data Integration Facility Runtime

The DIF Runtime is configured through the properties file `difsrvc.pro`. Users cannot modify this file. Instead, the custom property values should be added to `difuser.pro`.

The following configuration steps should be performed:

1. Establish the Services to be started by DIF.
2. Associate an Actor with each Service, as required by the Service.
3. Configure detailed options for each Service and Actor.
4. Configure JMS or MQSeries Everyplace connectivity, if required.

1. Establish the Services to be started by DIF

Each Service must be given a unique, descriptive name that identifies the Service throughout the properties file. The name can contain any characters except for whitespace and is case-sensitive. The Services are declared by the following property:

```
container.services=ServiceId1 ServiceId2 ServiceId3 ...
```

This property establishes the list of Services that are started by the DIF Runtime. It can appear anywhere in the properties file. Note that the ServiceId names are not necessarily Java class names; they can be any valid identifier that describes the Service.

The Java class that implements the Service is given by the uri property, which is required for any ServiceId listed in `container.services` and can also appear anywhere in the properties file:

`ServiceId.uri=`*`valid-uri`*

For example, if a Service called DiskQService was defined in `container.services`, the following property would be required:

`DiskQService.uri=`*`valid-uri`*

URI is a Uniform Resource Identifier. Some Services define special URIs that embed configuration information; most URIs are simply the name of a local Java class that implements the Service.

To specify that a named Service is implemented by a specific Java class, use the `local:` URI notation. For example:

```
MySpecialService.uri=local:com.mycorp.dif.MySpecialService
```

The `local:` prefix indicates that the specified Java class can be found on the local CLASSPATH, and that the class implements the DIF `Service` interface (ie., it is a valid Service). Customers who write their own Services should place them in a file called `difuser.jar`, which is always in the CLASSPATH.

Services that do not use the `local:` prefix are described in Appendix A, “Standard Actors & Services”.

2. Associate an Actor with each Service, as required by the Service

Once each Service has been identified and its URI established, the Service should be associated with a specific Actor. In some cases, Services do not require an Actor; for those Services, this step can be skipped.

For most Services, however, an Actor must be specified. This is done with the `listener` property:

```
ServiceId.listener=ActorId
```

Note that the Service’s unique Id must prefix the listener property. Also note that the value of this property is a unique Actor identifier. This identifier has the same rules as a Service identifier; any characters except for whitespace are allowed, and the name is case-sensitive. Likewise, the name is not necessarily a Java classname; it can be any descriptive name.

However, the ActorId must be associated with a URI somewhere else in the properties file:

```
ActorId.uri=valid-uri
```

Just as with a Service, the Actor’s URI is an identifier that indicates which Java class to associate with the Actor. This is commonly done with the `local:` prefix:

```
MySpecialActor.uri=local:com.mycorp.dif.MySpecialActor
```

The `local:` prefix indicates that the specified Java class can be found on the local CLASSPATH, and that the class implements the DIF `Actor` interface (ie., it is a valid Actor). Customers who write their own Actors should place them in a file called `difuser.jar`, which is always in the CLASSPATH.

3. Configure detailed options for each Service and Actor

Each Actor and Service can be customized by setting additional properties. See Appendix A, “Standard Actors & Services” for configuration options for each of the pre-packaged Services and Actors.

In general, Services and Actors require their configuration properties to be prefixed with the unique Id assigned to them in the previous steps. For example, if an Actor requires a property called `workDirectory`, use the following complete property name:

```
ActorId.workDirectory=C:/tmp
```

4. Configure JMS or MQSeries Everyplace connectivity

Messaging systems are typically configured independently from Services and Actors. This section describes how to configure the most common messaging systems: Websphere MQSeries Everyplace and Websphere MQSeries JMS.

Websphere MQSeries Everyplace

For MQe-based messaging, the following properties are available or required:

Key	Description	Valid values
`mqe.ini`	Required. Specifies the complete path of the INI file of the MQe queue manager to use. Only one MQe queue manager can be used for a single instance of the DIF Runtime.	Any valid filepath for the local filesystem. The default is: `M:/MQe/StoreQM.ini`
`mqe.interface.uri`	Optional. Indicates whether the MQe queue manager should be executed in Server or Client mode. By default, Server mode is chosen.	• server • client

Websphere MQSeries JMS

For WMQ JMS-based messaging, more than one remote queue manager can be configured. Each queue manager is defined with a JMS profile, where each profile has its own unique Id. For Websphere MQ, each profile may have the following properties; substitute a unique profile Id where indicated:

Key	Description	Valid values
`jms.`*`profileId`*`.hostname`	Required. Specifies the remote hostname or IP address of the machine on which the queue manager is running.	Any valid hostname or IP address
`jms.profileId.qmgr`	Required. Specifies the name of the remote queue manager to connect to for this profile.	Any valid queue manager name residing on the machine given by the hostname property
`jms.profileId.channel`	Required. Specifies the name of the MQSeries channel definition to use when connecting as a client.	Any valid “server connection” channel established for the named queue manager
`jms.profileId.port`	Optional. Specifies the port on which the named queue manager is being served; if not given, 1414 is assumed.	Any valid port number

Configuring the Director and Message Profile Ids

The Director is configured like any other Actor. It is conventionally given the unique name `Director` and therefore requires the following entry in the configuration properties file `difsrvc.pro`:

```
Director.uri=local:com.ibm.retail.di.services.director.Director
```

The *uri* of the Director must be specified as shown.

The Director is capable of passing messages from one Actor to another. This process is called a *message flow*. In a message flow, the output from one Actor is fed to the next Actor as input. The output of the final Actor in the sequence represents the response returned by the Director.

For maximum flexibility, most configured Services should associate themselves with the Director; for example:

```
ServiceId.listener=Director
```

Any messages inbound to the Service (or produced by the Service) will be passed to the Director for processing. Remember that the Director itself does not perform any processing; it depends on a sequence of one or more Actors.

The Director defines one or more message flows. Each message flow is uniquely associated with a specific *message profile Id*. A message profile Id is a name that is associated with a type of message. These names are arbitrary and are assigned by users of DIF. When a message is created, a message profile Id is associated with that message. This Id is used by the Director to determine which message flow to run in order to process the message.

A message flow is defined by listing a sequence of Actor names as defined in the properties file. For example:

```
Director.MessageProfileId.actors=ActorId-1 ActorId-2 ActorId-3 ... ActorId-n
```

Each *ActorId* reference must have a corresponding *uri* property, as described previously:

```
ActorId.uri= valid-uri
```

Note: Any required properties for the configured Actor must also be defined. These properties vary by Actor; please see Appendix A, “Standard Actors & Services” for details.

When the Director receives a message, it consults the configuration properties for the actors list corresponding to the message profile Id of the message. If the actors list is not defined, the message processing is aborted with a *fault*. (See “Actors, the Director, and faults”.)

Processing proceeds as follows:

1. The message is handed to *ActorId-1*.
2. If no fault occurs, the output from *ActorId-1* is passed to *ActorId-2*.
3. This process continues until *ActorId-n*.
4. The output of *ActorId-n* is returned by the Director.

Actors, the Director, and faults

If an Actor experiences an error while processing a message, it generates a *fault*. A fault describes the error that occurred and is attached to the affected message.

When a fault occurs, processing of the message normally stops. If a message is being processed under the guidance of the Director, no further Actors are called. The message and its fault are returned immediately. The calling Service or client can interrogate the message for detailed fault information.

Additionally, the Director can write fault information to disk into a special `faults` directory. This feature must be enabled by adding properties to `difuser.pro`:

```
Director.faults.record=true | false
Director.faults.directory= valid-local-directory
```

When enabled, the Director will write three files for each faulty message. The files have the following extensions:

.req Contains the unaltered content of the message body

.hdr Contains all message header properties from the message

.err Contains the complete error text and stack trace

The base names of the three files are identical; the extensions distinguish the content. The base name is the message profile ID of the message followed by a unique number. (The unique number is the assurance ID of the message, if available; otherwise, a unique number is generated). The filename has this format:

```
{MessageProfileId}{AssuranceId} or {MessageProfileId}NA{UniqueId}
```

Note: If files are written to the C:\ drive of a 4690 machine, the MessageProfileId is dropped and the numeric portion of the filename is truncated to eight digits if necessary.

Optionally, users may specify that the Director should *ignore* faults. When the Director ignores faults, it stops processing the current message. The message is immediately returned. However, it will not include fault information. The message will appear to the calling Service as if no fault occurred.

When "ignore faults" is enabled, fault information will still be written to disk by the Director as long as `Director.faults.record` is **true**. This feature can be useful for testing; it allows many messages to flow through the system, with all faults optionally recorded but not allowing those faults to suspend processing. To ignore faults, use the following property of the configuration file:

```
Director.faults.ignore=true | false
```

Logging Configuration

Logging is provided by the Log4J facility. This is configured via the `DIFLOG4J.PRO` file. The top of this file lists the level of logging and the desired outputs:

```
log4j.rootLogger=[debug | info] {, stdout} {, file}
```

Choose one of debug or info to determine the level of logging. Debug mode should be used sparingly, as it produces a tremendous amount of output and significantly slows processing.

Any combination of outputs can be specified; by default, two options are offered: stdout and file. The stdout option forces all logging output to the screen; the file option writes all logging output to a local file.

Note: If stdout is specified and the DIF Runtime is running as a background task, no output is seen on the screen.

Logging output can be customized by setting the following properties, all of which must be prefixed by `log4j.appender`:

Key	Description	Valid values
`stdout.layout.ConversionPattern`	Required. Specifies the format of log events written to the screen.	*

Key	Description	Valid values
file.File	Required. Specifies the complete path of the local logging file.	Any valid local filename
file.MaxFileSize	Required. Determines the maximum size, in kilobytes, of the logging file. The file will wrap itself or spill into another file if this size is exceeded.	Example: 100KB **Note:** "KB" must be used as a suffix
file.MaxBackupIndex	Required. Indicates the total number of log files to be saved other than the primary log file. When the primary log file is full, it will be backed up if this value is greater than zero. The backup file will have the same name as the original, with an integer suffix appended.	Non-negative integer
file.ConversionPattern	Required. Specifies the format of log events written to the screen.	*

* For available tokens for this pattern, refer to:
http://jakarta.apache.org/log4j/docs/api/org/apache/log4j/PatternLayout.html

Starting the Data Integration Facility Runtime

Start the task by executing the \adx_ipgm\difsrvc.bat file.

Note: The DIF Runtime can be run as a background task on a 4690 controller. Specify the complete path to \adx_ipgm\difsrvc.bat as the Program Name, and do not list any parameters.

Chapter 3. Developing custom extensions

Note: This chapter assumes that the reader is an experienced Java programmer.

Users can create custom Actors and Services for use with the DIF Runtime. These Actors and Services are specified in `difuser.pro` by associating the *uri* property with a custom Java class.

For example:

```
MySpecialService.uri=local:com.mycorp.dif.MySpecialService
MySpecialActor.uri=local:com.mycorp.dif.MySpecialActor
```

When providing a custom Actor or Service, the Java class must implement the corresponding DIF Java interface:

- An Actor must implement the `com.ibm.retail.di.xml.messaging.service.Actor` interface.
- A Service must implement the `com.ibm.retail.di.net.service.Service` interface.

Writing a Service

A DIF Service must define the following methods, which are described in detailed in the javadoc provided with the product.

The `com.ibm.retail.di.net.service.Service` interface combines these interfaces:

- `com.ibm.retail.di.naming.Named`
- `com.ibm.retail.di.net.service.Initializable`
- `com.ibm.retail.di.net.service.Startable`

The required methods are:

```
public void init(Context context) throws RetailException;
public void close();
public boolean isInitialized();
public boolean isClosed();

public void start() throws RetailException;
public void stop();
public boolean isStopping();
public boolean isActive();

public String getName();
public void setName(String aName);
```

The implementation for these methods will be similar from one Service to another, so the DIF API provides a common, tested implementation in class `com.ibm.retail.di.net.service.AbstractService`.

AbstractService

(`com.ibm.retail.di.net.service.AbstractService`)

`AbstractService` provides implementations for the lifecycle methods required by the Service interface. An `AbstractService` has these features:

- It is not associated with an Actor by default.
- It does not read any Context keys by default.

- It does not start any internal threads by default.
- It is the most trivial starting point for a user-defined Service.

Subclasses need only add their own specific background function, and must define these abstract methods, all of which must return immediately:

doInit()
Reads configuration keys, determines viability of the service

doStart()
Starts any background processing of the service

doStop()
Stops any background processing of the service

Notes:

1. Do not perform long-running operations within any of the above methods. These methods are called on the main DIF Runtime thread and any blockage of this thread will affect all Services.
2. If any of these methods throws an exception, the DIF Runtime will stop all Services and will terminate.

The following example is a `MessageService` that simply outputs a message to standard output. The service remains in the started/active state until asked to stop. The message is a required key retrieved from the Context (which is normally represented by `DIFUSER.PRO`).

```
package examples.services;

import com.ibm.retail.di.net.service.AbstractService;
import com.ibm.retail.di.naming.Context;

public class MessageService extends AbstractService
{
    private String message;

    protected void doInit() throws Exception
    {
        message = getContextString ("message", Context.REQUIRED);
    }

    protected void doStart() throws Exception
    {
        System.out.println (message);
    }

    protected void doStop() throws Exception
    {
        ; // do nothing
    }
}
```

And here is a sample configuration for `MessageService` as would appear in the services properties file:

```
container.services=MyMessageService

MyMessageService.uri=local:examples.services.MessageService
MyMessageService.message=Hello, world!
```

In general, when extending `AbstractService`, the following techniques should be applied:

- Verify dependencies and retrieve required properties from the Context in `doInit()`, throwing an exception if necessary.
- Since `doStart()` *must* return immediately, any background processing performed by the Service must be done on its own Thread; such a Thread should be started within `doStart()`.
- In `doStop()`, be sure to stop any spawned threads and interrupt any blocking I/O.

The `AbstractService` class provides a collection of utility methods for extracting values from the Context: `getContextBoolean()`, `getContextInt()`, `getContextLong()`, and `getContextString()`. Some of these methods throw an exception automatically if the requested key is absent but is declared a *required* key.

AbstractInteractiveService

(`com.ibm.retail.di.net.service.interactive.AbstractInteractiveService`)

An interactive Service is a Service that has an associated Actor (called a *listener* in many of the API methods) that performs processing on behalf of the Service. `AbstractInteractiveService` extends `AbstractService`.

An `AbstractInteractiveService` has the following features:

- It is associated with an Actor by default.
- It retrieves its Actor from the Context by looking for a key called `InteractiveListener.INTERACTIVE_LISTENER_CONTEXT_KEY`, which must be associated with an instance of the `InteractiveListener` interface. (All Actors are instances of this interface.) By default, the DIF Runtime automatically instantiates the Actor associated with this Service and sets this key in the Context, so developers do not normally need to refer to this key.
- It does not start any internal threads by default.
- It establishes a SOAP `MessageFactory` so that the Service may generate SOAP messages if necessary. The factory can be retrieved by calling `getMessageFactory()`.
- It is the most trivial starting point for a user-defined Service that must be associated with an Actor.

Here is a variation of the `MessageService` example. In this version, a SOAPMessage is generated and passed to the listener/Actor associated with the Service.

```
package examples.services;

import com.ibm.retail.di.net.service.AbstractInteractiveService;
import com.ibm.retail.di.naming.Context;
import com.ibm.retail.di.xml.messaging.soap.SoapUtil;

public class InteractiveMessageService extends AbstractInteractiveService
{
    private String text;
    private String msgProfId;

    protected void doInit() throws Exception
    {
        super.doInit();
        text = getContextString ("text", Context.REQUIRED);
   aMsgProfId = getContextString ("mpid", Context.REQUIRED);
    }

    protected void doStart() throws Exception
    {
```

```
        SOAPMessage msg = SoapUtil.createSoapMessage (
            getMessageFactory(),
            msgProfId,
            text);

        doInteraction (msg); // pass to Actor (once, then done)
    }

    protected void doStop() throws Exception
    {
        ; // do nothing
    }
}
```

And here is a sample configuration for `InteractiveMessageService` as would appear in the services properties file:

```
container.services=MyMessageService

MyMessageService.uri=local:examples.services.InteractiveMessageService
MyMessageService.text=Hello, world!
MyMessageService.mpid=HelloMessageType
MyMessageService.listener=MyFavoriteActor

# Sample Actor Configuration
MyFavoriteActor.uri=local:examples.actors.MyFavoriteActor
```

Note that this service calls its Actor once, passing the generated SOAP message whose attachment is a piece of text retrieved from the Context. The Actor is only called once because the Service is started once by the DIF Runtime.

In general, when extending `AbstractInteractiveService`, the following techniques should be applied:

- If overriding `doInit()`, care must be taken to first call `super.doInit()` in order to preserve the base functionality of this class.
- To indicate that an associated Actor is not required for this service, override `isListenerRequired()` and return `false`.
- To create a specialized SOAP message factory, override `createMessageFactory()`.
- To ask the base class to pass a message to the associated Actor, call `doInteraction()`.

RunnableService

(`com.ibm.retail.di.net.service.interactive.RunnableService`)

A *runnable* Service is a Service that has an associated Actor (called a *listener* in many of the API methods) and also has a background thread that performs a task. Often, the background thread generates or receives messages and passes those messages to the associated Actor. `RunnableService` extends `AbstractInteractiveService`.

A `RunnableService` has these features:

- It is associated with an Actor by default.
- It starts a single “work” thread by default, and subclasses must only define the run() method of the thread.
- It establishes a SOAP `MessageFactory` so that the Service may generate SOAP messages if necessary. The factory can be retrieved by calling `getMessageFactory()`.

- It is a good starting point for a user-defined Service that would like to perform work on a background thread, optionally calling an associated Actor.

Here is a final variation of the MessageService example. In this version, a SOAPMessage is generated and passed to the listener/Actor associated with the Service. The message is passed at a regular interval defined by the Context.

```
package examples.services;

import com.ibm.retail.di.net.service.AbstractService;
import com.ibm.retail.di.naming.Context;
import com.ibm.retail.di.xml.messaging.soap.SoapUtil;

public class InteractiveMessageService extends RunnableService
{
    private String text;
    private String msgProfId;
    private int interval;

    protected void doInit() throws Exception
    {
        super.doInit();
        text = getContextString ("text", Context.REQUIRED);
   aMsgProfId = getContextString ("mpid", Context.REQUIRED);
        interval = getContextInt ("interval", 5000);
    }

    public void run() throws Exception
    {
        while (!isStopping()) {

            SOAPMessage msg = SoapUtil.createSoapMessage (
                getMessageFactory(),
                msgProfId,
                text);

            doInteraction (msg); // pass to Actor

            pause (interval);
        }
    } // service terminates here, DIF Runtime also terminates

    // note: the base class provides implementations of
    // doStart() and doStop(), so we do not need to provide
    // those here
}
```

Note that the service retrieves an additional item from the Context: an interval representing the time (in milliseconds) to wait between sending each new message object. A default of 5000 is established if the key does not exist in the Context.

The base RunnableService class defines doStart() and doStop(), so these do not need to be implemented unless additional function is required. The base class takes care of setting up the thread and destroying it when the service stops.

In general, when extending RunnableService, the following techniques should be applied:

- If overriding doInit(), doStart(), or doStop(), care must be taken to first call the super implementation in order to preserve the base functionality of this class. Failure to do this will cause the thread to not be started or stopped correctly!
- When the run() method ends, the service is automatically stopped, which forces the DIF Runtime to terminate. Do not allow run() to terminate unless there is a problem with the service, or unless isStopping() is **true**.

Note: Failure to consult `isStopping()` in a `run()` loop will likely result in improper shutdown or hangs of the DIF Runtime.

- Do not perform unlimited blocking operations within `run()`. Always specify a timeout on a blocking operation so that a prompt shutdown of the service can be accomplished.
- To pause within `run()`, use the `pause()` method. It automatically interrupts itself if a `stop` request is received.
- If necessary, the subclass can create the work Thread by overriding `createThread()`.
- If an associated Actor is not required by a subclass of `RunnableService`, override `isListenerRequired()` to return **false**.

Writing an Actor

A DIF Actor must define the following methods, which are described in detailed in the javadoc provided with the product.

The `com.ibm.retail.di.xml.messaging.Actor` interface combines these interfaces:

- `com.ibm.retail.di.xml.Named`
- `com.ibm.retail.di.net.service.Initializable`
- `com.ibm.retail.di.net.service.interactive.InteractiveListener`
- `com.ibm.retail.di.xml.messaging.soap.ReqRespListener`

The required methods are:

```
public void init(Context context) throws RetailException;
public void close();
public boolean isInitialized();
public boolean isClosed();

public String getName();
public void setName(String aName);

public Object onMessage(MessageHeaders theRequestHeaders,
                        Object theRequest,
                        MessageHeaders theResponseHeaders);

public SOAPMessage onMessage(SOAPMessage aMessage);
```

The implementation for most of these methods will be similar from one Actor to another, so the DIF API provides a common, tested implementation in class `com.ibm.retail.di.xml.messaging.service.AbstractActor`.

AbstractActor

`(com.ibm.retail.di.xml.messaging.AbstractActor)`

`AbstractActor` provides implementations for the lifecycle methods required by the Actor interface and is able to distinguish between non-SOAP and SOAP messages. By convention, an Actor must know how to cope with SOAP messages; it may optionally accept non-SOAP messages. For more details, see “StreamActor” on page 3-9.

An `AbstractActor` has these features:

- It rejects non-SOAP messages by default, generating a fault message.
- It automatically retrieves a SOAP MessageFactory from the associated Context, or creates a new MessageFactory.

- It provides support for user-defined message header properties.
- It provides support for message “breakpointing”, which is a mechanism for halting the flow of a message through a Director’s message flow.
- It provides convenient methods for retrieving Context keys and for generating fault messages.

The following Actor subclasses `AbstractActor` and generates a text message response regardless of the input to the Actor:

```
package examples.actors;

import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPMessage;

import com.ibm.retail.di.naming.Context;
import com.ibm.retail.di.xml.messaging.service.AbstractActor;
import com.ibm.retail.di.xml.messaging.soap.SoapUtil;

public class TextMessageActor extends AbstractActor
{
    private String text;

    protected void doInit() throws Exception
    {
        super.doInit();
        text = getContextString ("text", Context.REQUIRED);
    }

    public SOAPMessage onMessage(SOAPMessage aMessage)
    {
        SOAPMessage response = null;

        try {
         response = SoapUtil.createSoapMessage(
          getMessageFactory(),
          null, // no mpid is set
          text);
        }
        catch (SOAPException soapE) {
         response = createSoapFaultMessage (soapE);
        }

        return response;
    }
}
```

This Actor always returns the same response message. The response is a SOAP message that has a single attachment. The attachment is a text message whose content is retrieved from the Context. Note that `onMessage()` cannot throw an exception; it must return a response. Therefore, exceptions must be caught and handled; a convenience method called `createSoapFaultMessage()` can encapsulate the exception and report it as the response.

If a non-SOAP message is passed to this Actor, an exception is thrown automatically by the superclass. This exception is then encapsulated into MessageHeaders of the response and reported back to the caller, similar to the SOAP fault message generated if the input is a SOAP message.

Note: The “fault” concept is defined by the SOAP specification. As such, when an error occurs while processing a SOAP message, a special SOAP fault object can be returned as the result. For non-SOAP messages, DIF attempts to provide a similar mechanism. It does this by defining header properties

corresponding to a fault count and fault message. If an error occurs while processing a non-SOAP message within DIF, these fault properties are set on the response. The message body of the response is either the same as the request, or null.

AbstractActor provides a method for implementing non-SOAP message responses: `processNonSoapRequest()`. Override this method to provide a response to non-SOAP messages; the default implementation simply throws an exception. Consider this modified `TextMessageActor`:

```
package examples.actors;

import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPMessage;

import com.ibm.retail.di.naming.Context;
import com.ibm.retail.di.net.protocol.MessageHeaders;
import com.ibm.retail.di.xml.messaging.service.AbstractActor;
import com.ibm.retail.di.xml.messaging.soap.SoapUtil;

public class TextMessageActor extends AbstractActor
{
    private String text;

    protected void doInit() throws Exception
    {
        super.doInit();
        text = getContextString ("text", Context.REQUIRED);
    }

    protected java.lang.Object processNonSoapRequest(
        MessageHeaders theRequestHeaders,
        java.lang.Object theRequest,
        MessageHeaders theResponseHeaders)
        throws Exception
    {
        byte[] b = text.getBytes();
        return b;
    }

    public SOAPMessage onMessage(SOAPMessage aMessage)
    {
        SOAPMessage response = null;

        try {
            response = SoapUtil.createSoapMessage(
                getMessageFactory(),
                null, // no mpid is set
                text);
        }
        catch (SOAPException soapE) {
            response = createSoapFaultMessage (soapE);
   }

        return response;
    }
}
```

Suppose that a subclass of `AbstractActor` is used in a Director's message flow. In some cases, an Actor may wish to filter inbound messages; for example, if the inbound message has extraneous content or null content, or if the Actor has determined that the message should not be processed for pacing or performance reasons. In these cases, the Actor would like to return a response and indicate to

the Director that no additional Actors should be called for the current message flow. This action (called "breakpointing") is not considered a failure, but rather is treated as a successful return.

To breakpoint a message, call one of the following `AbstractActor` methods:

```
protected void setMessageBreakpoint(MessageHeaders theHeaders)
protected void setMessageBreakpoint(SOAPMessage aSoapMessage)
```

Use the first signature for non-SOAP messages; pass the response message headers as an argument. Use the second signature for SOAP messages; pass the SOAP response as an argument. The Director will return the response of this Actor directly to the caller, ignoring any successive Actors in the flow.

To summarize, the following results are possible when calling a subclass of `AbstractActor`:

- `onMessage()` or `processNonSoapRequest()` successfully return a result
- `onMessage()` returns a SOAP fault message, or `processNonSoapRequest()` throws an exception, which results in fault headers being set: the message flow is aborted
- `onMessage()` or `processNonSoapRequest()` calls `setMessageBreakpoint()`: the associated response is returned as-is, ending the message flow

Note:

Whenever an Actor is involved in a Director's message flow, the non-SOAP response properties returned by the current Actor will serve as the inbound request properties for the next Actor in the flow. This means that any of the inbound request properties for the first Actor in the flow will be lost unless the Actor explicitly copies those properties into its response message headers. For example, suppose Actor A is the first actor in a flow, and is followed by Actor B. Actor B depends on the presence of a request header property called `MessageFormat`. If a request is inbound to Actor A, and the request contains the `MessageFormat` request header property, this property will be lost unless Actor A explicitly copies it to the response headers. The response headers are used by the Director as input to Actor B. Hence, Actor B will not see the `MessageFormat` header property unless Actor A preserves the property. There are two exceptions to this rule:

- The *MessageProfileId* and *AssuranceId* properties are automatically copied by the Director.
- Any property set using the `AbstractActor`'s `setUserProperty()` will be copied automatically by the Director.

In general, when extending `AbstractActor`, the following techniques should be applied:

- If overriding `doInit()` or `doClose()`, care must be taken to first call the super implementation in order to preserve the base behavior.
- Handle non-SOAP requests by overriding `processNonSoapRequest()`; or use StreamActor (see "StreamActor").
- To ensure that a header property is passed from the current Actor to the next Actor, copy the property to the response message headers; or, set the property on the response headers using `setUserProperty()`.

StreamActor

`(com.ibm.retail.di.xml.messaging.StreamActor)`

StreamActor extends AbstractActor and therefore inherits all of its capabilities. However, a StreamActor specializes in processing a raw stream of data rather than a SOAP message. As such, it is a good choice for non-SOAP input messages and supports them easily. Likewise, inbound SOAP messages can be processed as streams, but subclasses must determine how to convert the SOAP message or its parts to a stream for processing. All stream processing is done by a single method, regardless of the type of inbound message: doInteraction().

When writing an Actor, extend from StreamActor if:

- the actor must operate on one or more binary SOAP attachments
- the actor must operate on one or more SOAP body elements after they have been converted to raw bytes
- the actor processes non-SOAP messages from a CBASIC or C client
- the actor produces a byte stream as its response

A StreamActor has these features:

- It passes non-SOAP messages directly to doInteraction() for processing.
- For SOAP messages, it provides two mechanisms for converting SOAP to a stream and also allows subclasses to replace this function (see “SOAP messages” on page 3-11).
- It provides an easy-to-use infrastructure for processing only the attachments of a SOAP message.

When developing a subclass of StreamActor, determine whether it must handle SOAP messages; if not, the implementation can be as trivial as the following example which merely “echoes” its input:

```
package examples.actors;

import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPMessage;

import com.ibm.retail.di.naming.Context;
import com.ibm.retail.di.net.protocol.MessageHeaders;
import com.ibm.retail.di.xml.messaging.service.StreamActor;
import com.ibm.retail.di.xml.messaging.soap.SoapUtil;

import com.ibm.retail.di.util.ByteTools;

public class EchoActor extends StreamActor
{
    protected byte[] doInteraction(
   String aProfileId,
        MessageHeaders theRequestHeaders,
        InputStream theContent,
        MessageHeaders theResponseHeaders)
        throws Exception
    {
        byte[] b = ByteTools.getStreamAsBytes (theContent);
        return b;
    }

    public SOAPMessage onMessage(SOAPMessage aMessage)
    {
        Exception e = new RetailException (
            "SOAP messages not supported!");

        SOAPMessage response = createSoapFaultMessage (e);
```

```
        return response;
    }
}
```

Note the `doInteraction()` method. This method handles all stream processing for the Actor, regardless of the type of the inbound message. In the example above, it simply reads the inbound stream completely, converts it to a byte array, and returns the result. (The `ByteTools` class provides some useful byte manipulation functions, such as `getStreamAsBytes()`.)

SOAP messages

Suppose that the Actor must support SOAP messages as well. The default `StreamActor` handles SOAP messages in this way:

- If the SOAP message has no attachments, a MIME property called `soap.content` is retrieved from the inbound message.
  - If not defined, or if its value is "soap.default", the entire SOAP message will be serialized into a stream using `ByteTools.getObjectAsStream()`. This stream is passed to `doInteraction()`.
  - If its value starts with `soap.body:`*`ElementName`*, an element of the SOAP message body having the name *ElementName* is written to a stream; if no such element exists, a fault is generated. This stream is passed to `doInteraction()`.
- If the SOAP message has at least one attachment, each attachment is processed independently via the following sequence:
  1. `shouldProcessAttachment()` returns a Boolean indicating whether the attachment should be processed.
  2. If **true**, the attachment is passed to `getAttachmentContentAsStream()`, which returns a stream representing the attachment content.
  3. The stream is passed to `doInteraction()` to generate a response; all MIME headers are copied to the request headers passed to the method.
  4. The value returned by `doInteraction()` is passed to `applyAttachmentResponse()`, which by default adds the resulting byte array as a binary attachment on the SOAP response message. All response properties from `doInteraction()` are copied to the MIME headers of the resulting attachment.

Note: If `doInteraction()` sets a message breakpoint for all attachments, the entire SOAP request will be breakpointed.

The `StreamActor` provides a means to override all of this behavior for SOAP messages. For SOAP messages without attachments, override `processSoapBody()`. For SOAP messages with attachments, override `processSoapAttachments()`.

In general, when extending `StreamActor`, the following techniques should be applied:

- If overriding `doInit()` or `doClose()`, care must be taken to first call the super implementation to ensure that the base behavior is preserved.
- Note that passing a SOAP message with one or more binary attachments to the Actor is roughly equivalent to passing those attachments as individual non-SOAP messages, collecting the results, and adding the results as attachments to a new SOAP message.
- By default, attachments are converted to input streams by calling `ByteTools.getObjectAsStream()`. This convenience method will fail unless the

object is a byte array, stream, String, or SOAPMessage; therefore, if a custom `StreamActor` must handle special Object types as attachments, the actor must override `getAttachmentContentAsStream()` to correctly convert each attachment to a stream.

- To change the way MIME headers are set on the SOAP response message with attachments, override `applyAttachmentResponse()`.
- To selectively filter specific attachments during processing of a SOAP message, override `shouldProcessAttachment()` and return **false**; or, in `doInteraction()`, call `setMessageBreakpoint()` on the response headers of the appropriate inbound attachment.

Appendix A. Standard Actors & Services

This appendix lists all Actors and Services pre-packaged with the Data Integration Facility and describes the configuration properties associated with each. By default, the configuration properties file is found in the `config/` directory and is called `services.complete.properties`. All of the keys listed in the following tables must be prefixed with the unique name of the Service or Actor, unless noted. For example, *uri* becomes `ServiceId.uri`, for a Service whose unique name is `ServiceId`.

TCP/IP client Service

`(com.ibm.retail.di.service.soeps.SoepsTcpipService)`

Listens to a local port for inbound requests from clients which communicate via the SOEPS protocol. Inbound request messages are passed to the associated Actor for processing. The response is passed back to the client through the established TCP/IP socket connection, which is then closed.

Key	Description	Valid values
`uri`	`soeps{:port}` By default, listens on port 6696; or, specify another port. For example, `soeps:9120`.	Any available port
`listener`	Required. Specifies the ActorId of the actor that will process inbound requests. Usually this is the Director.	Any configured Actor

4690 pipes client Service

`(com.ibm.retail.di.service.soep.SoepPipeService)`

Listens to a local pipe for inbound requests from clients which communicate via the SOEP protocol. Inbound request messages are passed to the associated Actor for processing. The response is passed back to the client through a pipe named in the request; the response pipe is then closed.

Key	Description	Valid values
uri	soep{:*prs-pipe*} | soep{:*named-pipe* where *prs-pipe* is a single alpha character representing an available, valid PRS pipe on the local machine, and *named-pipe* is a valid named pipe on the local machine (which must be a controller). The *named-pipe* has the form: pi:*name* where name is eight characters or less. If only soep is specified, the service listens to the local Controller named pipe pi:server.	Any valid local PRS or named pipe
listener	Required. Specifies the ActorId of the actor that will process inbound requests. Usually this is the Director.	Any configured Actor

4690 DiskQ Service

(com.ibm.retail.di.service.diskq.PosDiskQService)

Reads 4690 transactions from the 4690 disk queue, which must be defined with the logical name DIFQUEUE. Transactions must be prefixed with the DIF header (defined in the appendix). Relevant data from the header is copied to the message header properties and the header itself is removed from the data. The resulting message is passed to the associated Actor.

Key	Description	Valid values
uri	local:com.ibm.retail.di.service.diskq.PosDiskQService	Fixed
listener	Required. Specifies the ActorId of the actor that will process inbound requests. Usually this is the Director.	Any configured Actor
message-profile-id	Required. Specifies the message profile Id to associate with all messages generated by this service.	Any unique Id
*	Optional pacing/bundling settings	

The DiskQService offers the ability to bundle multiple diskq messages per outgoing message, and can regulate the flow of messages using a built-in or custom pacing algorithm. Refer to Appendix D, “DiskQService bundling and pacing” for details.

ParserActor

(com.ibm.retail.di.transform.actor.ParserActor)

Parses the inbound request, which must be either a raw 4690 transaction or a SOAP message whose attachments are each 4690 transactions. The resulting message is an XML document whose structure is a one-to-one mapping of the records and fields in the input transaction. This XML document can be used as input to the TransformerActor to perform XSLT mapping to standard XML formats.

Key	Description	Valid values
ParserActor.parseFormatFile	Specifies the complete path and filename of the XML document that defines the inbound binary format. The following specifications are available, corresponding to SA, GSA, and ACE applications: saspecs.xml gsaspecs.xml acespecs.xml	Any XML document adhering to tlogschm.xsd

TransformerActor

(com.ibm.retail.di.transform.actor.TransformerActor)

Translates the inbound document by running an XSLT map. The input can be a single piece of data or a multi-attachment SOAP message. Typically this actor is used to transform a parsed retail transaction to an IXRetail POSLog document. Note: Requires 32MB free RAM for GSA and SA, or 64MB free RAM for ACE.

Key	Description	Valid values
TransformerActor. transformationFileName	Specifies the complete path and filename of the XSL mapping file to execute on the parsed transaction data. The following specifications are available, corresponding to SA/ACE and GSA applications: poslgace.xsl poslggsa.xsl	Any valid XSL document whose input is a parsed transaction

ParsingTransformerActor

(com.ibm.retail.di.transform.actor.ParsingTransformerActor)

This actor is a combination of ParserActor and TransformerActor. Using ParsingTransformerActor provides better performance than using ParserActor and TransformerActor individually.

Key	Description	Valid values
ParsingTransformerActor. parseFormatFile	Specifies the complete path and filename of the XML document that defines the inbound binary format. The following specifications are available, corresponding to SA, GSA, and ACE applications: saspecs.xml gsaspecs.xml acespecs.xml	Any XML document adhering to tlogschm.xsd
ParsingTransformerActor. transformationFileName	Specifies the complete path and filename of the XSL mapping file to execute on the parsed transaction data. The following specifications are available, corresponding to SA/ACE and GSA applications: poslgace.xsl poslggsa.xsl	Any valid XSL document whose input is a parsed transaction

MqeActor

(com.ibm.retail.di.service.mom.mqe.MqeActor)

Passes the inbound message to the local MQe queue manager for optional routing to a remote queue, such as an MQSeries queue. The actor supports oneway assured messages or simple request/reply messages. In either case, the target queue manager and queue must be specified in the message's header properties or in the actor's configuration. All messages are MQSeries-compatible.

Key	Description	Valid values
mpid.mom.target.qmgr	Required for each unique message profile Id (*mpid*). Specifies the target queue manager for all messages having the given message profile Id. The target queue manager must be defined on the local MQe queue manager.	Any remote queue manager defined as a Connection on the local queue manager instance.

Key	Description	Valid values
mpid.mom.target.queue	Required for each unique message profile Id (*mpid*). Specifies the target queue for all messages having the given profile Id. The target queue must be defined on the local MQe queue manager and must belong to the remote queue manager named in *mpid*.mom.target.qmgr.	Any remote queue associated with the given target queue manager, as configured on the local MQe queue manager instance.
mpid.mom.interaction.uri	Required for each unique message profile Id (*mpid*). Specifies the style of messaging to use for all messages having the given profile Id. The current options are: **oneway** assured oneway messaging **reqresp** request/response messaging (unassured) If *oneway* is specified, the message must also define an *assurance Id*, which is a unique value associated exclusively with the message and used to ensure that the message is put only once. If *reqresp* is specified, the mom.replyto.qmgr and mom.replyto.queue keys must also be set in configuration or on the message.	One of these: • oneway • reqresp
mpid.mom.replyto.qmgr	Required for each unique message profile Id (*mpid*). Specifies the queue manager that owns the associated reply queue for all messages having the given message profile Id. The replyto queue manager must be defined on the local MQe queue manager.	Any local or remote queue manager defined on the local queue manager instance.

Key	Description	Valid values
mpid.mom.replyto.queue	Required for each unique message profile Id (*mpid*). Specifies the queue on which replies will be placed for *reqresp* interactions for all messages having the given message profile Id. The replyto queue must be defined on the local MQe queue manager.	Any local or remote queue associated with the given replyto queue manager, as configured on the local MQe queue manager instance.

mpid.state.uri	Optional for each unique message profile Id (*mpid*).
	
	Specifies the resource that stores the persistent state information for assured messages having the given message profile Id.
	
	By default, the state is stored in a local file called:
	`\adx_idt4\diftlog.mqe`
		`file://`*complete-filepath*
		
		For example:
		`file://C:/adx_idt4/diftlog.mqe`
		
		To distribute to alternate controller:
		`posfile://C:/adx_idt4/diftlog.mqe`

mpid.state.backup.uri	Optional for each unique message profile ID. Specifies the backup resource to maintain for persistent state information.
	
	By default, a backup is not created. A backup state offers the most protection against failures, including corruption of the primary state.
		`file://`*complete-filepath*
		
		For example:
		`file://C:/adx_idt4/diftlog.mqe`
		
		To distribute to alternate controller:
		`posfile://C:/adx_idt4/diftlog.mqe`

JmsActor

(`com.ibm.retail.di.service.mom.jms.JmsActor`)

Passes the inbound message to the JMS queue manager associated with this actor. The actor supports one-way assured messages or simple request/reply messages. In either case, the target queue manager and queue must be specified in the message's header properties or in the actor's configuration. Each instance of `JmsActor` is associated with a JMS profile, which defines the associated queue manager. See Chapter 2, "Configuration and operation" for instructions for configuring JMS profiles.

Note: For assured one-way messaging, a DI synchronization queue must be created on the associated queue manager. The queue name must have the form `DI.SYNCQ.`*mpid*, where *mpid* is a message profile ID that requires assured messaging. One of these queues must be created for every message profile ID that uses assured one-way messaging.

Key	Description	Valid values
mpid.mom.target.qmgr	Required for each unique message profile Id (*mpid*). Specifies the target queue manager for all messages having the given message profile Id. This is usually the queue manager associated with the Actor.	Any queue manager recognized by the associated queue manager.
mpid.mom.target.queue	Required for each unique message profile Id (*mpid*).	Any queue recognized by the associated queue manager.
mpid.mom.interaction.uri	Required for each unique message profile Id (*mpid*). Specifies the style of messaging to use for all messages having the given profile Id. The current options are: **oneway** assured oneway messaging to be used for any style of asynchronous puts, including remote MQe or MQSeries target queues **reqresp** request/response messaging (unassured)	One of these: • oneway • reqresp
mpid.mom.replyto.qmgr	Required for each unique message profile Id (*mpid*). Specifies the queue manager that owns the associated reply queue for all messages having the given message profile Id. This is usually the associated queue manager.	Any queue manager recognized by the associated queue manager.
mpid.mom.replyto.queue	Required for each unique message profile Id (*mpid*). Specifies the queue on which replies will be placed for reqresp interactions for all messages having the given message profile Id. The replyto queue must be defined on the associated queue manager.	Any local or remote queue associated with the given replyto queue manager, as recognized by the queue manager associated with this actor.

Key	Description	Valid values
mpid.aos.queue	Optional for each unique message profile Id (*mpid*). Specifies the queue of the associated queue manager that stores the persistent state information for assured messages having the given message profile Id. By default, the state is stored in a local queue called DI.SYNC.*mpid*, where *mpid* is the message profile Id of the message.	Any queue defined on the associated queue manager; the queue may not be used by other message profile Ids.
mpid.aos.msgid	Optional for each unique message profile Id (*mpid*). Specifies a String that is an enterprise-wide unique identifier for this message type at this source location. If not specified, the hostname of the local machine is used. On most 4690 systems, this cannot be determined automatically and therefore this key must be defined. In most cases it is sufficient to use the store number, unless multiple assured oneway message types will be sent from the same store; in this case, the store number plus a unique field can be used, as in the example S100TLOG.	Any string up to 24 characters in length. (Example: S100TLOG)

WmqiRetailFormatActor

(com.ibm.retail.di.service.wmqi.WmqiRetailFormatActor)

Converts the inbound message to SOAP, if it is non-SOAP. Adds SOAP header elements required for WMQI parsing, including store number and message profile ID. Other elements can be added dynamically. This actor typically precedes the JmsActor or MqeActor.

Key	Description	Valid values
uri	local:com.ibm.retail.di.service.wmqi.WmqiRetailFormatActor	Fixed

Key	Description	Valid values
header.localName	Required. Specifies the name of the DIF header element to add to the SOAP header. By convention, this is Header.	Header
header.prefix	Required. Specifies the element name prefix (namespace) for the elements to add to the SOAP header. By convention, this is dif.	dif
header.uri	Required. Specifies the URI of the namespace for the named elements that are added to the SOAP header. By convention, this is http://www.ibm.com/retail/namespace/dif.	Fixed
mpid.header-element.element-name	Optional. Names additional elements to be added to the DIF header per message profile Id type. Elements will have the name give by element-name. The value of the element is the text assigned to this property.	Any text; associated with the named element

WMQI will typically require these additional elements in the DIF header:

```
POSType=IBM ACE | IBM GSA |  IBM SA
TlogFormat=Binary | XML | IXRetail
```

Abstract Services

The following list describes services that are provided as abstract base classes that can be extended to provide customer-specific function. These services cannot be executed directly.

MQe Listener Service

(com.ibm.retail.di.service.mom.mqe.MqeListenerService)

Registers as a listener of a specific local MQe queue and performs processing on each inbound message as it appears.

Subclasses must minimally define:

```
onMessage(MQeMsgObject aMessage);
```

Key	Description	Valid values
uri	local:com.ibm.retail.di.service.mom.MqeListenerService	Fixed
queue	Required. Identifies the local MQe queue whose messages should be processed by this service.	Any local MQe queue

JMS Listener Service

(com.ibm.retail.di.service.mom.jms.JmsListenerService)

Registers as a listener of a specific JMS queue associated with this Service's queue manager, and performs processing on each inbound message as it appears. The Service automatically handles error conditions by repairing the JMS connection asynchronously as required.

Subclasses must minimally define:

```
onMessage(javax.jms.Message aMessage);
```

Key	Description	Valid values
uri	local:com.ibm.retail.di.service.mom.MqeListenerService	Fixed
queue	Required. Identifies the JMS queue whose messages should be processed by this service; the queue must be owned by the queue manager associated with this Service	Any local MQe queue
profile-id	Required. Names the JMS profile ID to use for this Service; this determines the queue manager that is used.	Any JMS profile Id defined in the services properties file

Additional actors

The following actors are available for lower-level functions:

- SoapToBytesConversionActor
- BytesToSoapConversionActor

For details of these Actors, refer to the Javadoc.

Appendix B. Message formats

This appendix describes the format of outbound messages generated by a combination of the `WmqiRetailFormatActor` and one of the MOM actors: `JmsActor` or `MqeActor`. Developers of host-side systems receiving messages from DIF should use this appendix as a specification for the message structures received in the body of the MOM message object.

Websphere MQSeries Integrator

For messages destined to Websphere MQSeries Integrator, a standard MQSeries message object is created using the base WMQ Java API (`com.ibm.mq.MQMessage`). The resulting message has a body whose content has leading MIME headers followed by a "streamed" SOAP message. The SOAP message has a header and one or more attachments.

The current implementation does not set any MQMD header fields, but the header will be present with default values. The CCSID and encoding fields will be set accordingly and should be used to interpret the leading MIME headers.

An RFH2 header is not present. In addition, attachments are never compressed. See "MOM message body format for SOAP messages" on page B-2.

JMS Systems

DIF uses a JMS `BytesMessage` (`javax.jms.BytesMessage`) for messages destined to JMS systems. This includes JMS messages sent to MQSeries queues. The payload of the message is either a raw byte stream or a SOAP message. For a SOAP message, the payload has leading MIME headers followed by a "streamed" SOAP message.

The current implementation does not set any MQMD or RFH2 headers by default, but can do so if configured appropriately. This will be customer-specific. The CCSID and encoding fields of the MQMD header will be set accordingly and should be used to interpret the leading MIME headers.

In an MQ-based JMS system, the resulting JMS `BytesMessage` can be made compatible with WMQI and other non-JMS receivers by using the URI notation for specifying the target queue; for example:

```
MyActor.mom.target.queue=queue:///Q.TARGET?targetClient=1
```

See "MOM message body format for SOAP messages" on page B-2 and the WMQ/JMS documentation for details.

Websphere MQSeries Everyplace

By default, DIF creates an MQSeries-compatible message object (`com.ibm.mqemqmessage.MQeMQMsgObject`). The resulting message is similar to a JMS message, without an RFH2 header. The payload is either a raw byte stream or a SOAP message. For a SOAP message, the payload has leading MIME headers followed by a "streamed" SOAP message.

The current implementation does not set any MQMD headers by default, but can do so if configured appropriately. This will be customer-specific. The CCSID and encoding fields of the MQMD header will be set accordingly and should be used to interpret the leading MIME headers.

A WMQe Gateway/Bridge at the host will convert the inbound message to a “native” MQSeries message. This will be compatible with a WMQI input node. See “MOM message body format for SOAP messages”.

MOM message body format for SOAP messages

For a SOAP message payload, a series of MIME headers precedes the SOAP message stream. The first MIME header is always the MIME-Version header, whose value is currently 1.0. In addition, the Content-Type given by the leading MIME headers is always `text/xml` to represent a SOAP message.

As currently implemented, each attachment has the same Content-Type. The possible options are:

text/plain
Non-SOAP XML document (typically IXRetail)

application/octet-stream
Raw binary data

For binary attachment data (application/octet-stream), the Content-Transfer-Encoding MIME header may be set (per-attachment). If present, it indicates the encoding used to represent the binary data. Currently, it has one of the following values:

binary The data is not encoded; default.

base64
The data is encoded using Base64.

The message body has the following structure:

```
 MIME-Version: 1.0
Content-Type: multipart/related; type="text/xml"; boundary=unique-boundary

Property: Value


--unique-boundary   (indicates start of the SOAP message)
Content-Type: text/xml
<?xml version="1.0" encoding="UTF-8"?>
<soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:dif="http://www.ibm.com/retail/namespace/dif/">
<soap-env:Header> (DIF Header) </soap-env:Header>
<soap-env:Body/>
</soap-env:Envelope>

--unique-boundary   (indicates end of SOAP body, start of attachment)

Content-Type: text/plain   Additional MIME headers per attachment

Attachment Content

--unique-boundary   (indicates end of previous attachment, start of attachment)

Content-Type: text/plain  Additional MIME headers per attachment
```

```
Attachment Content

--unique-boundary--  (indicates end of SOAP message)
```

The SOAP Header is given by the `DIFMessage.xsd` schema (see included file). The following is an example:

```
<soap-env:Header>
 <dif:Header>
  <dif:MessageProfileId>Tlog</dif:MessageProfileId>
  <dif:StoreNumber>112</dif:StoreNumber>
  <dif:POSType>IBM GSA</dif:POSType>
  <dif:TlogFormat>IXRetail</dif:TlogFormat>
 </dif:Header>
</soap-env:Header>
```

Note: Content-Type is a required MIME header because it specifies a unique key found in the boundary of the SOAP message and its attachments. This key must match the boundaries found in the streamed SOAP message. In fact, the SAAJ (SOAP with Attachments API for Java) *requires* that MIME headers with an accurate Content-Type be supplied in order to rebuild a streamed SOAP message.

Examples

The following examples illustrate message formats.

Single IXRetail transaction

The following is an example of a message created by DIF for a single IXRetail transaction.

```
MIME-Version: 1.0
Content-Type: multipart/related; type="text/xml";
boundary=329832539.1052154477517.JavaMail.javamailuser.localhost
Content-Length: 2030
SOAPAction: ""
Assurance-Id: 10670
Content-Transfer-Size: 2030
Message-Profile-Id: TLOG
user.Store-Id: 1118
--329832539.1052154477517.JavaMail.javamailuser.localhost
Content-Type: text/xml

<?xml version="1.0" encoding="UTF-8"?>
<soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:dif="http://www.ibm.com/retail/namespace/dif/"><soap-env:Header>
<dif:Header><dif:MessageProfileId>TLOG</dif:MessageProfileId>
<dif:StoreNumber>1118</dif:StoreNumber><dif:POSType>IBM GSA</dif:POSType>
<dif:TlogFormat>IXRetail</dif:TlogFormat></dif:Header></soap-env:Header>
<soap-env:Body/></soap-env:Envelope>
--329832539.1052154477517.JavaMail.javamailuser.localhost
Content-Type: text/plain
Assurance-Id: 10670
Store-Id: 1118

<?xml version="1.0" encoding="UTF-8"?>
<POSLog xmlns="http://www.nrf-arts.org/IXRetail/namespace/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:fo="http://www.w3.org/1999/XSL/Format"
xmlns:ace="Lookup table for taxes"
xmlns:sa="Lookup table for SA extensions"
xsi:schemaLocation="http://www.nrf-arts.org/IXRetail/namespace/
POSLogRetailTransactionIBMGrocery.xsd">
<Transaction xsi:type="POSLogRetailTransactionGSA" Version="1.0" CancelFlag="false"
```

```
TrainingModeFlag="false" OfflineFlag="false" OutsideSalesFlag="false"
SuspendFlag="false">
<RetailStoreID>Retail Store Solutions 123</RetailStoreID>
<WorkstationID>0008</WorkstationID>
<SequenceNumber>1</SequenceNumber>
<EndDateTime>2002-08-01T10:57:00</EndDateTime>
<OperatorID>00000000</OperatorID>
<CurrencyCode>USD</CurrencyCode>
<GSALineItem VoidFlag="false" EntryMethod="Keyed">
<SequenceNumber>10</SequenceNumber>
<EndDateTime>2002-08-01T10:57:00</EndDateTime>
<GSALogTimeAnalysis>
<GSAOperator>00000000</GSAOperator>
<GSATerminal>0008</GSATerminal>
<GSARingTime>0</GSARingTime>
<GSATenderTime>8</GSATenderTime>
<GSANonSalesTime>0</GSANonSalesTime>
<GSAInactiveTime>166</GSAInactiveTime>
<GSATime>1057</GSATime>
<GSADate>020801</GSADate>
</GSALogTimeAnalysis>
</GSALineItem>
</Transaction>
</POSLog>

--329832539.1052154477517.JavaMail.javamailuser.localhost—
```

Multiple IXRetail transactions

The following is an example of a message created by DIF for multiple IXRetail transactions (bundling).

```
MIME-Version: 1.0
Content-Type: multipart/related; type="text/xml";
boundary=1190412550.1052157878298.JavaMail.javamailuser.localhost
Content-Length: 4944
SOAPAction: ""
Assurance-Id: 10750
Content-Transfer-Size: 4944
Fault-Count: 0
Message-Profile-Id: TLOG
user.Store-Id: 1118
--1190412550.1052157878298.JavaMail.javamailuser.localhost
Content-Type: text/xml

<?xml version="1.0" encoding="UTF-8"?>
<soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:dif="http://www.ibm.com/retail/namespace/dif/"><soap-env:Header>
<dif:Header><dif:MessageProfileId>TLOG</dif:MessageProfileId>
<dif:StoreNumber>1118</dif:StoreNumber><dif:POSType>IBM GSA</dif:POSType>
<dif:TlogFormat>IXRetail</dif:TlogFormat></dif:Header></soap-env:Header>
<soap-env:Body/></soap-env:Envelope>
--1190412550.1052157878298.JavaMail.javamailuser.localhost
Content-Type: text/plain
Assurance-Id: 10730
Store-Id: 1118

<?xml version="1.0" encoding="UTF-8"?>
<POSLog xmlns="http://www.nrf-arts.org/IXRetail/namespace/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:fo="http://www.w3.org/1999/XSL/Format"
xmlns:ace="Lookup table for taxes" xmlns:sa="Lookup table for SA extensions"
xsi:schemaLocation="http://www.nrf-arts.org/IXRetail/namespace/
POSLogRetailTransactionIBMGrocery.xsd">
<Transaction xsi:type="POSLogRetailTransactionGSA" Version="1.0" CancelFlag="false"
TrainingModeFlag="false" OfflineFlag="false" OutsideSalesFlag="false"
SuspendFlag="false">
```

```
<RetailStoreID>Retail Store Solutions 123</RetailStoreID>
<WorkstationID>0063</WorkstationID>
<SequenceNumber>1</SequenceNumber>
<EndDateTime>2002-08-01T10:57:00</EndDateTime>
<OperatorID>00000000</OperatorID>
<CurrencyCode>USD</CurrencyCode>
<GSALineItem VoidFlag="false" EntryMethod="Keyed">
<SequenceNumber>10</SequenceNumber>
<EndDateTime>2002-08-01T10:57:00</EndDateTime>
<GSALogTimeAnalysis>
<GSAOperator>00000000</GSAOperator>
<GSATerminal>0063</GSATerminal>
<GSARingTime>0</GSARingTime>
<GSATenderTime>20</GSATenderTime>
<GSANonSalesTime>0</GSANonSalesTime>
<GSAInactiveTime>410</GSAInactiveTime>
<GSATime>1057</GSATime>
<GSADate>020801</GSADate>
</GSALogTimeAnalysis>
</GSALineItem>
</Transaction>
</POSLog>

--1190412550.1052157878298.JavaMail.javamailuser.localhost
Content-Type: text/plain
Assurance-Id: 10740
Store-Id: 1118

<?xml version="1.0" encoding="UTF-8"?>
<POSLog xmlns="http://www.nrf-arts.org/IXRetail/namespace/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:fo="http://www.w3.org/1999/XSL/Format"
xmlns:ace="Lookup table for taxes" xmlns:sa="Lookup table for SA extensions"
xsi:schemaLocation="http://www.nrf-arts.org/IXRetail/namespace/
POSLogRetailTransactionIBMGrocery.xsd">
<Transaction xsi:type="POSLogRetailTransactionGSA" Version="1.0" CancelFlag="false"
TrainingModeFlag="false" OfflineFlag="false" OutsideSalesFlag="false"
SuspendFlag="false">
<RetailStoreID>Retail Store Solutions 123</RetailStoreID>
<WorkstationID>0036</WorkstationID>
<SequenceNumber>1</SequenceNumber>
<EndDateTime>2002-08-01T10:57:00</EndDateTime>
<OperatorID>00000000</OperatorID>
<CurrencyCode>USD</CurrencyCode>
<GSALineItem VoidFlag="false" EntryMethod="Keyed">
<SequenceNumber>10</SequenceNumber>
<EndDateTime>2002-08-01T10:57:00</EndDateTime>
<GSALogTimeAnalysis>
<GSAOperator>00000000</GSAOperator>
<GSATerminal>0036</GSATerminal>
<GSARingTime>0</GSARingTime>
<GSATenderTime>0</GSATenderTime>
<GSANonSalesTime>0</GSANonSalesTime>
<GSAInactiveTime>114</GSAInactiveTime>
<GSATime>1057</GSATime>
<GSADate>020801</GSADate>
</GSALogTimeAnalysis>
</GSALineItem>
</Transaction>
</POSLog>

--1190412550.1052157878298.JavaMail.javamailuser.localhost
Content-Type: text/plain
Assurance-Id: 10750
Store-Id: 1118

<?xml version="1.0" encoding="UTF-8"?>
```

```
<POSLog xmlns="http://www.nrf-arts.org/IXRetail/namespace/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:fo="http://www.w3.org/1999/XSL/Format"
xmlns:ace="Lookup table for taxes" xmlns:sa="Lookup table for SA extensions"
xsi:schemaLocation="http://www.nrf-arts.org/IXRetail/namespace/
POSLogRetailTransactionIBMGrocery.xsd">
<Transaction xsi:type="POSLogRetailTransactionGSA" Version="1.0" CancelFlag="false"
TrainingModeFlag="false" OfflineFlag="false" OutsideSalesFlag="false"
SuspendFlag="false">
<RetailStoreID>Retail Store Solutions 123</RetailStoreID>
<WorkstationID>0016</WorkstationID>
<SequenceNumber>1</SequenceNumber>
<EndDateTime>2002-08-01T10:57:00</EndDateTime>
<OperatorID>00000000</OperatorID>
<CurrencyCode>USD</CurrencyCode>
<GSALineItem VoidFlag="false" EntryMethod="Keyed">
<SequenceNumber>10</SequenceNumber>
<EndDateTime>2002-08-01T10:57:00</EndDateTime>
<GSALogTimeAnalysis>
<GSAOperator>00000000</GSAOperator>
<GSATerminal>0016</GSATerminal>
<GSARingTime>0</GSARingTime>
<GSATenderTime>0</GSATenderTime>
<GSANonSalesTime>0</GSANonSalesTime>
<GSAInactiveTime>7</GSAInactiveTime>
<GSATime>1057</GSATime>
<GSADate>020801</GSADate>
</GSALogTimeAnalysis>
</GSALineItem>
</Transaction>
</POSLog>

--1190412550.1052157878298.JavaMail.javamailuser.localhost--
```

Appendix C. Sample scenarios

This appendix describes the sample code that is packaged with DIF. The sample scenarios attempt to model common store integration tasks and can be extended to support customized functions.

Enterprise-to-Store request/response using MQe and 4690 keyed kiles

Source code: `examples/keyedfile`

The keyed file example shows how DI can be used to perform a request/response using a 4690 keyed file to look up the response.

The required setup is a 4690 machine with a keyed file with the desired records, and a remote machine to perform the request for the keyed file records. After setting up MQe and creating and populating the keyed file, start DI on the 4690 machine using the provided `services.keyedfile.properties` as `difuser.pro`. The location of the keyed file as well as the MQe details must be provided in the properties file; for help with MQe setup on both machines, see “MQe setup”.

Once DI is running on the 4690 machine, run the `examples.keyedfile.MqeKeyedFileClient` program, using the correct parameters. This will start MQe on the remote (requestor) machine, which will then forward the request to 4690, where the key is looked up in the keyed file, and the associated record is returned.

MQe setup

The MQe setup for this example is a bit complicated. On the 4690 machine, a local queue must be created. This is the queue that incoming messages will arrive on, and is the queue that the DI service will listen to. Also, a Connection and remote queue are required. The Connection should point to the MQe queue manager on the remote machine, where the response will be sent. The remote queue should use this Connection and point to the remote machine’s local queue. The remote machine needs two queues also. It needs a local queue for responses to arrive on; this queue is what the 4690’s remote queue should point to. The remote machine also needs a Connection and remote queue; the Connection should point to the 4690 machine’s queue manager, and the remote queue should point to the 4690’s local queue. The remote machine will use this remote queue to send requests to the 4690 machine’s local queue, which the DI service is listening to. For example, the following commands will set up MQe on each machine.

Note: You may have to substitute `difrun.bat` for `java` in each of the following commands. Also, the \ indicates the next line should be typed in on the same line. The remote machine is called *`remote.machine`* and the 4690 machine is called *`4690.machine`*; you should substitute the actual names or IP addresses.

On the 4690 machine:

```
java examples.administration.commandline.IniFileCreator \
 ./QM4690.ini QM4690 ./Registry Server 8085 1234567 MySecret default \
 FileRegistry FastNetwork
java examples.install.SimpleCreateQM ./QM4690.ini ./Queues
java examples.administration.commandline.LocalQueueCreator \
 Q.REQUEST null null null -1 -1 QM4690 ./QM4690.ini \
 com.ibm.mqe.adapters.MQeDiskFieldsAdapter:./Qdir
```

```
java examples.administration.commandline.ConnectionCreator \
 QMremote QM4690 FastNetwork:remote.machine:8082 DefaultChannel ./QM4690.ini
java examples.administration.commandline.RemoteQueueCreator \
 Q.RESPONSE QMremote Synchronous null null null QM4690 ./QM4690
```

On the remote machine:

```
java examples.administration.commandline.IniFileCreator \
 ./QMremote.ini QMremote ./Registry Server 8082 1234567 MySecret default \
 FileRegistry FastNetwork
java examples.install.SimpleCreateQM ./QMremote.ini ./Queues
java examples.administration.commandline.LocalQueueCreator \
 Q.RESPONSE null null null -1 -1 QMremote ./QMremote.ini \
 com.ibm.mqe.adapters.MQeDiskFieldsAdapter:./Qdir
java examples.administration.commandline.ConnectionCreator \
 QM4690 QMremote FastNetwork:4690.machine:8085 DefaultChannel ./QMremote.ini
java examples.administration.commandline.RemoteQueueCreator \
 Q.REQUEST QM4690 Synchronous null null null QMremote ./QMremote
```

To run the `MqeKeyedFileClient` program (use a valid key):

```
java examples.keyedfile.MqeKeyedFileClient \
 ./QMremote.ini QM4690 Q.REQUEST key QMremote Q.RESPONSE
```

The response will then appear on the Q.RESPONSE queue, if the record lookup was successful.

Keyed file creation

In order to easily create a keyed file on 4690, the program `examples.keyedfile.MakeKeyedFile` is provided. It will create and fill the specified keyed file with some sample records. Refer to the source code of the class for details on what records are used. The request from the remote machine must specify a key that is contained in the keyed file. Note that keyed file keys are a specific length, and are the first part of their associated record.

Store-to-enterprise request/response using SOAP over HTTP

Source code: `examples/reqresp`

The request/response example demonstrates how a reqest can be converted between a byte[] and a SOAPMessage, using an arbitrary conversion pattern. It also shows how a Tomcat server can be used to process and/or modify a SOAPMessage.

The conversion between byte[] and SOAPMessage is done by the `SoapToBytesConversionActor` and the `BytesToSoapConversionActor`. Both actors extend the SoapStreamConversionActor, and simply provide functionality to enclose a byte[] message inside a SOAPMessage, or remove that previously-encoded byte[] from the SOAPMessage.

The Tomcat server must be set up to include the provided servlet. In the example, JWDSP was used, it can be downloaded at:

`http://java.sun.com/webservices/downloads/webservicespack.html`

To add the servlet to JWSDP, create a directory in the top level of the `JWDSP` directory; this example uses `di`. Place the provided `di.war` file in the `di` directory. Place the provided `di.xml` file in the webapps directory. If you use a directory other than `di`, you will need to modify the `di.xml` file. To start JWDSP, you must run the `bin/catalina` script provided by JWDSP. The servlet will now be available at `http://localhost:8080/di/diservlet` The logs from the servlet will be available in

the logs/launcher.server.log file in the JWDSP installation. If you have configured Tomcat and/or JWDSP on a different machine or using a different port, you should substitute that in the URL above.

To use the example, start DI using the provided services.reqresp.properties file as difuser.pro. This properties file is set up to convert byte[] input (ASCII) to SOAP, then send that to the servlet, and convert the response back to a byte[] (ASCII). You can send ASCII to DI using the examples.testing.SendFile class.

This example does not actually modify the message, besides the byte[] to SOAP conversion. The original message will be restored by the SOAP to byte[] conversion. The provided servlet, whose source is available in the di.war file, simply passes the message back unmodified.

Enterprise-to-store: assured one-way messaging over MQe

Source code: examples/syncpointed

The setup requires two systems, one with a local queue and one with a remote queue. For this example, the local queue system will be assumed to be a 4690 machine, and the remote queue system is the remote system. MQe setup in this example is not difficult. The following examples will generate a working MQe setup. The 4690 system is called 4690.machine, and the remote system is called remote.machine; you should substitute the correct values. All files and directories are placed in the current directory (using ./), which should be changed to whatever directory is appropriate.

On the 4690 machine:

```
java examples.administration.commandline.IniFileCreator \
 ./QM4690.ini QM4690 ./Registry Server 8085 1234567 MySecret default \
 FileRegistry FastNetwork
java examples.install.SimpleCreateQM ./QM4690.ini ./Queues
java examples.administration.commandline.LocalQueueCreator \
 Q.REQUEST null null null -1 -1 QM4690 ./QM4690.ini \
 com.ibm.mqe.adapters.MQeDiskFieldsAdapter:./Qdir
```

On the remote machine:

```
java examples.administration.commandline.IniFileCreator \
 ./QMremote.ini QMremote ./Registry Server 8082 1234567 MySecret default \
 FileRegistry FastNetwork
java examples.install.SimpleCreateQM ./QMremote.ini ./Queues
java examples.administration.commandline.ConnectionCreator \
 QM4690 QMremote FastNetwork:4690.machine:8085 DefaultChannel ./QMremote.ini
java examples.administration.commandline.RemoteQueueCreator \
 Q.REQUEST QM4690 Synchronous null null null QMremote ./QMremote
```

Note that this MQe setup is a subset of the MQe setup for the keyed file example.

After setup, start DI on the 4690 machine using the provided services.syncpointed.properties file as difuser.pro. Some parameters in the file must be modified, such as the location of the fault directory, and the queue name. Also, the examples.syncpointed.GenerateFaultActor has a property that must be set defining whether or not to generate a fault when a message is processed. This should be set to false to allow the message to pass through successfully, or true if a fault should be generated so the message is saved to the faults directory.

To generate a message, use the examples.syncpointed.MqeClient or examples.syncpointed.MqeSoapClient to generate either a byte[] message, or

SOAPMessage, respectively. Both programs require the correct parameters. The provided service and Actors do not actually process the data included with either type of message.

Appendix D. DiskQService bundling and pacing

Bundling

Message Bundling can be configured when running the `DiskQService`. The Message Bundling property defines the number of messages that will be processed together by the `DiskQService`.

When compression is activated, it may be desirable to enable Message Bundling. This is because data compression algorithms are typically more effective when they operate on blocks of data that have repeatable patterns, like Retail Transactions.

The setting for this configurable property depends on many factors such as:

- Whether Parsing and Transformation is configured to run in the store
- Available network bandwidth in the store
- Average size of a transaction
- Average rate of sale transactions

Key	Description	Valid values	
`DiskQService.MessageBundleCount`	Optional. Specifies the number of Messages (i.e., transactions) to pull from the DiskQ and process in a single outbound message to the host.	**0**	No bundling
		>0	Number of Messages (transactions) to process in a single iteration of the `DiskQService`

Pacing

Message Pacing can be configured when running the `DiskQService`. Pacing should be configured to limit the amount of network bandwidth that is used by the Data Integration Messaging process.

The user can define a set of Pacing properties for up to three time periods. This enables the user to have a better control of the amount of network bandwidth that is used during periods of peak sales activity in the store.

For example, a store may wish to reduce the amount of network bandwidth used by the Data Integration process during peak sales periods to ensure that sufficient network bandwidth exists to process debit and credit approvals. The store can likewise increase the amount of network bandwidth during periods of slower sales activity when there may not be as much contention for the bandwidth.

The Pacing Policy is also useful when replaying a Tlog or when the network has been down for an extended period of time and then comes back online. In these cases the Pacing Policy ensures that the network is not flooded with the Messages from the TLog data integration process.

The following properties can be set for each of the three time periods:

StartHour
The starting hour (0-23) for this pacing Policy Time Interval. Must be less than the value of **EndHour**

EndHour
The ending hour (0-23) for this pacing Policy Time Interval.

BytesPerSecondRate
On Average, the Pacing Policy will limit the number of bytes processed to the specified number of Bytes Per Second.

QueueDepthThreshold
The user specifies the allowable number of messages to build up on the DiskQ before adjusting the Bytes Per Second rate.

QueueDepthAdjustment
This value is a percentage that is applied to the Bytes Per Second rate when the `QueueDepthThreshold` is exceeded. The intent here is that the Pacing Policy can allow the Data Integration process to "catch up" by allowing more bytes to be processed.

Default settings

To better understand the operations of the Pacing Policy function it is helpful to look at the default settings.

Note: These default settings are for illustration purposes only.

```
#
# User can specify settings for up to 3 different hourly intervals
#
PacingPolicy.Interval1.StartHour=9
PacingPolicy.Interval1.EndHour=18
PacingPolicy.Interval1.BytesPerSecondRate=7168
PacingPolicy.Interval1.QueueDepthThreshold=15
PacingPolicy.Interval1.QueueDepthAdjustment=10
#
# Hourly Interval 2 settings
#
PacingPolicy.Interval2.StartHour=19
PacingPolicy.Interval2.EndHour=23
PacingPolicy.Interval2.BytesPerSecondRate=7168
PacingPolicy.Interval2.QueueDepthThreshold=10
PacingPolicy.Interval2.QueueDepthAdjustment=20
#
# Hourly Interval 3 settings
#
PacingPolicy.Interval3.StartHour=0
PacingPolicy.Interval3.EndHour=8
PacingPolicy.Interval3.BytesPerSecondRate=7168
PacingPolicy.Interval3.QueueDepthThreshold=5
PacingPolicy.Interval3.QueueDepthAdjustment=50
```

The defaults settings show that three intervals are specified:

- 9:00 a.m. through 6:00 p.m.
- 7:00 p.m. through midnight
- 0 hour (after midnight) through 8:00 a.m.

Interval 1: The base rate is 7168 bytes per second (56K bits per second) When more than 15 transactions are placed in the DiskQ, then the base rate is increased by 10% to allow 7885 bytes per second to be processed.

Interval 2: The base rate is 7168 bytes per second (56K bits per second) When more than 10 transactions are placed in the DiskQ, then the base rate is increased by 20% to allow 8602 bytes per second to be processed. In this time period the sales activity in the store is less and therefore the `QueueDepthThreshold` is lower and the QueueDepthAdjustment is higher.

Interval 3: The base rate is 7168 bytes per second (56K bits per second) When more than 5 transactions are placed in the DiskQ, then the base rate is increased by 50% to allow 10752 bytes per second to be processed. In this time period the sales activity in the store is the lowest and therefore the QueueDepthThreshold is lowest and the QueueDepthAdjustment is highest.

The setting for the Pacing Policy configurable properties depends on many factors such as:

- Whether Parsing and Transformation is configured to run in the store
- Available network bandwidth in the store
- Average size of a transaction
- Average rate of sale transactions
- Whether compression is enabled

Key	Description	Valid values
`DiskQService.PacingPolicy.Enabled`	Optional. Used to enable the Pacing Policy.	**true**, **false** (Default=**false**).
`PacingPolicy.Interval`*n*`.StartHour`	Starting Hour for this Pacing Policy Interval	0 - 23
`PacingPolicy.Interval`*n*`.EndHour`	Ending Hour for this Pacing Policy Interval	0 - 23
`PacingPolicy.Interval`*n*`.BytesPerSecondRate`	For this Pacing Policy Period: Bytes Per Second allocated to Tlog Data Integration.	Java integer
`PacingPolicy.Interval`*n*`.QueueDepthThreshold`	For this Pacing Policy Period: The limit of the number of messages on the DiskQ before adjusting the configured rate.	Java integer
`PacingPolicy.Interval`*n*`.QueueDepthAdjustment`	For this Pacing Policy Period: This is the percentage adjustment made to the rate when the Queue Depth threshold is exceeded.	Java integer

Note: Values for the hours, rate, threshold, and adjustment are programmatically converted to a Java `int` value. An exception is raised if the entry in the properties file cannot be converted to an integer value and an error will be

logged. In addition to the integer validation, the hour values are also validated to ensure that they are in the range of 0 through 23.

Additional pacing policy properties

Three additional Pacing Policy properties can be globally set. These properties are strictly optional and are externalized for future enhancements and to assist in the support of the product.

PacingPolicy.ClassName

Key `PacingPolicy.ClassName`

Value Any fully qualified class name that is a subclass of `DiPacingPolicyImpl` or the fully qualified class name of a new Pacing Policy that implements the `DiPacingPolicyInterface`. Default setting is `com.ibm.retail.di.service.policy.DiPacingPolicyImpl`.

This property is provided as a means for IBM and/or third parties to extend the functionality of the base Data Integration Pacing Policy. If the TLog Data Integration function moves to another messaging based system there may be additional attributes that are available to the application by the underlying message queuing system. In this case, for example, the `evaluateRules()` method of the `DIPacingPolicyImpl` could be subclassed and the support for the additional attributes available from the underlying message queuing system could be factored into the base pacing policy rule evaluation.

PacingPolicy.LogStatistics

Key `PacingPolicy.LogStatistics`

Value **True** or **false**. (Default = **false**.)

When enabled, on every hourly boundary of operation the Pacing Policy logs a set of statistics. The statistics can be useful in monitoring the operation of the Pacing Policy as well as providing information that may be useful in tuning the pacing policy with the optimum set configuration parameters. When enabled, the data is logged to a file with the following naming convention: `C:\pacing`*`lg`*`.dat`

Two levels of log information are maintained. Each level will contain up to 48 hours of log information. The example shows the file name of the most current level of logged statistics. The previous 48 hours of logged statistics (if available) are stored in the file `C:\pacing`*`lp`*`.dat`, where *lp* indicates the previous set of logged statistics.

The following statistics are logged at each hourly boundary:

Timer Ticks
Amount of time (in seconds) that the pacing policy has been active

Bytes Total bytes processed

NumMsgs
Total messages processed

MinMsgSize
Size (in bytes) of the smallest message to process

MaxMsgSize
Size (in bytes) of the largest message to process

MinAdjustedMsgSize
Size (in bytes) of the smallest message after processing

MaxAdjustedMsgSize
Size (in bytes) of the largest message after processing

MinArrivalTime
Shortest delay (in milliseconds) between arrival of messages

MaxArrivalTime
Longest delay (in milliseconds) between reception of messages

DepthOfQueueTotalCounter
Number of messages on the queue detected during all evaluations of the pacing policy rules

MaxDepthOfQueue
Greatest queue size observed

QueueDepthThreshholdExceededCounter
Number of times the Queue Depth has exceeded the Threshold

QueueDepthThreshholdWrapCount
Number of times the depth of queue counter has wrapped

DepthOfQueueWrapCount
Number of times the depth of queue total counter has wrapped

TimerTicksWrapCount
Number of times the timer ticks counter has wrapped or been reset

ByteCounterWrapCount
Number of times the byte counter has wrapped or been reset

NumMsgsCounterWrapCount
Number of times the Number of Messages counter has wrapped

The following is an example of a logged entry from the pacing log file:

```
Fri Mar 07 12:14:34 EST 2003 :
Timer Ticks: 3600
Bytes        3176906
NumMsgs      252
MinMsgSize   1761
MaxMsgSize   19959
MinAdjustedMsgSize  0
MaxAdjustedMsgSize  0
MinArrivalTime 76
MaxArrivalTime 211089
DepthOfQueueTotalCounter 1921
MaxDepthOfQueue 16
QueueDepthThreshholdExceededCounter 2
QueueDepthThreshholdWrapCount 0
DepthOfQueueWrapCount 0
TimerTicksWrapCount 0
ByteCounterWrapCount 0
NumMsgsCounterWrapCount 0
```

Explanation: The Pacing policy has been running for one hour (3600 seconds) and has processed 3,176,906 bytes. Therefore the average rate for this hour of operation is 882 bytes per second and the disk queue was never holding more than 16 messages. The queue depth threshold counter was exceeded twice. (The other values should be self-explanatory.)

Message compression

Key `DiskQService.MessageCompression`

Value **True** or **false**. (Default = **false**.)

This property must be set to **true** if compression is turned on in the underlying message transport layer (i.e., Websphere MQe compression). This setting allows the Pacing Policy to more accurately control the flow of data to the network when compression is enabled.

Note: The Data Integration process does not currently have direct access to the actual number of bytes that were sent for a message after data compression has been applied to the message. A compression factor is applied to the size of the original messages to determine the approximate size of the message that has been compressed by the underlying message transport layer. Any discrepancy between the statistics from the messaging system and the statistics reported by the Pacing Policy can be attributed to this approximation.

Appendix E. 4690 disk queue facility

The 4690 disk queue facility (DiskQ for short) provides a robust mechanism through which messages can be passed from one process to another within the 4690 store environment. Unlike with pipes, PRS pipes, and similar in-memory techniques, the messages in the disk queue persist across power failures or other traumas that interrupt processing.

The disk queue uses a disk file to store messages. On the 4690, this file enjoys the same protection and file distribution capabilities as any other file supported by the 4690 OS and its multiple controller feature. Messages in the queue are handled on a first-in-first-out (FIFO) basis. Messages are added to the end or tail of the queue. They are removed from the front or head of the queue. The user is not normally concerned with the file's internal structure, but the file is internally subdivided into fixed length elements. One or more elements are allocated to each message as the message is added to the queue. Message size is limited to the smaller of 32,767 bytes or 127 elements.

Data Integration Facility support for DiskQ

The Data Integration Facility supports disk queue functionality with its *4690 PosDiskQService*. The DiskQService reads messages from the disk queue and passes each message to an Actor (or the Director) for processing.

The disk queue is used to achieve maximum performance for the DIF tlog trickle functionality. Messages flow from one of the POS applications (SA, GSA, ACE) to the disk queue, then from the disk queue to the DIF Runtime process via a special implementation of the DiskQService, called the *PosDiskQService*. This service reads tlog transaction messages from the disk queue and assuredly delivers those messages to an enterprise messaging Actor, such as the *MqeActor* or *JmsActor*. Optionally, the transactions flow through a data transformer before being transmitted.

Note: PosDiskQService requires the `diskq` file to be defined by the `DIFQUEUE` logical name. Refer to the DiskQService documentation for information about how to customize the service's behavior through subclassing.

Comparing 4690 disk queue and Websphere MQSeries Everyplace

Both of these solutions provide disk-based storage for persistent messaging on the 4690. However, these products are targeted to solve different problems:

- Use the 4690 disk queue for high-performance assured message exchange between applications running in the store, especially C and CBASIC applications
- Use Websphere MQe for exchanging assured messages with enterprise systems, or for exchanging messages with Java-based applications in the store

Programming APIs

API's for manipulating messages on a queue are available for BASIC (on 4690 systems) and C (on 4690, AIX, and NT systems). C programs using the C API have access to more functions. The 4690 BASIC API contains sufficient function for 4690 BASIC applications to add messages to a queue. The BASIC API does not support removing messages from a queue.

The disk queue can only be used in combination with the DIF tlog trickle function. Documentation and support for custom projects using the disk queue are not available as part of the DIF product. However, support can be purchased from the IBM National Retail Services Center (NRSC).

Disk queue maintenance and test programs

Several utility programs are available to display and manipulate disk queues. On the 4690 these utilities have names in the form `DQPROG.286`. On NT systems the same utilities have names in the form dqprog.exe and on AIX systems they have names in the form `dqprog`. They are all referred to in the form `DQPROG` in the descriptions below.

Each of these utilities will display information describing the last time the utility was updated and the version of the disk queue support used. This information can be useful in investigating problems. It is displayed in the following format:

```
version 08/17/1999
using dq version V.04/20/2000
```

DQCREATE - Disk Queue Creation Utility

DQCREATE is used to create a disk queue file. It accepts the parameters used in defining the disk queue and generates the initial empty queue file.

Note: The Data Integration Facility (DIF) provides an automated tool for creating a disk queue when using the tlog trickle application. Please see Chapter 2, “Configuration and operation”.

DQCREATE’s operation is modified by the following command line parameters:

`DQCREATE -q` *`difqueue`* `-s` *`nn`* `-n` *`nnnn`* `[-x` *`nnnn`*`] [-X` *`nn`*`] [-t] [-N]`

Parameter	Description	Default value
-q *difqueue*	Specifies the name of the queue file to be created.	None
-s *nn*	*nn* specifies the size (in bytes) of the elements on the queue. (Each message on the queue requires one or more elements, depending on the element and message sizes.)	None
-n *nnnn*	*nnnn* specifies the number of elements to be allocated in the queue.	None
-x *nnnn*	Specifies that, if the queue becomes full, it can be automatically extended by adding room for *nnnn* more elements. This option is used in conjunction with -X below.	0
-X *nn*	specifies that, if the queue becomes full, it can be automatically extended *nn* times. This option is used in conjunction with -x above.	0

Parameter	Description	Default value
-N	Specifies that elements on the queue are to be filled with nulls as they are removed.	Elements are not filled with nulls

When creating a queue, it is important to consider the size messages expected to be sent via the queue. Each message will require at least one element on the queue. If the message is larger than the element size, as many elements will be used as are required to contain the entire message. The maximum size of a message is limited to 127 elements or 32,767 bytes, whichever is smaller.

In choosing an element size, a trade off is made between wasting space because the elements are larger than most messages and excessive message segmentation because most messages are larger than the element size. For example, assume most messages are 100 bytes or less but some messages are 200 bytes. If we choose an element size of 100, most messages will fit in one element, but a few will require two. If we choose an element size of 200, all messages will fit in a single element, but most will waste at least 100 bytes of the element.

The amount of message segmenting anticipated should also be considered in choosing the number of elements to be allocated for the queue.

DQPEEK - Display the next message on the queue

DQPEEK can be used to display the messages on a queue. The messages are not removed from the queue.

DQPEEK's operation is specified by the following command line parameters:

`DQPEEK -q difqueue [-v] [-d]`

Parameter	Description	Default Value
-q *difqueue*	Specifies the name of the queue from which a message is to be displayed	None
-v	Specifies that contents of messages are to be displayed	Status messages are not displayed
-d	Specifies that debug messages are to be displayed	Debug messages are not displayed

DQLIST - Display all of the messages on the Queue

DQLIST can be used to display the messages on a queue. The messages are not removed from the queue.

DQLIST's operation is specified by the following command line parameters:

`DQLIST -q difqueue [-v] [-d]`

Parameter	Description	Default value
-q *difqueue*	Specifies the name of the queue from which a messages are to be displayed.	None
-v	Specifies that status messages are to be displayed	Status messages are not displayed
-d	Specifies that debug messages are to be displayed	Debug messages are not displayed

DQREMOVE - Remove the next message from the queue

DQREMOVE can be used to remove the next message on a queue.

DQREMOVE's operation is specified by the following command line parameters:

```
DQREMOVE -q difqueue [-v] [-d]
```

Parameter	Description	Default value
-q *difqueue*	Specifies the name of the queue from which a message is to be removed.	None
-v	Specifies that status messages are to be displayed	Status messages are not displayed
-d	Specifies that debug messages are to be displayed	Debug messages are not displayed

DQRESET - Resets a queue to an empty state

CAUTION:
DQRESET will cause any data currently on the queue to be lost.

DQRESET can be used to reset a queue to an empty state. It updates the control block in the queue to indicate that there are no messages on the queue. It does not null or "blank out" the existing messages, but they will be overwritten as new messages are added to the queue. This utility can be useful if it is necessary to discard the contents of a queue without the time or overhead associated with recreating the queue. It is not necessary to stop other processes accessing the queue when DQRESET is run.

DQRESET's operation is specified by the following command line parameters:

```
DQRESET -q difqueue [-v] [-d]
```

Parameter	Description	Default value
-q *difqueue*	Specifies the name of the queue that is to be reset.	None
-v	Specifies that status messages are to be displayed	Status messages are not displayed

Parameter	Description	Default value
-d	Specifies that debug messages are to be displayed	Debug messages are not displayed

DQSTATUS - Display the status of a queue

DQSTATUS can be used to display the current status of a queue.

DQSTATUS's operation is specified by the following command line parameters:

`DQSTATUS -q` *`difqueue`* `[-v] [-d]`

Parameter	Description	Default value
-q *difqueue*	Specifies the name of the queue for which the status is to be displayed	None
-v	Specifies that status messages are to be displayed	Status messages are not displayed
-d	Specifies that debug messages are to be displayed	Debug messages are not displayed

The queue's status is reported as follows:

```
version 08/17/1999
using dq version V.04/20/2000
------------------------------------------------------------
DISK QUEUE NAME                 = DIFQUEUE
------------------------------------------------------------
Created or reset at             = 18:11:23 on 08/27/2003
Last modified at                = 14:10:54 on 10/26/2003
Percent full                    = 0 percent
Current element count           = 0
Current message count           = 0
Element Size                    = 128
Element # for next add          = 1
Element # for next get          = 0
Initial Queue Size              = 100
Current working Max element cnt = 100
Max Number of extends allowed   = 5
Remaining extensions allowed    = 5
Number of elements in extension = 1000
Null element on remove          = OFF
------------------------------------------------------------
```

Readers' Comments — We'd Like to Hear from You

Store Integration Framework
Data Integration Facility
Programming Guide

Publication No. GA27-4309-01

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

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

Name _____ Address _____

Company or Organization _____

Phone No. _____

Readers' Comments — We'd Like to Hear from You
GA27-4309-01

IBM

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Design & Information Development
Dept. CJMA/Bldg. 645
PO Box 12195
Research Triangle Park, NC 27709-9990

Fold and Tape | **Please do not staple** | Fold and Tape

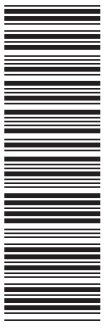
OBSOLETE AFTER - 12/31/2003

Cut or Fold
Along Line

Part Number: 03R5994

August 7, 2003
Printed in U.S.A.

(1P) P/N: 03R5994



GA27-4309-01