DB2 IBM Informix

IBM

Version 10.0

IBM Informix Built-In DataBlade Modules User's Guide

G251-2770-00

DB2 IBM Informix

IBM

Version 10.0

IBM Informix Built-In DataBlade Modules User's Guide

G251-2770-00

Note:
Before using this information and the product it supports, read the information in "Notices" on page B-1.

First Edition (July 2005)

This document contains proprietary information of IBM. It is provided under a license agreement and is protected by copyright law. The information contained in this publication does not include any product warranties, and any statements provided in this manual should not be interpreted as such.

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

© Copyright International Business Machines Corporation 2005. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Introduction vii
In This Introduction vii
About This Guide vii
Organization of This Guide vii
Types of Users viii
Documentation Conventions viii
Typographical Conventions viii
Feature, Product, and Platform Markup ix
Syntax Diagrams ix
Example Code Conventions xii
Additional Documentation xiii
IBM Informix Information Center xiii
Installation Guides xiii
Online Notes xiii
Informix Error Messages xv
Manuals xv
Online Help xvi
Accessibility xvi
IBM Informix Dynamic Server Version 10.0 and CSDK Version 2.90 Documentation Set xvi
Compliance with Industry Standards xviii
IBM Welcomes Your Comments xviii

Part 1. Large Object Locator DataBlade Module

Chapter 1. About Large Object Locator 1-1
In This Chapter 1-1
Using Large Object Locator 1-2
Large Object Locator Data Types 1-2
Large Object Locator Functions 1-2
Limitations 1-3
Installation and Registration 1-3

Chapter 2. Data Types 2-1
In This Chapter 2-1
lld_locator 2-1
lld_lob 2-2

Chapter 3. Functions 3-1
In This Chapter 3-1
Interfaces 3-1
API Library 3-2
ESQL/C Library 3-2
SQL Interface 3-2
Working with Large Objects 3-2
lld_close() 3-4
lld_copy() 3-5
lld_create() 3-7
lld_delete() 3-9
lld_open() 3-10
lld_read() 3-12
lld_seek() 3-13
lld_tell() 3-15
lld_write() 3-16
Client File Support 3-17

© Copyright IBM Corp. 2005

3 Examples 8-28
3 MQVersion() 8-28
3 Syntax 8-28
3 Example 8-28

3 **Chapter 9. MQ DataBlade Module Error Handling 9-1**
3 In This Chapter 9-1
3 Error Codes 9-1

Part 3. Appendixes

Appendix. Accessibility A-1

Notices B-1

Index X-1

Introduction

In This Introduction vii
About This Guide vii
Organization of This Guide vii
Types of Users viii
Documentation Conventions viii
Typographical Conventions viii
Feature, Product, and Platform Markup ix
Syntax Diagrams ix
How to Read a Command-Line Syntax Diagram xi
Keywords and Punctuation xii
Identifiers and Names xii
Example Code Conventions xii
Additional Documentation xiii
IBM Informix Information Center xiii
Installation Guides xiii
Online Notes xiii
Locating Online Notes xiv
Online Notes Filenames xv
Informix Error Messages xv
Manuals xv
Online Manuals xv
Printed Manuals xvi
Online Help xvi
Accessibility xvi
IBM Informix Dynamic Server Version 10.0 and CSDK Version 2.90 Documentation Set xvi
Compliance with Industry Standards xviii
IBM Welcomes Your Comments xviii

In This Introduction

This chapter introduces the *IBM Informix: Built-In DataBlade Modules User's Guide*, which replaces the *IBM Informix: Large Object Locator DataBlade Module User's Guide*. The *IBM Informix: Built-In DataBlade Modules User's Guide* incorporates all of the chapters from the *IBM Informix: Large Object Locator DataBlade Module User's Guide* and includes new chapters on how to use the IBM Informix MQ DataBlade module. Read this chapter for an overview of the information provided in this manual and for an understanding of the conventions used throughout.

About This Guide

This guide explains how to use the IBM Informix Large Object Locator DataBlade module and the IBM Informix MQ DataBlade module.

Organization of This Guide

The *IBM Informix: Built-In DataBlade Modules User's Guide* includes the following chapters:

- This introduction provides an overview of the contents of the guide, describes documentation conventions used, and lists additional books to supplement the information in the *IBM Informix: Built-In DataBlade Modules User's Guide*.
- Part 1: The Large Object Locator DataBlade module:

© Copyright IBM Corp. 2005

- Chapter 1, “About Large Object Locator,” on page 1-1, explains the basic concepts and operations of the IBM Informix Large Object Locator DataBlade module.
- Chapter 2, “Data Types,” on page 2-1, describes the Large Object Locator data types.
- Chapter 3, “Functions,” on page 3-1, provides a complete reference to the Large Object Locator functions.
- Chapter 4, “Example Code,” on page 4-1, provides example code that explains how to use the Large Object Locator functions and data types.
- Chapter 5, “Error Handling,” on page 5-1, describes how to handle errors that can be returned when using the Large Object Locator functions.

• Part 2: The MQ DataBlade module:

- Chapter 6, “About the MQ DataBlade Module,” on page 6-1 describes how to install, verify and use the MQ DataBlade module
- Chapter 7, “MQ DataBlade Tables,” on page 7-1 describes how to use the MQ DataBlade tables.
- Chapter 8, “MQ DataBlade Functions,” on page 8-1 describes how to use the MQ DataBlade Functions.
- Chapter 9, “MQ DataBlade Module Error Handling,” on page 9-1 provides information on MQ DataBlade error codes.

• An Accessibility appendix describes how to read syntax diagrams in the HTML version of this manual using a screen reader.
• A Notices appendix describes IBM products, features, and services.
• An index directs you to areas of particular interest.

Types of Users

Large Object Locator is intended for database administrators who want to add Large Object Locator functionality to their databases and for developers who want to add Large Object Locator functionality to their applications.

MQ DataBlade module is intended for users who want enable their IBM Informix database applications to communicate with other MQ applications.

Documentation Conventions

This section describes the conventions that this manual uses. These conventions make it easier to gather information from this and other volumes in the documentation set.

The following conventions are discussed:

• Typographical conventions
• Other conventions
• Syntax diagrams
• Command-line conventions
• Example code conventions

Typographical Conventions

This manual uses the following conventions to introduce new terms, illustrate screen displays, describe command syntax, and so forth.

Convention	Meaning
KEYWORD	Keywords of SQL, SPL, and some other programming languages appear in uppercase letters in a serif font.
italics	Within text, new terms and emphasized words appear in italics. Within syntax and code examples, variable values that you are to specify appear in italics.
boldface	Names of program entities (such as classes, events, and tables), environment variables, file and pathnames, and interface elements (such as icons, menu items, and buttons) appear in boldface.
`monospace`	Information that the product displays and information that you enter appear in a monospace typeface.
`KEYSTROKE`	Keys that you are to press appear in uppercase letters in a sans serif font.
>	This symbol indicates a menu item. For example, "Choose **Tools > Options**" means choose the **Options** item from the **Tools** menu.

Tip: When you are instructed to "enter" characters or to "execute" a command, immediately press `RETURN` after the entry. When you are instructed to "type" the text or to "press" other keys, no `RETURN` is required.

Feature, Product, and Platform Markup

Feature, product, and platform markup identifies paragraphs that contain feature-specific, product-specific, or platform-specific information. Some examples of this markup follow:

Dynamic Server

Identifies information that is specific to IBM Informix Dynamic Server

End of Dynamic Server

Windows Only

Identifies information that is specific to the Windows environment

End of Windows Only

This markup can apply to one or more paragraphs within a section. When an entire section applies to a particular product or platform, this is noted as part of the heading text, for example:

Table Sorting (Linux)

Syntax Diagrams

This guide uses syntax diagrams built with the following components to describe the syntax for statements and all commands other than system-level commands.

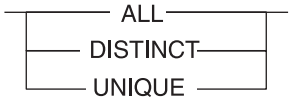
Syntax diagrams depicting SQL and command-line statements have changed in the following ways:

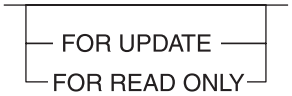
- The symbols at the beginning and end of statements are double arrows.
- The symbols at the beginning and end of syntax segment diagrams are vertical lines.

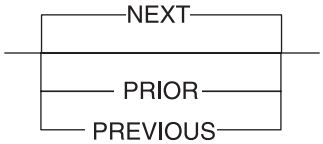
- How many times a loop can be repeated is explained in a diagram footnote, whose marker appears above the path that is describes.
- Syntax statements that are longer than one line continue on the next line.
- Product or condition-specific paths are explained in diagram footnotes, whose markers appear above the path that they describe.
- Cross-references to the descriptions of other syntax segments appear as diagram footnotes, whose markers immediately follow the name of the segment that they reference.

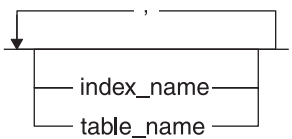
The following table describes syntax diagram components.

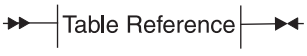
Component represented in PDF	Component represented in HTML	Meaning
	`>>----------`	Statement begins.
	`---------->`	Statement continues on next line.
	`>----------`	Statement continues from previous line.
	`----------><`	Statement ends.
	`--------SELECT----------`	Required item.
	`--+-----------------+---` `  '------LOCAL------'`	Optional item.

	`---+-----ALL-------+---`
	`   +--DISTINCT-----+`
	`   '---UNIQUE------'`
		Required item with choice. One and only one item must be present.

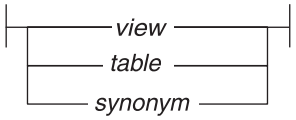
	`---+------------------+---`
	`    +--FOR UPDATE-----+`
	`    '--FOR READ ONLY--'`
		Optional items with choice are shown below the main line, one of which you might specify.

	`    .---NEXT---------.`
	`----+----------------+---`
	`    +---PRIOR--------+`
	`    '---PREVIOUS-----'`
		The values below the main line are optional, one of which you might specify. If you do not specify an item, the value above the line will be used as the default.

	`  .-------,-----------.`
	`  V                  |`
	`---+-----------------+---`
	`    +---index_name---+`
	`    '---table_name---'`
		Optional items. Several items are allowed; a comma must precede each repetition.

	`>>-| Table Reference |-><`	Reference to a syntax segment.

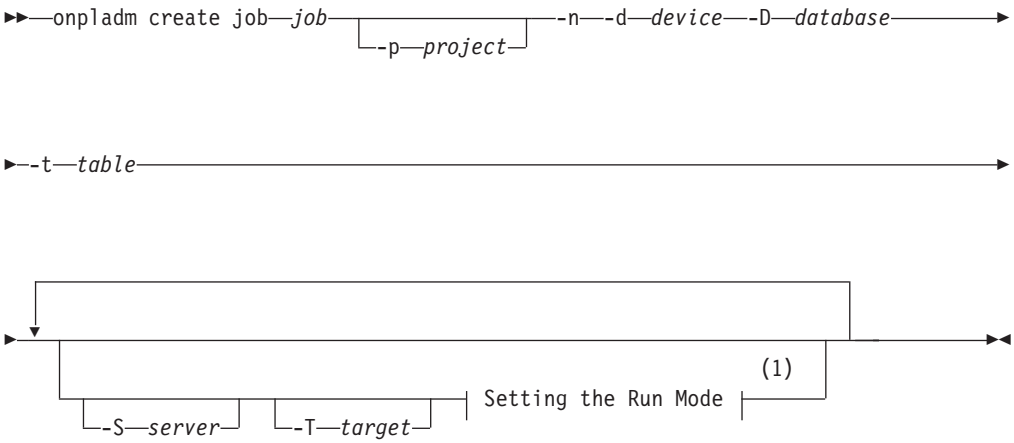
Component represented in PDF	Component represented in HTML	Meaning

Table Reference
 
	Table Reference
	`|--+-----view--------+--|`
	`  +------table------+`
	`  '----synonym------'`
		Syntax segment.

How to Read a Command-Line Syntax Diagram

The following command-line syntax diagram uses some of the elements listed in the table in the previous section.

Creating a No-Conversion Job

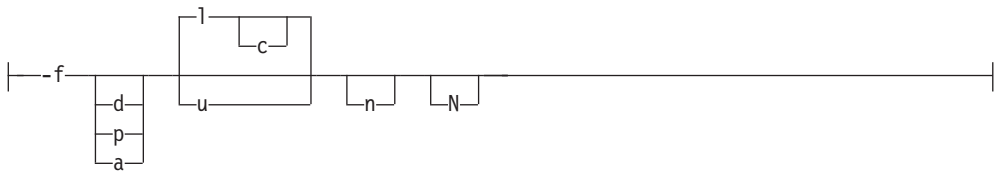


Notes:

1 See page Z-1

The second line in this diagram has a segment named “Setting the Run Mode,” which according to the diagram footnote, is on page Z-1. If this was an actual cross-reference, you would find this segment in on the first page of Appendix Z. Instead, this segment is shown in the following segment diagram. Notice that the diagram uses segment start and end components.

Setting the Run Mode:



To see how to construct a command correctly, start at the top left of the main diagram. Follow the diagram to the right, including the elements that you want. The elements in this diagram are case sensitive because the illustrates utility syntax. Other types of syntax, such as SQL, are not case sensitive.

The Creating a No-Conversion Job diagram illustrates the following steps:

1. Type **onpladm create job** and then the name of the job.
2. Optionally, type **-p** and then the name of the project.
3. Type the following required elements:
   - **-n**
   - **-d** and the name of the device
   - **-D** and the name of the database
   - **-t** and the name of the table
4. Optionally, you can choose one or more of the following elements and repeat them an arbitrary number of times:
   - **-S** and the server name
   - **-T** and the target server name
   - The run mode. To set the run mode, follow the Setting the Run Mode segment diagram to type **-f**, optionally type **d**, **p**, or **a**, and then optionally type **l** or **u**.
5. Follow the diagram to the terminator.

Your diagram is complete.

Keywords and Punctuation

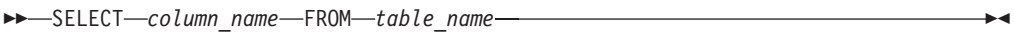
Keywords are words reserved for statements and all commands except system-level commands. When a keyword appears in a syntax diagram, it is shown in uppercase letters. When you use a keyword in a command, you can write it in uppercase or lowercase letters, but you must spell the keyword exactly as it appears in the syntax diagram.

You must also use any punctuation in your statements and commands exactly as shown in the syntax diagrams.

Identifiers and Names

Variables serve as placeholders for identifiers and names in the syntax diagrams and examples. You can replace a variable with an arbitrary name, identifier, or literal, depending on the context. Variables are also used to represent complex syntax elements that are expanded in additional syntax diagrams. When a variable appears in a syntax diagram, an example, or text, it is shown in *lowercase italic*.

The following syntax diagram uses variables to illustrate the general form of a simple SELECT statement.



When you write a SELECT statement of this form, you replace the variables *column_name* and *table_name* with the name of a specific column and table.

Example Code Conventions

Examples of SQL code occur throughout this manual. Except as noted, the code is not specific to any single IBM Informix application development tool.

If only SQL statements are listed in the example, they are not delimited by semicolons. For instance, you might see the code in the following example:

```
CONNECT TO stores_demo
...
```

```
DELETE FROM customer
   WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```

To use this SQL code for a specific product, you must apply the syntax rules for that product. For example, if you are using DB–Access, you must delimit multiple statements with semicolons. If you are using an SQL API, you must use EXEC SQL at the start of each statement and a semicolon (or other appropriate delimiter) at the end of the statement.

Tip: Ellipsis points in a code example indicate that more code would be added in a full application, but it is not necessary to show it to describe the concept being discussed.

For detailed directions on using SQL statements for a particular application development tool or SQL API, see the manual for your product.

Additional Documentation

For additional information, refer to the following types of documentation:

- Installation guides
- Online notes
- Informix error messages
- Manuals
- Online help

IBM Informix Information Center

The Informix Dynamic Server Information Center integrates the entire IBM Informix Dynamic Server 10.0 and IBM Informix Client SDK (CSDK) 2.90 documentation sets in both HTML and PDF formats. The Information Center provides full text search, a master index, logical categories, easy navigation, and links to troubleshooting and support files.

The IBM Informix Information Center site is located at http://publib.boulder.ibm.com/infocenter/ids9help/index.jsp.

Installation Guides

Installation guides are located in the **/doc** directory of the product CD or in the **/doc** directory of the product's compressed file if you downloaded it from the IBM Web site. Alternatively, you can obtain installation guides from the IBM Informix Online Documentation site at http://www.ibm.com/software/data/informix/pubs/library/ or the IBM Informix Information Center at http://publib.boulder.ibm.com/infocenter/ids9help/index.jsp.

Online Notes

The following sections describe the online files that supplement the information in this manual. Please examine these files before you begin using your IBM Informix product. They contain vital information about application and performance issues.

Online File	Description	Format
TOC Notes	The TOC (Table of Contents) notes file provides a comprehensive directory of hyperlinks to the release notes, the fixed and known defects file, and all the documentation notes files for individual manual titles.	HTML
Documentation Notes	The documentation notes file for each manual contains important information and corrections that supplement the information in the manual or information that was modified since publication.	HTML, text
Release Notes	The release notes file describes feature differences from earlier versions of IBM Informix products and how these differences might affect current products. For some products, this file also contains information about any known problems and their workarounds.	HTML, text
Machine Notes	(Non-Windows platforms only) The machine notes file describes any platform-specific actions that you must take to configure and use IBM Informix products on your computer.	text
Fixed and Known Defects File	This text file lists issues that have been identified with the current version. It also lists customer-reported defects that have been fixed in both the current version and in previous versions.	text

Locating Online Notes

Online notes are available from the IBM Informix Online Documentation site at http://www.ibm.com/software/data/informix/pubs/library/ and in the IBM Informix Information Center at http://publib.boulder.ibm.com/infocenter/ids9help/index.jsp. Additionally you can locate these files before or after installation as described below.

Before Installation

All online notes are located in the **/doc** directory of the product CD. The easiest way to access the documentation notes, the release notes, and the fixed and known defects file is through the hyperlinks from the TOC notes file.

The machine notes file and the fixed and known defects file are only provided in text format.

After Installation

On UNIX platforms in the default locale, the documentation notes, release notes, and machine notes files appear under the **$INFORMIXDIR/release/en_us/0333** directory.

Dynamic Server

On Windows the documentation and release notes files appear in the **Informix** folder. To display this folder, choose **Start > Programs > IBM** ***product name version*** **> Documentation Notes** or **Release Notes** from the taskbar.

Machine notes do not apply to Windows platforms.

End of Dynamic Server

Online Notes Filenames

Online notes have the following file formats:

Online File	File Format	Examples
TOC Notes	*prod_os*_toc_*version*.html	**ids_win_toc_10.0.html**
Documentation Notes	*prod_bookname*_docnotes_*version*.html/txt	**ids_hpl_docnotes_10.0.html**
Release Notes	*prod_os*_relnotes_*version*.html/txt	**ids_unix_relnotes_10.0.txt**
Machine Notes	*prod*_machine_notes_*version*.txt	**ids_machine_notes_10.0.txt**
Fixed and Known Defects File	*prod*_defects_*version*.txt ids_win_fixed_and_known _defects_*version*.txt	**ids_defects_10.0.txt** **client_defects_2.90.txt** **ids_win_fixed_and_known _defects_10.0.txt**

Informix Error Messages

This file is a comprehensive index of error messages and their corrective actions for the Informix products and version numbers.

On UNIX platforms, use the **finderr** command to read the error messages and their corrective actions.

Dynamic Server

On Windows, use the Informix Error Messages utility to read error messages and their corrective actions. To display this utility, choose **Start > Programs > IBM *product name version* > Informix Error Messages** from the taskbar.

End of Dynamic Server

You can also access these files from the IBM Informix Online Documentation site at http://www.ibm.com/software/data/informix/pubs/library/ or in the IBM Informix Information Center at http://publib.boulder.ibm.com/infocenter/ids9help/index.jsp.

Manuals

Online Manuals

A CD that contains your manuals in electronic format is provided with your IBM Informix products. You can install the documentation or access it directly from the CD. For information about how to install, read, and print online manuals, see the installation insert that accompanies your CD. You can also obtain the same online manuals from the IBM Informix Online Documentation site at http://www.ibm.com/software/data/informix/pubs/library/ or in the IBM Informix Information Center at http://publib.boulder.ibm.com/infocenter/ids9help/index.jsp.

Printed Manuals

To order hardcopy manuals, contact your sales representative or visit the IBM Publications Center Web site at http://www.ibm.com/software/howtobuy/data.html.

Online Help

IBM Informix online help, provided with each graphical user interface (GUI), displays information about those interfaces and the functions that they perform. Use the help facilities that each GUI provides to display the online help.

Accessibility

IBM is committed to making our documentation accessible to persons with disabilities. Our books are available in HTML format so that they can be accessed with assistive technology such as screen reader software. The syntax diagrams in our manuals are available in dotted decimal format, which is an accessible format that is available only if you are using a screen reader. For more information about the dotted decimal format, see the Accessibility appendix.

IBM Informix Dynamic Server Version 10.0 and CSDK Version 2.90 Documentation Set

The following tables list the manuals that are part of the IBM Informix Dynamic Server, Version 10.0 and the CSDK Version 2.90, documentation set. PDF and HTML versions of these manuals are available at http://www.ibm.com/software/data/informix/pubs/library/ or in the IBM Informix Information Center at http://publib.boulder.ibm.com/infocenter/ids9help/index.jsp. You can order hardcopy versions of these manuals from the IBM Publications Center at http://www.ibm.com/software/howtobuy/data.html.

Table 1. Database Server Manuals

Manual	Subject
Administrator's Guide	Understanding, configuring, and administering your database server.
Administrator's Reference	Reference material for Informix Dynamic Server, such as the syntax of database server utilities **onmode** and **onstat**, and descriptions of configuration parameters, the **sysmasters** tables, and logical-log records.
Backup and Restore Guide	The concepts and methods you need to understand when you use the **ON-Bar** and **ontape** utilities to back up and restore data.
Built-In DataBlade Modules User's Guide	Using the following DataBlade modules that are included with Dynamic Server: • MQ DataBlade module, to allow IBM Informix database applications to communicate with other MQSeries applications. • Large Object Locator, a foundation DataBlade module that can be used by other modules that create or store large-object data.
DB-Access User's Guide	Using the **DB-Access** utility to access, modify, and retrieve data from Informix databases.
DataBlade API Function Reference	The DataBlade API functions and the subset of ESQL/C functions that the DataBlade API supports. You can use the DataBlade API to develop client LIBMI applications and C user-defined routines that access data in Informix databases.
DataBlade API Programmer's Guide	The DataBlade API, which is the C-language application-programming interface provided with Dynamic Server. You use the DataBlade API to develop client and server applications that access data stored in Informix databases.

Table 1. Database Server Manuals (continued)

Manual	Subject
Database Design and Implementation Guide	Designing, implementing, and managing your Informix databases.
Enterprise Replication Guide	How to design, implement, and manage an Enterprise Replication system to replicate data between multiple database servers.
Error Messages file	Causes and solutions for numbered error messages you might receive when you work with IBM Informix products.
Getting Started Guide	Describes the products bundled with IBM Informix Dynamic Server and interoperability with other IBM products. Summarizes important features of Dynamic Server and the new features for each version.
Guide to SQL: Reference	Information about Informix databases, data types, system catalog tables, environment variables, and the stores_demo demonstration database.
Guide to SQL: Syntax	Detailed descriptions of the syntax for all Informix SQL and SPL statements.
Guide to SQL: Tutorial	A tutorial on SQL, as implemented by Informix products, that describes the basic ideas and terms that are used when you work with a relational database.
High-Performance Loader User's Guide	Accessing and using the High-Performance Loader (HPL), to load and unload large quantities of data to and from Informix databases.
Installation Guide for Microsoft Windows	Instructions for installing IBM Informix Dynamic Server on Windows.
Installation Guide for UNIX and Linux	Instructions for installing IBM Informix Dynamic Server on UNIX and Linux.
J/Foundation Developer's Guide	Writing user-defined routines (UDRs) in the Java programming language for Informix Dynamic Server with J/Foundation.
Migration Guide	Conversion to and reversion from the latest versions of Informix database servers. Migration between different Informix database servers.
Optical Subsystem Guide	The Optical Subsystem, a utility that supports the storage of BYTE and TEXT data on optical disk.
Performance Guide	Configuring and operating IBM Informix Dynamic Server to achieve optimum performance.
R-Tree Index User's Guide	Creating R-tree indexes on appropriate data types, creating new operator classes that use the R-tree access method, and managing databases that use the R-tree secondary access method.
SNMP Subagent Guide	The IBM Informix subagent that allows a Simple Network Management Protocol (SNMP) network manager to monitor the status of Informix servers.
Storage Manager Administrator's Guide	Informix Storage Manager (ISM), which manages storage devices and media for your Informix database server.
Trusted Facility Guide	The secure-auditing capabilities of Dynamic Server, including the creation and maintenance of audit logs.
User-Defined Routines and Data Types Developer's Guide	How to define new data types and enable user-defined routines (UDRs) to extend IBM Informix Dynamic Server.
Virtual-Index Interface Programmer's Guide	Creating a secondary access method (index) with the Virtual-Index Interface (VII) to extend the built-in indexing schemes of IBM Informix Dynamic Server. Typically used with a DataBlade module.
Virtual-Table Interface Programmer's Guide	Creating a primary access method with the Virtual-Table Interface (VTI) so that users have a single SQL interface to Informix tables and to data that does not conform to the storage scheme of Informix Dynamic Server.

Table 2. Client/Connectivity Manuals

Manual	Subject
Client Products Installation Guide	Installing IBM Informix Client Software Developer's Kit (Client SDK) and IBM Informix Connect on computers that use UNIX, Linux, and Windows.
Embedded SQLJ User's Guide	Using IBM Informix Embedded SQLJ to embed SQL statements in Java programs.
ESQL/C Programmer's Manual	The IBM Informix implementation of embedded SQL for C.
GLS User's Guide	The Global Language Support (GLS) feature, which allows IBM Informix APIs and database servers to handle different languages, cultural conventions, and code sets.
JDBC Driver Programmer's Guide	Installing and using Informix JDBC Driver to connect to an Informix database from within a Java application or applet.
.NET Provider Reference Guide	Using Informix .NET Provider to enable .NET client applications to access and manipulate data in Informix databases.
ODBC Driver Programmer's Manual	Using the Informix ODBC Driver API to access an Informix database and interact with the Informix database server.
OLE DB Provider Programmer's Guide	Installing and configuring Informix OLE DB Provider to enable client applications, such as ActiveX Data Object (ADO) applications and Web pages, to access data on an Informix server.
Object Interface for C++ Programmer's Guide	The architecture of the C++ object interface and a complete class reference.

Table 3. DataBlade Developer's Kit Manuals

Manual	Subject
DataBlade Developer's Kit User's Guide	Developing and packaging DataBlade modules using BladeSmith and BladePack.
DataBlade Module Development Overview	Basic orientation for developing DataBlade modules. Includes an example illustrating the development of a DataBlade module.
DataBlade Module Installation and Registration Guide	Installing DataBlade modules and using BladeManager to manage DataBlade modules in Informix databases.

Compliance with Industry Standards

The American National Standards Institute (ANSI) and the International Organization of Standardization (ISO) have jointly established a set of industry standards for the Structured Query Language (SQL). IBM Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992. In addition, many features of IBM Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL Common Applications Environment (CAE) standards.

IBM Welcomes Your Comments

We want to know about any corrections or clarifications that you would find useful in our manuals, which will help us improve future versions. Include the following information:

- The name and version of the manual that you are using
- Section and page number
- Your suggestions about the manual

Send your comments to us at the following email address:

docinf@us.ibm.com

This email address is reserved for reporting errors and omissions in our documentation. For immediate help with a technical problem, contact IBM Technical Support. For instructions, see the IBM Informix Technical Support website at http://www-306.ibm.com/software/data/informix/support/contact.html.

We appreciate your suggestions.

Part 1. Large Object Locator DataBlade Module

© Copyright IBM Corp. 2005

Chapter 1. About Large Object Locator

In This Chapter 1-1
Using Large Object Locator 1-2
Large Object Locator Data Types 1-2
Large Object Locator Functions 1-2
Limitations 1-3
Transaction Rollback 1-3
Concurrent Access 1-3
Installation and Registration 1-3

In This Chapter

This chapter provides an overview of the IBM Informix Large Object Locator DataBlade module.

Large Object Locator enables you to create a single consistent interface to large objects. It extends the concept of large objects to include data stored outside the database.

IBM Informix Dynamic Server stores large object data (data that exceeds a length of 255 bytes or contains non-ASCII characters) in columns in the database. You can access this data using standard SQL statements. The server also provides functions for copying data between large object columns and files. See *IBM Informix: Guide to SQL Syntax* and *IBM Informix: Guide to SQL Tutorial* for more information.

With Large Object Locator you create a reference to a large object and store the reference as a row in the database. The object itself can reside outside the database: for example, on a file system (or it could be a BLOB or CLOB type column in the database). The reference identifies the type, or access protocol, of the object and points to its storage location. For example, you could identify an object as a file and provide a pathname to it or identify it as a binary or character smart large object stored in the database. Smart large objects are a category of large objects, including BLOB and CLOB, that store text and images, are stored and retrieved in pieces, and have database properties such as crash recovery and transaction rollback.

You access a large object by passing its reference to a Large Object Locator function. For example, to open a large object for reading or writing, you pass the object's reference to the **lld_open()** function. This function uses the reference to find the location of the object and to identify its type. Based on the type, it calls the appropriate underlying function to open the object. For example, if the object is stored on a UNIX file system, **lld_open()** calls a UNIX function to open the object.

Important: In theory, you could use Large Object Locator to reference any type of large object in any storage location. In practice, access protocols must be built into Large Object Locator for each type of supported object. Because support for new types can be added at any time, be sure to read the release notes accompanying this manual—not the manual itself—to see the types of large objects Large Object Locator currently supports.

© Copyright IBM Corp. 2005

Using Large Object Locator

Large Object Locator is implemented through two data types and a set of functions, described next.

Large Object Locator Data Types

Large Object Locator defines two data types, lld_locator and lld_lob.

You use the lld_locator type to identify the access protocol for a large object and to point to its location. This type is a row type, stored as a row in the database. You can insert, select, delete, and update instances of lld_locator rows in the database using standard SQL INSERT, SELECT, DELETE, and UPDATE statements.

You can also pass an lld_locator row to various Large Object Locator functions. For example, to create, delete, or copy a large object, and to open a large object for reading or writing, you pass an lld_locator row to the appropriate Large Object Locator function. See "lld_locator" on page 2-1 for a detailed description of this data type.

The lld_lob type enables Large Object Locator to reference smart large objects, which are stored as BLOB or CLOB data in the database. The lld_lob type is identical to the BLOB and CLOB types except that, in addition to pointing to the data, it tracks whether the underlying smart large object contains binary or character data.

See "lld_lob" on page 2-2 for a complete description of this data type.

Large Object Locator Functions

Large Object Locator provides a set of functions similar to UNIX I/O functions for manipulating large objects. You use the same functions regardless of how or where the underlying large object is stored.

The Large Object Locator functions can be divided into four main categories:

- **Basic functions** for creating, opening, closing, deleting, and reading from and writing to large objects.
- **Client functions** for creating, opening, and deleting client files and for copying large objects to and from client files. After you open a client file, you can use the basic functions to read from and write to the file.
- **Utility functions** for raising errors and converting errors to their SQL state equivalents.
- **Smart large object functions** for copying smart large objects to files and to other smart large objects.

There are three interfaces to the Large Object Locator functions:

- An API library
- An ESQL/C library
- An SQL interface

All Large Object Locator functions are implemented as API library functions. You can call Large Object Locator functions from user-defined routines within an application you build.

All Large Object Locator functions, except **lld_error_raise()**, are implemented as ESQL/C functions. You can use the Large Object Locator functions to build ESQL/C applications.

A limited set of the Large Object Locator functions are implemented as user-defined routines that you can execute within SQL statements. See "SQL Interface" on page 3-2 for a list of the Large Object Locator functions that you can execute directly in SQL statements.

Chapter 3, "Functions," on page 3-1, describes all the Large Object Locator functions and the three interfaces in detail.

Limitations

Certain limitations are inherent in using large objects with a database, because the objects themselves, except for smart large objects, are not stored in the database and are not subject to direct control by the server. Two specific areas of concern are transaction rollback and concurrency control.

Transaction Rollback

Because large objects, other than smart large objects, are stored outside the database, any changes to them take place outside the server's control and cannot be rolled back if a transaction is aborted. For example, when you execute **lld_create()**, it calls an operating system routine to create the large object itself. If you roll back the transaction containing the call to **lld_create()**, the server has no way of deleting the object that you have just created.

Therefore, you are responsible for cleaning up any resources you have allocated if an error occurs. For example, if you create a large object and the transaction in which you create it is aborted, you should delete the object you have created. Likewise, if you have opened a large object and the transaction is aborted (or is committed), you should close the large object.

Concurrent Access

For the same reason, Large Object Locator provides no direct way of controlling concurrent access to large objects. If you open a large object for writing, it is possible to have two separate processes or users simultaneously alter the large object. You must provide a means, such as locking a row, to guarantee that multiple users cannot access a large object simultaneously for writing.

Installation and Registration

Large Object Locator is distributed with IBM Informix Dynamic Server. To use the Large Object Locator functions, you must use BladeManager to register the functions and data types with each database for which you want Large Object Locator functionality. See the *IBM Informix: DataBlade Module Installation and Registration Guide* for more information. This guide also contains some information about installing DataBlade modules.

Chapter 2. Data Types

In This Chapter 2-1
lld_locator 2-1
lld_lob 2-2

In This Chapter

This chapter describes the Large Object Locator data types, lld_locator and lld_lob.

lld_locator

The lld_locator data type identifies a large object. It specifies the kind of large object and provides a pointer to its location. lld_locator is a row type and is defined as follows:

```
create row type informix.lld_locator
   (
   lo_protocol                  char(18
   lo_pointer           informix.lld_lob0
   lo_location           informix.lvarchar
   )
```

lo_protocol identifies the kind of large object.

lo_pointer is a pointer to a smart large object, or is NULL if the large object is any kind of large object other than a smart large object.

lo_location is a pointer to the large object, if it is not a smart large object. Set to NULL if it is a smart large object.

In the *lo_protocol* field, specify the kind of large object to create. The kind of large object you specify determines the values of the other two fields:

- If you specify a smart large object:
  - use the *lo_pointer* field to point to it.
  - specify NULL for the *lo_location* field.
- If you specify any other kind of large object:
  - specify NULL for the *lo_pointer* field.
  - use the *lo_location* field to point to it.

The *lo_pointer* field uses the lld_lob data type, which is defined by Large Object Locator. This data type allows you to point to a smart large object and specify whether it is of type BLOB or type CLOB. See the description of lld_lob on page 2-2 for more information.

The *lo_location* field uses an lvarchar data type, which is a varying-length character type.

Table 2-1 lists the current protocols and summarizes the values for the other fields based on the protocol that you specify. Be sure to check the release notes shipped with this manual to see if Large Object Locator supports additional protocols not listed here.

© Copyright IBM Corp. 2005

Tip: Although the lld_locator type is not currently extensible, it might become so later. To avoid future name space collisions, the protocols established by Large Object Locator all have an IFX prefix.

Table 2-1. Fields of lld_locator Data Type

lo_protocol	lo_pointer	lo_location	Description
IFX_BLOB	Pointer to a smart large object	NULL	Smart large object
IFX_CLOB	Pointer to a smart large object	NULL	Smart large object
IFX_FILE	NULL	pathname	File accessible on server

Important: The lo_protocol field is case insensitive. It is shown in uppercase letters for display purposes only.

The lld_locator type is an instance of a row type. You can insert a row into the database using an SQL INSERT statement, or you can obtain a row by calling the DataBlade API **mi_row_create()** function. See the *IBM Informix: ESQL/C Programmer's Manual* for information on row types. See the *IBM Informix: DataBlade API Programmer's Guide* for information on the **mi_row_create()** function.

To reference an existing large object, you can insert an lld_locator row directly into a table in the database.

To create a large object, and a reference to it, you can call the **lld_create()** function and pass an lld_locator row.

You can pass an lld_locator type to these Large Object Locator functions, described in Chapter 3, "Functions," on page 3-1:

- **lld_copy(),** page 3-5
- **lld_create(),** page 3-7
- **lld_delete(),** page 3-9
- **lld_open(),** page 3-10
- **lld_from_client(),** page 3-20
- **lld_to_client(),** page 3-24

lld_lob

The lld_lob data type is a user-defined type. You can use it to specify the location of a smart large object and to specify whether the object contains binary or character data.

The lld_lob data type is defined for use with the API as follows:

```
typedef struct
   {
   MI_LO_HANDLE                lo;
   mi_integer                  type;
   } lld_lob_t;
```

It is defined for ESQL/C as follows:

```
typedef struct
   {
   ifx_lo_t                    lo;
   int                         type;
   } lld_lob_t;
```

lo is a pointer to the location of the smart large object.

type is the type of the object. For an object containing binary data, set *type* to LLD_BLOB; for an object containing character data, set *type* to LLD_CLOB.

The lld_lob type is equivalent to the CLOB or BLOB type in that it points to the location of a smart large object. In addition, it specifies whether the object contains binary or character data. You can pass the lld_lob type as the *lo_pointer* field of an lld_locator row. You should set the **lld_lob_t.***type* field to LLD_BLOB for binary data and to LLD_CLOB for character data.

See "Using the lld_lob Type" on page 4-1 for example code that uses that features the lld_lob type.

LOB Locator provides explicit casts from:

- a CLOB type to an lld_lob type.
- a BLOB type to an lld_lob type.
- an lld_lob type to the appropriate BLOB or CLOB type.

Tip: If you attempt to cast an lld_lob type containing binary data into a CLOB type or an lld_lob type containing character data into a BLOB type, Large Object Locator returns an error message.

You can pass an lld_lob type to these functions, described in Chapter 3, "Functions," on page 3-1:

- **LOCopy,** page 3-29
- **LOToFile,** page 3-30
- **LLD_LobType,** page 3-31

Note that **LOCopy** and **LOToFile** are overloaded versions of built-in server functions. The only difference is that you pass an lld_lob to the Large Object Locator versions of these functions and a BLOB or CLOB type to the built-in versions.

Chapter 3. Functions

In This Chapter . . . 3-1
Interfaces . . . 3-1
API Library . . . 3-2
ESQL/C Library . . . 3-2
SQL Interface . . . 3-2
Working with Large Objects . . . 3-2
lld_close() . . . 3-4
lld_copy() . . . 3-5
lld_create() . . . 3-7
lld_delete() . . . 3-9
lld_open() . . . 3-10
lld_read() . . . 3-12
lld_seek() . . . 3-13
lld_tell() . . . 3-15
lld_write() . . . 3-16
Client File Support . . . 3-17
lld_create_client() . . . 3-18
lld_delete_client() . . . 3-19
lld_from_client() . . . 3-20
lld_open_client() . . . 3-22
lld_to_client() . . . 3-24
Error Utility Functions . . . 3-25
lld_error_raise() . . . 3-26
lld_sqlstate() . . . 3-27
Smart Large Object Functions . . . 3-28
LOCopy . . . 3-29
LOToFile . . . 3-30
LLD_LobType . . . 3-31

In This Chapter

This chapter briefly describes the three interfaces to Large Object Locator and describes in detail all the Large Object Locator functions.

Interfaces

Large Object Locator functions are available through three interfaces:

- An API library
- An ESQL/C library
- An SQL interface

If the syntax for a function depends on the interface, each syntax appears under a separate subheading. Because there are few differences between parameters and usage in the different interfaces, there is a single parameter description and one "Usage," "Return," and "Related Topics" section for each function. Where there are differences between the interfaces, these differences are described.

The naming convention for the SQL interface is different from that for the ESQL/C and API interfaces. For example, the SQL client copy function is called **LLD_ToClient(),** whereas the API and ESQL/C client copy functions are called **lld_to_client()**. This manual uses the API and ESQL/C naming convention unless referring specifically to an SQL function.

© Copyright IBM Corp. 2005

API Library

All Large Object Locator functions except the smart large object functions are implemented as API functions defined in header and library files (**lldsapi.h** and **lldsapi.a**).

You can call the Large Object Locator API functions from your own user-defined routines. You execute Large Object Locator API functions just as you do functions provided by the IBM Informix DataBlade API. See the *IBM Informix: DataBlade API Programmer's Guide* for more information.

See "Using the API" on page 4-7 for an example of a user-defined routine that calls Large Object Locator API functions to copy part of a large object to another large object.

ESQL/C Library

All Large Object Locator functions except **lld_error_raise()** and the smart large object functions are implemented as ESQL/C functions, defined in header and library files (**lldesql.h** and **lldesql.so**).

Wherever possible, the ESQL/C versions of the Large Object Locator functions avoid server interaction by directly accessing the underlying large object.

See the *IBM Informix: ESQL/C Programmer's Manual* for more information on using the ESQL/C interface to execute Large Object Locator functions.

SQL Interface

The following Large Object Locator functions are implemented as user-defined routines that you can execute within SQL statements:

- **LLD_LobType()**
- **LLD_Create()**
- **LLD_Delete()**
- **LLD_Copy()**
- **LLD_FromClient()**
- **LLD_ToClient()**
- **LOCopy()**
- **LOToFile()**

See the following three-volume set for further information about the IBM Informix SQL interface:

- *IBM Informix: Guide to SQL Reference*
- *IBM Informix: Guide to SQL Syntax*
- *IBM Informix: Guide to SQL Tutorial*

Working with Large Objects

This section describes functions that allow you to:

- create large objects.
- open, close, and delete large objects.
- return and change the current position within a large object.
- read from and write to large objects.

- copy a large object.

Generally, you use the functions described in this section in the following order.

1. You use **lld_create()** to create a large object. It returns a pointer to an lld_locator row that points to the large object.

   If the large object already exists, you can insert an lld_locator row into a table in the database to point to the object without calling **lld_create()**.
2. You can pass the lld_locator type to the **lld_open()** function to open the large object you created. This function returns an **LLD_IO** structure that you can pass to various Large Object Locator functions to manipulate data in the open object (see Step 3).

   You can also pass the lld_locator type to the **lld_copy(), lld_from_client(),** or **lld_to_client()** functions to copy the large object.
3. After you open a large object, you can pass the **LLD_IO** structure to:
   - **lld_tell()** to return the current position within the large object.
   - **lld_seek()** to change the current position within the object.
   - **lld_read()** to read from large object.
   - **lld_write()** to write to the large object.
   - **lld_close()** to close an object. You should close a large object if the transaction in which you open it is aborted or committed.

Tip: To delete a large object, you can pass the lld_locator row to **lld_delete()** any time after you create it. For example, if the transaction in which you created the object is aborted and the object is not a smart large object, you should delete the object because the server's rollback on the transaction cannot delete an object outside the database.

The functions within this section are presented in alphabetical order, not in the order in which you might use them.

lld_close()

This function closes the specified large object.

Syntax

API:

```
mi_integer lld_close (conn, io, error)
   MI_CONNECTION*              conn;
   LLD_IO*                     io;
   mi_integer*                 error;
```

ESQL/C:

```
int lld_close (LLD_IO* io, int* error);
```

conn is the connection descriptor established by a previous call to the **mi_open()** or **mi_server_connect()** functions. This parameter is for the API interface only. In the ESQL/C version of this function, you must already be connected to a server.

io is a pointer to an **LLD_IO** structure created with a previous call to the **lld_open()** function.

error is an output parameter in which the function returns an error code.

Usage

The **lld_close()** function closes the open large object and frees the memory allocated for the **LLD_IO** structure, which you cannot use again after this call.

Return Codes

For an API function, returns `MI_OK` if the function succeeds and `MI_ERROR` if it fails.

For an ESQL/C function, returns `0` if the function succeeds and `-1` if it fails.

Context

lld_open(), page 3-10

lld_copy()

This function copies the specified large object.

Syntax

API:

```
MI_ROW* lld_copy(conn, src, dest, error);
   MI_CONNECTION*              conn,
   MI_ROW*                     src,
   MI_ROW*                     dest,
   mi_integer*                 error
```

ESQL/C:

```
ifx_collection_t* lld_copy (src, dest, error);
   EXEC SQL BEGIN DECLARE SECTION;
      PARAMETER ROW src;
      PARAMETER ROW dest;
   EXEC SQL END DECLARE SECTION;
   int* error;
```

SQL:

```
CREATE FUNCTION LLD_Copy (src LLD_Locator, dest LLD_Locator)
   RETURNS LLD_Locator;
```

conn is the connection descriptor established by a previous call to the **mi_open()** or **mi_server_connect()** function. This parameter is for the API interface only. In the ESQL/C and SQL versions of this function, you must already be connected to a server.

src is a pointer to the lld_locator row, identifying the source object.

dest is a pointer to an lld_locator row, identifying the destination object. If the destination object itself does not exist, it is created.

error is an output parameter in which the function returns an error code. The SQL version of this function does not have an *error* parameter.

Usage

This function copies an existing large object.

If the destination object exists, pass a pointer to its lld_locator row as the *dest* parameter.

If the destination object does not exist, pass an lld_locator row with the following values as the *dest* parameter to **lld_copy()**:

In the *lo_protocol* field, specify the type of large object to create.

If you are copying to any type of large object other than a smart large object:

- specify NULL for the *lo_pointer* field.
- point to the location of the new object in the *lo_location* field.

The **lld_copy()** function creates the type of large object that you specify, copies the source object to it, and returns the row you passed, unaltered.

If you are copying to a smart large object, specify NULL for the *lo_pointer* and *lo_location* fields of the lld_locator row that you pass as the *dest* parameter. The **lld_copy()** function returns an lld_locator row with a pointer to the new smart large object in the *lo_pointer* field.

The server deletes a new smart large object at the end of a transaction if there are no disk references to it and if it is closed. Therefore, after copying to a newly created smart large object, either open it or insert it into a table.

If **lld_copy()** creates a new smart large object, it uses system defaults for required storage parameters such as *sbspace*. If you want to override these parameters, you can use the server large object interface to create the smart large object and specify the parameters you want in an **MI_LO_SPEC** structure. You can then call **lld_copy()** and set the *lo_pointer* field of the lld_locator row to point to the new smart large object.

Likewise, if protocols are added to Large Object Locator for new types of large objects, these objects might require creation attributes or parameters for which Large Object Locator supplies predefined default values. As with smart large objects, you can create the object with **lld_copy()** and accept the default values, or you can use the creation routines specific to the new protocol and supply your own attributes and parameters. After you create the object, you can call **lld_copy()** and pass it an lld_locator row that points to the new object.

Return Codes

On success, this function returns a pointer to an lld_locator row, specifying the location of the copy of the large object. If the destination object already exists, **lld_copy()** returns a pointer to the unaltered lld_locator row you passed in the *dest* parameter. If the destination object does not already exist, **lld_copy()** returns a pointer to an lld_locator row, pointing to the new object it creates.

On failure, this function returns NULL.

Context

lld_from_client(), page 3-20

lld_to_client(), page 3-24

lld_create()

This function creates a new large object with the protocol and location you specify.

Syntax

API:

```
MI_ROW* lld_create(conn, lob, error)
   MI_CONNECTION*              conn
   MI_ROW*                     lob;
   mi_integer*                 error;
```

ESQL/C:

```
ifx_collection_t* lld_create (lob, error);
   EXEC SQL BEGIN DECLARE SECTION;
      PARAMETER ROW lob;
   EXEC SQL END DECLARE SECTION;
   int* error;
```

SQL:

```
CREATE FUNCTION LLD_Create (lob LLD_Locator)
   RETURNS LLD_Locator;
```

conn is the connection descriptor established by a previous call to the **mi_open()** or **mi_server_connect()** functions. This parameter is for the API interface only. In the ESQL/C and SQL versions of this function, you must already be connected to a server.

lob is a pointer to an lld_locator row, identifying the object to create.

error is an output parameter in which the function returns an error code. The SQL version of this function does not have an *error* parameter.

Usage

You pass an lld_locator row, with the following values, as the *lob* parameter to **lld_create()**:

In the *lo_protocol* field, specify the type of large object to create.

For any type of large object other than a smart large object:

- specify `NULL` for the *lo_pointer* field.
- point to the location of the new object in the *lo_location* field.

The **lld_create()** function returns the row you passed, unaltered.

If you are creating a smart large object, specify `NULL` for the *lo_pointer* and *lo_location* fields of the lld_locator row. The **lld_create()** function returns an lld_locator row with a pointer to the new smart large object in the *lo_pointer* field.

The server deletes a new smart large object at the end of a transaction if there are no disk references to it and if it is closed. Therefore, after creating a smart large object, either open it or insert it into a table.

Large Object Locator does not directly support transaction rollback, except for smart large objects. Therefore, if the transaction in which you call **lld_create()** is aborted, you should call **lld_delete()** to delete the object and reclaim any allocated resources.

See "Transaction Rollback" on page 1-3 for more information.

When you create a smart large object, **lld_create()** uses system defaults for required storage parameters such as *sbspace*. If you want to override these parameters, you can use the server large object interface to create the smart large object and specify the parameters you want in an **MI_LO_SPEC** structure. You can then call **lld_create()** and set the *lo_pointer* field of the lld_locator row to point to the new smart large object.

Likewise, if protocols are added to Large Object Locator for new types of large objects, these objects might require creation attributes or parameters for which Large Object Locator supplies predefined default values. As with smart large objects, you can create the object with **lld_create()** and accept the default values, or you can use the creation routines specific to the new protocol and supply your own attributes and parameters. After you create the object, you can call **lld_create()** and pass it an lld_locator row that points to the new object.

Return Codes

On success, this function returns a pointer to an lld_locator row specifying the location of the new large object. For a smart large object, **lld_create()** returns a pointer to the location of the new object in the *lo_pointer* field of the lld_locator row. For all other objects, it returns a pointer to the unaltered lld_locator row you passed in the *lob* parameter.

The **lld_open** function can use the lld_locator row that **lld_create()** returns.

On failure, this function returns `NULL`.

Context

lld_delete(), page 3-9

lld_open(), page 3-10

lld_delete()

This function deletes the specified large object.

Syntax

API:

```
mi_integer lld_delete(conn, lob, error)
   MI_CONNECTION*              conn;
   LLD_Locator                 lob;
   mi_integer*                 error;
```

ESQL/C:

```
int lld_delete (lob, error);
EXEC SQL BEGIN DECLARE SECTION;
   PARAMETER ROW lob;
EXEC SQL END DECLARE SECTION;
   int* error;
```

SQL:

```
CREATE FUNCTION LLD_Delete (lob LLD_Locator)
   RETURNS BOOLEAN;
```

conn is the connection descriptor established by a previous call to the **mi_open()** or **mi_server_connect()** functions. This parameter is for the API interface only. In the ESQL/C and SQL versions of this function, you must already be connected to a server.

lob is a pointer to an lld_locator row, identifying the object to delete.

error is an output parameter in which the function returns an error code. The SQL version of this function does not have an *error* parameter.

Usage

For large objects other than smart large objects, this function deletes the large object itself, not just the lld_locator row referencing it. For smart large objects, this function does nothing.

To delete a smart large object, delete all references to it, including the lld_locator row referencing it.

Return Codes

For an API function, returns `MI_OK` if the function succeeds and `MI_ERROR` if it fails.

For an ESQL/C function, returns `0` if the function succeeds and `-1` if the function fails.

lld_open()

This function opens the specified large object.

Syntax

API:

```
LLD_IO* lld_open(conn, lob, flags, error)
   MI_CONNECTION*              conn;
   MI_ROW*                     lob;
   mi_integer                  flags,
   mi_integer*                 error);
```

ESQL/C:

```
LLD_IO* lld_open(lob, flags, error);
   EXEC SQL BEGIN DECLARE SECTION;
      PARAMETER ROW lob;
   EXEC SQL END DECLARE SECTION;
      int flags;int* error;
```

conn is the connection descriptor established by a previous call to the **mi_open()** or **mi_server_connect()** functions. This parameter is for the API interface only. In the ESQL/C version of this function, you must already be connected to a server.

lob is a pointer to an lld_locator row, identifying the object to open.

flags is a set of flags that you can set to specify attributes of the large object after it is opened. The flags are as follows:

LLD_RDONLY
opens the large object for reading only. You cannot use the **lld_write** function to write to the specified large object when this flag is set.

LLD_WRONLY
opens the large object for writing only. You cannot use the **lld_read()** function to read from the specified large object when this flag is set.

LLD_RDWR
opens the large object for both reading and writing.

LLD_TRUNC
clears the contents of the large object after opening.

LLD_APPEND
seeks to the end of the large object for writing. When the object is opened, the file pointer is positioned at the beginning of the object. If you have opened the object for reading or reading and writing, you can seek anywhere in the file and read. However, any time you call **lld_write()** to write to the object, the pointer moves to the end of the object to guarantee that you do not overwrite any data.

LLD_SEQ
opens the large object for sequential access only. You cannot use the **lld_seek()** function with the specified large object when this flag is set.

error is an output parameter in which the function returns an error code.

Usage

In the *lob* parameter, you pass an lld_locator row to identify the large object to open. In the *lo_protocol* field of this row, you specify the type of the large object to open. The **lld_open()** function calls an appropriate open routine based on the type you specify. For example, for a file, **lld_open()** uses an operating system file function to open the file, whereas, for a smart large object, it calls the server's **mi_lo_open()** routine.

Large Object Locator does not directly support two fundamental database features, transaction rollback and concurrency control. Therefore, if the transaction in which you call **lld_open()** is aborted, you should call **lld_close()** to close the object and reclaim any allocated resources.

Your application should also provide some means, such as locking a row, to guarantee that multiple users cannot write to a large object simultaneously.

See "Limitations" on page 1-3 for more information about transaction rollback and concurrency control.

Return Codes

On success, this function returns a pointer to an **LLD_IO** structure it allocates. The **LLD_IO** structure is private, and you should not directly access it or modify its contents. Instead, you can pass the **LLD_IO** structure's pointer to Large Object Locator routines such as **lld_write()**, **lld_read()**, and so on, that access open large objects.

A large object remains open until you explicitly close it with the **lld_close()** function. Therefore, if you encounter error conditions after opening a large object, you are responsible for reclaiming resources by closing it.

On failure, this function returns `NULL`.

Context

lld_close(), page 3-4

lld_create(), page 3-7

lld_read(), page 3-12

lld_seek(), page 3-13

lld_tell(), page 3-15

lld_write(), page 3-16

lld_read()

This function reads from a large object, starting at the current position.

Syntax

API:

```
mi_integer lld_read (io, buffer, bytes, error)

LLD_IO*                         io,
void*                           buffer,
mi_integer                      bytes,
mi_integer*                     error);
```

ESQL/C:

```
int lld_read (LLD_IO* io,
            void* buffer, int bytes,
            int* error);
```

io is a pointer to an **LLD_IO** structure created with a previous call to the **lld_open()** function.

buffer is a pointer to a buffer into which to read the data. The buffer must be at least as large as the number of bytes specified in the *bytes* parameter.

bytes is the number of bytes to read.

error is an output parameter in which the function returns an error code.

Usage

Before calling this function, you must open the large object with a call to **lld_open()** and set the LLD_RDONLY or LLD_RDWR flag. The **lld_read()** function begins reading from the current position. By default, when you open a large object, the current position is the beginning of the object. You can call **lld_seek()** to change the current position.

Return Codes

On success, the **lld_read()** function returns the number of bytes that it has read from the large object.

On failure, for an API function, it returns `MI_ERROR`; for an ESQL/C function, it returns `-1`.

Context

lld_open(), page 3-10

lld_seek(), page 3-13

lld_tell(), page 3-15

lld_seek()

This function sets the position for the next read or write operation to or from a large object that is open for reading or writing.

Syntax

API:

```
mi_integer lld_seek(conn, io, offset, whence, new_offset, error)
   MI_CONNECTION*              conn
   LLD_IO*                     io;
   mi_int8*                    offset;
   mi_integer                  whence;
   mi_int8*                    new_offset;
   mi_integer*                 error;
```

ESQL/C:

```
int lld_seek(io,offset, whence, new_offset, error)
   LLD_IO* io;
EXEC SQL BEGIN DECLARE SECTION;
   PARAMETER int8* offset;
EXEC SQL END DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
   PARAMETER int8* new_offset;
EXEC SQL END DECLARE SECTION;
   int whence;
   int* error;
```

conn is the connection descriptor established by a previous call to the **mi_open()** or **mi_server_connect()** functions. This parameter is for the API interface only. The ESQL/C version of this function is based on the assumption that you are already connected to a server.

io is a pointer to an **LLD_IO** structure created with a previous call to the **lld_open()** function.

offset is a pointer to the offset. It describes where to seek in the object. Its value depends on the value of the *whence* parameter.

- If *whence* is LLD_SEEK_SET, the offset is measured relative to the beginning of the object.
- If *whence* is LLD_SEEK_CUR, the offset is relative to the current position in the object.
- If *whence* is LLD_SEEK_END, the offset is relative to the end of the file.

whence determines how the offset is interpreted.

new_offset is a pointer to an **int8** that you allocate. The function returns the new offset in this **int8**.

error is an output parameter in which the function returns an error code.

Usage

Before calling this function, you must open the large object with a call to **lld_open()**.

Although this function takes an 8-byte offset, this offset is converted to the appropriate size for the underlying large object storage system. For example, if the large object is stored in a 32-bit file system, the 8-byte offset is converted to a 4-byte offset, and any attempt to seek past 4 GB generates an error.

lld_sqlstate()

This function translates integer error codes into their corresponding SQL states.

Syntax

API:

```
mi_string* lld_sqlstate (error);
   mi_integer                   error
```

ESQL/C:

```
int* lld_sqlstate (int error);
```

error is an error code.

Return Codes

On success, this function returns the SQL state value corresponding to the error code. On failure, returns `NULL`.

Important: This function returns a pointer to a constant, not to an allocated memory location.

Smart Large Object Functions

The functions described in this section allow you to copy a smart large object to a file and to copy a smart large object to another smart large object. There is also a function that tells you whether the data in an lld_lob column is binary or character data.

LOCopy

This function creates a copy of a smart large object.

Syntax

SQL:

```
CREATE FUNCTION LOCopy (lob LLD_Lob)
   RETURNS LLD_Lob ;

CREATE FUNCTION LOCopy (lob, LLD_Lob, table_name, CHAR(18),
column_name, CHAR(18))
   RETURNS LLD_Lob;
;
```

lob is a pointer to the smart large object to copy.

table_name is a table name. This parameter is optional.

column_name is a column name. This parameter is optional.

Usage

This function is an overloaded version of the **LOCopy** built-in server function. This function is identical to the built-in version of the function, except the first parameter is an lld_lob type rather than a BLOB or CLOB type.

The *table_name* and *column_name* parameters are optional. If you specify a *table_name* and *column_name*, **LOCopy** uses the storage characteristics from the specified *column_name* for the new smart large object that it creates.

If you omit *table_name* and *column_name*, **LOCopy** creates a smart large object with system-specified storage defaults.

See the description of the **LOCopy** function in the *IBM Informix: Guide to SQL Syntax* for complete information about this function.

Return Codes

This function returns a pointer to the new lld_lob value.

Context

LOCopy in the *IBM Informix: Guide to SQL Syntax*

LOToFile

Copies a smart large object to a file.

Syntax

SQL:

```
CREATE FUNCTION LOToFile(lob LLD_Lob, pathname LVARCHAR,
file_dest CHAR(6)
   RETURNS LVARCHAR;
```

lob is a pointer to the smart large object.

pathname is a directory path and name of the file to create.

file_dest is the computer on which the file resides. Specify either `server` or `client`.

Usage

This function is an overloaded version of the **LOToFile** built-in server function. This function is identical to the built-in version of the function, except the first parameter is an lld_lob type rather than a BLOB or CLOB type.

See the description of the **LOToFile** function in the *IBM Informix: Guide to SQL Syntax* for complete information about this function.

Return Codes

This function returns the value of the new filename.

Context

LOToFile in the *IBM Informix: Guide to SQL Syntax*

LLD_LobType

Returns the type of data in an lld_lob column.

Syntax

SQL:

```
CREATE FUNCTION LLD_LobType(lob LLD_Lob)
   RETURNS CHAR(4);
```

lob is a pointer to the smart large object

Usage

An lld_lob column can contain either binary or character data. You pass an lld_lob type to the **LLD_LobType** function to determine the type of data that the column contains.

Return Codes

This function returns `blob` if the specified lld_lob contains binary data and `clob` if it contains character data.

Chapter 4. Example Code

In This Chapter 4-1
Using the SQL Interface 4-1
Using the lld_lob Type 4-1
Using Implicit lld_lob Casts 4-1
Using Explicit lld_lob Casts 4-2
Using the LLD_LobType Function 4-3
Using the lld_locator Type 4-4
Inserting an lld_locator Row into a Table 4-4
Creating a Smart Large Object 4-4
Copying a Client File to a Large Object 4-5
Copying a Large Object to a Large Object 4-5
Copying Large Object Data to a Client File 4-6
Creating and Deleting a Server File 4-6
Using the API 4-7
Creating the lld_copy_subset Function 4-7
Using the lld_copy_subset Routine 4-10

In This Chapter

This chapter provides example code that shows how to use some of the Large Object Locator functions together. It shows how to use all three of the Large Object Locator interfaces: SQL, server, and ESQL/C.

Using the SQL Interface

The examples in this section show how to use the SQL interface to Large Object Locator.

Using the lld_lob Type

The lld_lob is a user-defined type that you can use to specify the location of a smart large object and to specify whether the object contains binary or character data. The following subsections show how to use the lld_lob data type.

Using Implicit lld_lob Casts

This section shows how to insert binary and character data into an lld_lob type column of a table. The example in Figure 4-1 makes use of implicit casts from BLOB and CLOB types to the lld_lob type.

© Copyright IBM Corp. 2005

```
create table slobs (key int primary key, slo lld_lob);

--Insert binary and text large objects into an lld_lob field
--Implicitly cast from blob/clob to lld_lob
insert into slobs values (1, filetoblob ('logo.gif', 'client'));

insert into slobs values (2, filetoclob ('quote1.txt', 'client'));

select * from slobs;


key  1
slo  blob:00608460a6b7c8d900000002000000030000000200000018000000000001000000608
     460736c6f000010029a2a6c92070000000000006c000af0cdd900000080006082500af0c9d
     e

key  2
slo  clob:00608460a6b7c8d900000002000000030000000300000019000000000001000000608
     460736c6f000010029a2a6c930d0000000000006c000af0cdd900000016000000010af0c9d
     e
```

Figure 4-1. Implicit lld_lob Casts

The **slobs** table, created in this example, contains the **slo** column, which is of type lld_lob. The first INSERT statement uses the **filetoblob** function to copy a binary large object to a smart large object. There exists an implicit cast from a BLOB type to an lld_lob type, so the INSERT statement can insert the BLOB type large object into an lld_lob type column.

Likewise, there is an implicit cast from a CLOB type to an lld_lob type, so the second INSERT statement can insert a CLOB type large object into the **slo** column of the **slobs** table.

The SELECT statement returns the lld_lob types that identify the two smart large objects stored in the **slobs** table.

The **slo** column for key 1 contains an instance of an lld_lob type that identifies the data as BLOB data and contains a hexadecimal number that points to the location of the data.

The **slo** column for key 2 identifies the data as CLOB data and contains a hexadecimal number that points to the location of the data.

Using Explicit lld_lob Casts

The example in Figure 4-2 on page 4-3 shows how to select large objects of type BLOB and CLOB from a table and how to copy them to a file.

This example uses the **slobs** table created in Figure 4-1 on page 4-2.

```
--Explicitly cast from lld_lob to blob/clob
select slo::blob from slobs where key = 1;


(expression)  <SBlob Data>


select slo::clob from slobs where key = 2;


(expression)
Ask not what your country can do for you,
but what you can do for your country.
```

Figure 4-2. Explicit lld_lob Casts

The first SELECT statement retrieves the data in the **slo** column associated with key 1 and casts it as BLOB type data. The second SELECT statement retrieves the data in the **slo** column associated with key 2 and casts it as CLOB type data.

Using the LLD_LobType Function

The example in this section, Figure 4-3, shows how to use the **LLD_LobType** function to obtain the type of data—BLOB or CLOB—that an lld_lob column contains.

The **slobs** table in this example is the same one created in Figure 4-1 on page 4-2. That example created the table and inserted a BLOB type large object for key 1 and a CLOB type large object for key 2.

```
-- LLD_LobType UDR
select key, lld_lobtype(slo) from slobs;

        key (expression)

          1 blob
          2 clob


select slo::clob from slobs where lld_lobtype(slo) = 'clob';


(expression)
Ask not what your country can do for you,
but what you can do for your country.
```

Figure 4-3. Using LLD_LobType Function

The first SELECT statement returns:

```
1 blob
2 clob
```

indicating that the data associated with key 1 is of type BLOB and the data associated with key 2 is of type CLOB.

The second SELECT statement uses **LLD_LobType** to retrieve the columns containing CLOB type data. The second SELECT statement casts the **slo** column (which is of type lld_lob) to retrieve CLOB type data.

Using the lld_locator Type

The lld_locator type defines a large object. It identifies the type of large object and points to its location. It contains three fields:

lo_protocol identifies the kind of large object.

lo_pointer is a pointer to a smart large object or is NULL if the large object is any kind of large object other than a smart large object.

lo_location is a pointer to the large object, if it is not a smart large object. Set to NULL if it is a smart large object.

The examples in this section show how to:

- insert an lld_locator row for an existing server file into a table.
- create a smart large object.
- copy a client file to a large object.
- copy a large object to another large object.
- copy a large object to a client file.
- create and delete a server file.

Inserting an lld_locator Row into a Table

The example in Figure 4-4 creates a table with an lld_locator row and shows how to insert a large object into the row.

```
--Create lobs table
create table lobs (key int primary key, lo lld_locator);

-- Create an lld_locator for an existing server file
insert into lobs
    values (1, "row('ifx_file',null,'/tmp/quote1.txt')");
```

Figure 4-4. Inserting an lld_locator Row Into a Table

The INSERT statement inserts an instance of an lld_locator row into the **lobs** table. The protocol in the first field, IFX_FILE, identifies the large object as a server file. The second field, *lo_pointer*, is used to point to a smart large object. Because the object is a server file, this field is NULL. The third field identifies the server file as **quote1.txt**.

Creating a Smart Large Object

The example in Figure 4-5 creates a smart large object containing CLOB type data. The **lld_create** function in Figure 4-5 creates a smart large object. The first parameter to **lld_create** uses the IFX_CLOB protocol to specify CLOB as the type of object to create. The other two arguments are NULL.

The **lld_create** function creates the CLOB type large object and returns an lld_locator row that identifies it.

The insert statement inserts in the **lobs** table the lld_locator row returned by **lld_create**.

```
--Create a new clob using lld_create
insert into lobs
    values (2, lld_create ("row('ifx_clob',null,null)"::lld_locator));
```

Figure 4-5. Using lld_create

Copying a Client File to a Large Object

The example in Figure 4-6 uses the **lobs** table created in Figure 4-5.

In Figure 4-6, the **lld_fromclient** function in the first SELECT statement, copies the client file, **quote2.txt**, to an lld_locator row in the **lobs** table.

```
-- Copy a client file to an lld_locator
select lld_fromclient ('quote2.txt', lo) from lobs where key = 2;


(expression)  ROW('IFX_CLOB      ','clob:ffffffffa6b7c8d9000000020000000300
          0000090000001a0000000000010000000000000ad3c3dc000000000b06eec8000
          00000005c4e6000607fdc00000000000000000000000000',NULL)


select lo.lo_pointer::clob from lobs where key = 2;


(expression)
To be or not to be,
that is the question.
```

Figure 4-6. Copying a Client File to a Large Object

The **lld_fromclient** function returns a pointer to the lld_locator row that identifies the data copied from the large object. The first SELECT statement returns this lld_locator row.

The next SELECT statement selects the *lo_pointer* field of the lld_locator row, `lo.lo_pointer`, and casts it to CLOB type data. The result is the data itself.

Copying a Large Object to a Large Object

The example in Figure 4-7 uses the **lobs** table created in Figure 4-4 on page 4-4.

The **lld_copy** function in Figure 4-7 copies large object data from one lld_locator type row to another.

```
-- Copy an lld_locator to an lld_locator
select lld_copy (S.lo, D.lo) from lobs S, lobs D where S.key = 1 and D.key = 2;


(expression)  ROW('IFX_CLOB      ','clob:ffffffffa6b7c8d9000000020000000300
          0000090000001a000000000001000000000000ad3c3dc000000000b06eec8000
          00000005c4e6000607fdc00000000000000000000000000',NULL)


select lo.lo_pointer::clob from lobs where key = 2;


(expression)
Ask not what your country can do for you,
but what you can do for your country.
```

Figure 4-7. Copying a Large Object to a Large Object

The second SELECT statement casts `lo.lo_pointer` to a CLOB type to display the data in the column.

Copying Large Object Data to a Client File

The example in Figure 4-8 uses the **lobs** table created in Figure 4-4 on page 4-4. The **lld_toclient** function in "Copying Large Object Data to a Client File" on page 4-6 copies large object data to the **output.txt** client file. This function returns `t` when the function succeeds. The SELECT statement returns `t`, or `true`, indicating that the function returned successfully.

```
-- Copy an lld_locator to a client file
select lld_toclient (lo, 'output.txt') from lobs where key = 2;

(expression)

          t
```

Figure 4-8. Copying a Large Object to a Client File

Creating and Deleting a Server File

The example in Figure 4-9 on page 4-7 shows how to create a server file and then delete it.

The **lld_copy** function copies a large object to another large object. The lld_locator rows for the source and destination objects use the IFX_FILE protocol to specify a server file as the type of large object. The **lld_copy** function returns an lld_locator row that identifies the copy of the large object.

The INSERT statement inserts this row into the **lobs** table using 3 as the key.

```
-- Create and delete a new server file
insert into lobs
    values (3, lld_copy (
        "row('ifx_file',null,'/tmp/quote2.txt')"::lld_locator,
        "row('ifx_file',null,'/tmp/tmp3')"::lld_locator));

select lo from lobs where key = 3;


lo  ROW('IFX_FILE          ',NULL,'/tmp/tmp3')


select lld_delete (lo) from lobs where key = 3;

(expression)

           t


delete from lobs where key = 3;
```

Figure 4-9. Creating and Deleting a Server File

The first SELECT statement returns the lld_locator row identifying the large object.

The **lld_delete** function deletes the large object itself. The DELETE statement deletes the lld_locator row that referenced the large object.

Using the API

This section contains one example that shows how to use the Large Object Locator functions to create a user-defined routine. This routine copies part of a large object to another large object.

Creating the lld_copy_subset Function

Figure 4-10 on page 4-8 shows the code for the **lld_copy_subset** user-defined routine. This routine copies a portion of a large object and appends it to another large object.

```
/* LLD SAPI interface example */

#include <mi.h>
#include <lldsapi.h>

/* append a (small) subset of a large object to another large object */

MI_ROW*
lld_copy_subset (MI_ROW* src,           /* source LLD_Locator */
                 MI_ROW* dest,          /* destination LLD_Locator */
                 mi_int8* offset,       /* offset to begin copy at */
                 mi_integer nbytes,     /* number of bytes to copy */
                 MI_FPARAM* fp)
{
    MI_ROW*        new_dest;       /* return value */
    MI_CONNECTION* conn;           /* database server connection */
    mi_string*     buffer;         /* I/O buffer */
    LLD_IO*        io;             /* open large object descriptor */
    mi_int8        new_offset;     /* offset after seek */
    mi_integer     bytes_read;     /* actual number of bytes copied */
    mi_integer     error;          /* error argument */
    mi_integer     _error;         /* extra error argument */
    mi_boolean     created_dest;   /* did we create the dest large object? */

    /* initialize variables */
    new_dest = NULL;
    conn = NULL;
    buffer = NULL;
    io = NULL;
    error = LLD_E_OK;
    created_dest = MI_FALSE;

    /* open a connection to the database server */
    conn = mi_open (NULL, NULL, NULL);
    if (conn == NULL)
        goto bad;

    /* allocate memory for I/O */
    buffer = mi_alloc (nbytes);
    if (buffer == NULL)
        goto bad;

    /* read from the source large object */
    io = lld_open (conn, src, LLD_RDONLY, &error);
    if (error != LLD_E_OK)
        goto bad;

    lld_seek (conn, io, offset, LLD_SEEK_SET, &new_offset, &error);
    if (error != LLD_E_OK)
        goto bad;
```

Figure 4-10. The lld_copy_subset Function (Part 1 of 2)

```
    bytes_read = lld_read (conn, io, buffer, nbytes, &error);
    if (error != LLD_E_OK)
        goto bad;

    lld_close (conn, io, &error);
    if (error != LLD_E_OK)
        goto bad;

    /* write to the destination large object */
    new_dest = lld_create (conn, dest, &error);
    if (error == LLD_E_OK)
        created_dest = MI_TRUE;
    else if (error != LLD_E_EXISTS)
        goto bad;

    io = lld_open (conn, new_dest, LLD_WRONLY | LLD_APPEND | LLD_SEQ, &error);
    if (error != LLD_E_OK)
        goto bad;

    lld_write (conn, io, buffer, bytes_read, &error);
    if (error != LLD_E_OK)
        goto bad;

    lld_close (conn, io, &error);
    if (error != LLD_E_OK)
        goto bad;

    /* free memory */
    mi_free (buffer);

    /* close the database server connection */
    mi_close (conn);

    return new_dest;

    /* error clean up */
bad:
    if (io != NULL)
        lld_close (conn, io, &_error);
    if (created_dest)
        lld_delete (conn, new_dest, &_error);
    if (buffer != NULL)
        mi_free (buffer);
    if (conn != NULL)
        mi_close (conn);
    lld_error_raise (conn, error);
    mi_fp_setreturnisnull (fp, 0, MI_TRUE);
    return NULL;
}
```

Figure 4-10. The lld_copy_subset Function (Part 2 of 2)

The **lld_copy_subset** function defines four parameters:

- A source large object (lld_locator type)
- A destination large object (lld_locator type)
- The byte offset to begin copying
- The number of bytes to copy

It returns an lld_locator, identifying the object being appended.

The **mi_open** function opens a connection to the database. A buffer is allocated for I/O.

The following Large Object Locator functions are called for the source object:

- **lld_open**, to open the source object
- **lld_seek**, to seek to the specified byte offset in the object
- **lld_read**, to read the specified number of bytes from the object
- **lld_close**, to close the object

The following Large Object Locator functions are called for the destination object:

- **lld_open**, to open the destination object
- **lld_write**, to write the bytes read from the source into the destination object
- **lld_close**, to close the destination object

The **mi_close** function closes the database connection.

This function also contains error-handling code. If the database connection cannot be made, if memory cannot be allocated, or if any of the Large Object Locator functions returns an error, the error code is invoked.

The error code handling code (`bad`) does one or more of the following actions, if necessary:

- Closes the source file
- Deletes the destination file
- Frees the buffer
- Closes the database connection
- Raises an error

You should establish a callback for exceptions (this example code, in the interest of simplicity and clarity, does not do so). See the *IBM Informix: DataBlade API Programmer's Guide* for more information.

Using the lld_copy_subset Routine

The example in this section, Figure 4-11, shows how to use the **lld_copy_subset** user-defined routine defined in the previous section.

```
-- Using the lld_copy_subset function

create function lld_copy_subset (lld_locator, lld_locator, int8, int)
    returns lld_locator
    external name '/tmp/sapidemo.so'
    language c;

insert into lobs
    values (5, lld_copy_subset (
        "row('ifx_file',null,'/tmp/quote3.txt')"::lld_locator,
    "row('ifx_clob',null,null)"::lld_locator, 20, 70));

select lo from lobs where key = 5;
select lo.lo_pointer::clob from lobs where key = 5;
```

Figure 4-11. Using the lld_copy_subset Routine

The **lld_copy_subset** function copies 70 bytes, beginning at offset 20 from the **quote3.txt** file, and appends them to a CLOB object. The INSERT statement inserts this data into the **lobs** table.

The first SELECT statement returns the lld_locator that identifies the newly copied CLOB data. The second SELECT statement returns the data itself.

Chapter 5. Error Handling

In This Chapter 5-1
Handling Large Object Locator Errors 5-1
Handling Exceptions 5-1
Error Codes 5-2

In This Chapter

This chapter describes how to handle errors when calling Large Object Locator functions. It also lists and describes specific Large Object Locator errors.

There are two methods by which Large Object Locator returns errors to you:

- Through the error argument of a Large Object Locator function
- Through an exception

Both the API and ESQL/C versions of Large Object Locator functions use the error argument. Exceptions are returned only to the API functions.

Handling Large Object Locator Errors

All Large Object Locator functions use the return value to indicate failure. Functions that return a pointer return `NULL` in the event of failure. Functions that return an integer return `-1`.

Large Object Locator functions also provide an error code argument that you can test for specific errors. You can pass this error code to **lld_error_raise()**—which calls **mi_db_error_raise** if necessary to generate an MI_EXCEPTION—and propagate the error up the calling chain.

For ESQL/C functions, the LLD_E_SQL error indicates that an SQL error occurred. You can check the SQLSTATE variable to determine the nature of the error.

When an error occurs, Large Object Locator functions attempt to reclaim any outstanding resources. You should close any open large objects and delete any objects you have created that have not been inserted into a table.

A user-defined routine that directly or indirectly calls a Large Object Locator function (API version) can register a callback function. If this function catches and handles an exception and returns control to the Large Object Locator function, Large Object Locator returns the LLD_E_EXCEPTION error. You can handle this error as you would any other: close open objects and delete objects not inserted in a table.

Handling Exceptions

You should register a callback function to catch exceptions generated by underlying DataBlade API functions called by Large Object Locator functions. For example, if you call **lld_read()** to open a smart large object, Large Object Locator calls the DataBlade API **mi_lo_read()** function. If this function returns an error and generates an exception, you must catch the exception and close the object you have open for reading.

© Copyright IBM Corp. 2005

Use the **mi_register_callback()** function to register your callback function. The callback function should track all open large objects, and in the event of an exception, close them. You can track open large objects by creating a data structure with pointers to **LLD_IO** structures, the structure that the **lld_open()** function returns when it opens an object. Use the **lld_close()** function to close open large objects.

Error Codes

This section lists and describes the Large Object Locator error codes.

Error Code	SQL State	Description
LLD_E_INTERNAL	ULLD0	Internal Large Object Locator error. If you receive this error, call IBM Informix Technical Support.
LLD_E_OK	N.A.	No error.
LLD_E_EXCEPTION	N.A.	MI_EXCEPTION raised and handled. Applies to API only.
LLD_E_SQL	N.A.	SQL error code in SQLSTATE/SQLCODE. Applies to ESQL/C interface only.
LLD_E_ERRNO	ULLD1	OS (UNIX/POSIX)
LLD_E_ROW	ULLD2	Passed an invalid MI_ROW type. The type should be lld_locator. This is an API error only.
LLD_E_PROTOCOL	ULLD3	Passed an invalid or unsupported *lo_protocol* value.
LLD_E_LOCATION	ULLD4	Passed an invalid *lo_location* value.
LLD_E_EXISTS	ULLD5	Attempted to (re)create an existing large object.
LLD_E_NOTEXIST	ULLD6	Attempted to open a nonexistent large object.
LLD_E_FLAGS	ULLD7	Used invalid flag combination when opening a large object.
LLD_E_LLDIO	ULLD8	Passed a corrupted **LLD_IO** structure.
LLD_E_RDONLY	ULLD9	Attempted to write to a large object that is open for read-only access.
LLD_E_WRONLY	ULLDA	Attempted to read from a large object that is open for write-only access.
LLD_E_SEQ	ULLDB	Attempted to seek in a large object that is open for sequential access only.
LLD_E_WHENCE	ULLDC	Invalid whence (seek) value.
LLD_E_OFFSET	ULLDD	Attempted to seek to an invalid offset.
N.A.	ULLDO	Specified an invalid lld_lob input string.
N.A.	ULLDP	Specified an invalid lld_lob type.
N.A.	ULLDQ	Attempted an invalid cast of an lld_lobtype into a BLOB or CLOB type.
N.A.	ULLDR	Used an invalid import file specification with the lld_lob type.

Part 2. MQ DataBlade Module

© Copyright IBM Corp. 2005

Chapter 6. About the MQ DataBlade Module

In This Chapter 6-1
About IBM WebSphere MQ and MQ DataBlade Module 6-1
Using MQ DataBlade Tables and Functions 6-1
Limitations 6-2
Software Requirements 6-2
Preparing to use MQ DataBlade Module 6-2
  Installing WMQ. 6-2
  Configuring the WMQ Queues 6-2
  Configuring Dynamic Server 6-3
    Add the User informix to the mqm Group 6-3
    Creating the mq Virtual Processor Class 6-3
    Reviewing the "informix".mqi* Tables 6-3
    The "informix".mqiservice Table 6-4
    The "informix".mqipubsub Table 6-4
    The "informix".mqipolicy Table 6-4
    Default Table Values 6-8
  Registering the MQ DataBlade Module 6-9
Verification 6-9
  Inserting Data into a Queue 6-9
  Reading an Entry from a Queue 6-10
  Receiving an Entry from a Queue 6-10
  Publishing and Subscribing to a Queue 6-10
    Subscribing To a Queue. 6-10
    Unsubscribing From a Queue 6-11
    Publishing to a Queue 6-11

In This Chapter

This chapter describes how to prepare, verify, and use the IBM Informix MQ DataBlade module.

About IBM WebSphere MQ and MQ DataBlade Module

IBM WebSphere MQ (WMQ) messaging products provide an infrastructure for distributed, asynchronous communication of data in a distributed, heterogeneous environment. The WMQ message queue allows you to easily exchange information across platforms.

The IBM Informix MQ DataBlade (MQ DataBlade) module provides the functionality to exchange messages between IBM Informix Dynamic Server (Dynamic Server) databases and WMQ message queues.

Using MQ DataBlade Tables and Functions

The MQ DataBlade module uses either functions or tables to communicate between a Dynamic Server application and an MQ queue. For more information on using MQ DataBlade functions, see Chapter 8, "MQ DataBlade Functions." For more information on MQ DataBlade tables, see Chapter 7, "MQ DataBlade Tables."

© Copyright IBM Corp. 2005

Limitations

MQ DataBlade module has the following limitations:

- Each Dynamic Server instance can connect to only one WMQ queue manager .
- Dynamic Server and WMQ products must be installed on the same machine.
- Non-logged databases are not supported.
- Registration to an ANSI database is not supported.

Software Requirements

To use MQ DataBlade module, the following software must be installed:

- IBM WebSphere MQ, Version 5.3
- IBM Informix Dynamic Server, Version 10.00.xC3 and above

Note: IBM Informix MQ Datablade module, Version 2.0 is distributed with Dynamic Server 10.00.xC3 and above.

The MQ DataBlade module is available on the following platforms:

- AIX 32 bit
- HP 32 bit
- Solaris 32 bit
- Windows 32 bit

Review the machines notes files for additional platform-specific information.

Note: WMQ platform requirements are independent of Dynamic Server platform requirements. For more information on respective platform requirements, see the WMQ documentation and Dynamic Server machine notes.

Preparing to use MQ DataBlade Module

The MQ DataBlade module is included with Dynamic Server. When you install Dynamic Server, the MQ DataBlade module is installed automatically.

Before you can use the MQ DataBlade module, you must complete the following tasks, each of which is explained below:

1. Install WMQ.
2. Configure that WMQ queues.
3. Configure Dynamic Server.
4. Register the MQ DataBlade module.

Installing WMQ

You must install IBM WebSphere MQ, Version 5.3 before using the MQ DataBlade module. Information on how to install WMQ is included in the WMQ product documentation.

Configuring the WMQ Queues

A WMQ queue manager is a system program providing queuing services to applications. It provides an application programming interface for programs to access messages on the queues managed by WMQ message broker. Applications can send and receive messages to and from a queue.

As necessary, you need to complete the following WMQ queue configuration:

- Create a queue manager.
- Create a queue.
- Create a subscriber queue.

For instructions on how to create a queue manager, a queue, and a subscriber queue, see the platform-specific documentation received with your WMQ product or the following website:
http://www-306.ibm.com/software/integration/mqfamily/library/manualsa/

Configuring Dynamic Server

Dynamic Server database requires the following configuration steps, each of which is explained below:

1. Add the user **informix** to the **mqm** group.
2. Create a new virtual process class called **mq**.
3. Review the "**informix**".**mqi*** tables.

Add the User informix to the mqm Group

Only members of the **mqm** group are authorized to access to WMQ queues. User **informix** must be made a member of the **mqm** group. For information on how to add user **informix** to the **mqm** group, see the platform-specific documentation for WMQ.

After adding the user **informix** to the **mqm** group and before continuing with the next step, you must shut down and restart Dynamic Server.

Creating the mq Virtual Processor Class

The VPCLASS parameter of the Dynamic Server ONCONFIG file allows you to create a class of virtual processors. You must create an **mq** VP class with the **noyield** option. Create the VP class in one of the following two ways:

- Run

  ```
  onmode -p mq +1
  ```

- Manually change the ONCONFIG file, which is located in the **$INFORMIXDIR/etc** directory. Add the following parameter to your ONCONFIG file:

  ```
  VPCLASS mq,noyield,num=1
  ```

If you manually change the ONCONFIG file, you must shut down and restart Dynamic Server for these changes to be effective.

Note: If the VP class is not created, the following error will be returned when you attempt to use the MQ DataBlade module:

```
9799: User define routine (mqread) VP context switch failed.
```

Reviewing the "informix".mqi* Tables

During registration of MQ DataBlade module, the following three tables are created:

- "**informix**".**mqiservice**
- "**informix**".**mqipubsub**
- "**informix**".**mqipolicy**

These tables create the definitions for services and policies and are referred to by MQ DataBlade module. Each table is explained below.

The "informix".mqiservice Table

The "**informix**".**mqiservice** table creates the service definitions for service point (sender/receiver) attributes.

The "**informix**".**mqiservice** table has the following schema:

```
CREATE TABLE "informix".mqiservice
    servicename    LVARCHAR(256),
    queuemanager   VARCHAR(48) NOT NULL,
    queuename      VARCHAR(48) NOT NULL,
    defaultformat  VARCHAR(8) default ' ',
    ccsid          VARCHAR(6) default ' ',
    PRIMARY KEY (servicename) );
```

The attributes are defined as follows:

servicename is the service name used in the MQ functions.

queuemanager is the queue manager service provider.

queuename is the queue name to send the message to or receive the message from.

defaultformat defines the default format.

ccsid is the coded character set identifier of the destination application.

The "informix".mqipubsub Table

The "**informix**".**mqipubsub** table creates the policy definitions for the following attributes:

- Distribution list
- Receive
- Subscriber
- Publisher

The "**informix**".**mqipubsub** table has the following schema:

```
CREATE TABLE "informix".mqipubsub
    pubsubname      LVARCHAR(256) NOT NULL UNIQUE,
    servicebroker   LVARCHAR(256),
    receiver        LVARCHAR(256) default ' ',
    psstream        LVARCHAR(256) default ' ',
    pubsubtype      VARCHAR(20) CHECK (pubsubtype IN ('Publisher', 'Subscriber')),
FOREIGN KEY (servicebroker) REFERENCES "informix".mqiservice(servicename));
```

The attributes are defined as follows:

pubsubname is the name of the publish/subscribe service.

servicebroker is the service name of the publish/subscribe service.

receiver is the queue on which to receive messages after subscription.

psstream is the stream coordinating the publish/subscribe service.

pubsubtype is the service type.

The "informix".mqipolicy Table

The "**informix**".**mqipolicy** table creates the policy definitions for the following attributes:

- General
- Publish
- Receive

- Reply
- Send
- Subscribe

The "**informix**".**mqipolicy** table has the following schema:

```
CREATE TABLE "informix".mqipolicy
    policyname          VARCHAR(128) NOT NULL,
    messagetype         CHAR(1) DEFAULT 'D' CHECK (messagetype IN ('D', 'R')),
    messagecontext      CHAR(1) DEFAULT 'Q' CHECK (messagecontext IN
                           ('Q','P','A','N')),
    snd_priority        CHAR(1) DEFAULT 'T' CHECK (snd_priority IN
                           ('0','1','2','3','4', '5','6','7','8','9', 'T')),
    snd_persistence     CHAR(1) DEFAULT 'T' CHECK (snd_persistence IN
                           ('Y','N','T')),
    snd_expiry          INTEGER DEFAULT -1 CHECK ( snd_expiry > 0 OR snd_expiry
                            = -1 ),
    snd_retrycount      INTEGER DEFAULT 0 CHECK ( snd_retrycount >= 0 ),
    snd_retry_intrvl    INTEGER DEFAULT 1000 CHECK ( snd_retry_intrvl >= 0 ),
    snd_newcorrelid     CHAR(1) DEFAULT 'N' CHECK ( snd_newcorrelid IN ('Y','N')),
    snd_resp_correlid   CHAR(1) DEFAULT 'M' CHECK ( snd_resp_correlid IN ('M','C')),
    snd_xcption_action  CHAR(1) DEFAULT 'Q' CHECK ( snd_xcption_action IN
                           ('Q','D')),
    snd_report_data     CHAR(1) DEFAULT 'R' CHECK ( snd_report_data IN
                           ('R','D','F')),
    snd_rt_exception    CHAR(1) DEFAULT 'N' CHECK ( snd_rt_exception IN ('Y','N')),
    snd_rt_coa          CHAR(1) DEFAULT 'N', CHECK ( snd_rt_coa IN ('Y','N')),
    snd_rt_cod          CHAR(1) DEFAULT 'N' CHECK ( snd_rt_cod IN ('Y','N')),
    snd_rt_expiry       CHAR(1) DEFAULT 'N' CHECK ( snd_rt_expiry IN ('Y','N')),
    reply_q             VARCHAR(48) DEFAULT 'SAME AS INPUT_Q',
    reply_qmgr          VARCHAR(48) DEFAULT 'SAME AS INPUT_QMGR',
    rcv_truncatedmsg    CHAR(1) DEFAULT 'N' CHECK ( rcv_truncatedmsg IN ('Y','N')),
    rcv_convert         CHAR(1) DEFAULT 'Y' CHECK ( rcv_convert IN ('Y','N')),
    rcv_poisonmsg       CHAR(1) DEFAULT 'N' CHECK ( rcv_poisonmsg IN ('Y','N')),
    rcv_openshared      CHAR(1) DEFAULT 'Q' CHECK ( rcv_openshared IN
                           ('Y','N','Q')),
    rcv_wait_intrvl     INTEGER DEFAULT 0 CHECK ( rcv_wait_intrvl >= -1 ),
    pub_suppressreg     CHAR(1) DEFAULT 'Y' CHECK ( pub_suppressreg IN ('Y','N')),
    pub_anonymous       CHAR(1) DEFAULT 'N' CHECK ( pub_anonymous IN ('Y','N')),
    pub_publocal        CHAR(1) DEFAULT 'N' CHECK ( pub_publocal IN ('Y','N')),
    pub_direct          CHAR(1) DEFAULT 'N' CHECK ( pub_direct IN ('Y','N')),
    pub_correlasid      CHAR(1) DEFAULT 'N' CHECK ( pub_correlasid IN ('Y','N')),
    pub_retain          CHAR(1) DEFAULT 'N' CHECK ( pub_retain IN ('Y','N')),
    pub_othersonly      CHAR(1) DEFAULT 'N' CHECK ( pub_othersonly IN ('Y','N')),
    sub_anonymous       CHAR(1) DEFAULT 'N' CHECK ( sub_anonymous IN ('Y','N')),
    sub_sublocal        CHAR(1) DEFAULT 'N' CHECK ( sub_sublocal IN ('Y','N')),
    sub_newpubsonly     CHAR(1) DEFAULT 'N' CHECK ( sub_newpubsonly IN ('Y','N')),
    sub_pubonreqonly    CHAR(1) DEFAULT 'N' CHECK ( sub_pubonreqonly IN ('Y','N')),
    sub_correlasid      CHAR(1) DEFAULT 'N' CHECK ( sub_correlasid IN ('Y','N')),
    sub_informifret     CHAR(1) DEFAULT 'Y' CHECK ( sub_informifret IN ('Y','N')),
    sub_unsuball        CHAR(1) DEFAULT 'N' CHECK ( sub_unsuball IN ('Y','N')),
    PRIMARY KEY (policyname) );
```

The attributes are defined as follows:

policyname
: is the name of the policy.

messagetype
: is the type of message.

messagecontext
: defines how the message context is set in messages sent by the application:

  - The default is `Set By Queue Manager` (the queue manager sets the context).

- If set to `Pass Identity`, the identity of the request message is passed to any output messages.
- If set to `Pass All`, all the context of the request message is passed to any output messages.
- If set to `No Context`, no context is passed.

snd_priority
is the priority set in the message, where 0 is the lowest priority and 9 is the highest. When set to `As Transport`, the value from the queue definition is used. You must deselect `As Transport` before you can set a priority value.

snd_persistence
is the persistence set in the message, where `Yes` is persistent and `No` is not persistent. When set to `As Transport`, the value from the underlying queue definition is used.

snd_expiry
is a period of time (in tenths of a second) after which the message will not be delivered.

snd_retrycount
is the number of times a send will be retried if the return code gives a temporary error. Retry is attempted under the following conditions: Queue full, Queue disabled for put, Queue in use.

snd_retry_intrvl
is the interval (in milliseconds) between each retry.

snd_newcorrelid
is whether each message is sent with a new correlation ID (except for response messages, where this is set to the Message ID or Correl ID of the request message).

snd_resp_correlid
is the ID set in the Correl ID of a response or report message. This is set to either the Message ID or the Correl ID of the request message, as specified.

snd_xcption_action
is the action when a message cannot be delivered. When set to `DLQ`, the message is sent to the dead-letter queue. When set to `Discard`, the message is discarded.

snd_report_data
is the amount of data included in a report message, where `Report` specifies no data, `With Data` specifies the first 100 bytes, and `With Full Data` specifies all data.

snd_rt_exception
is whether Exception reports are required.

snd_rt_coa
is whether Confirm on Arrival reports are required.

snd_rt_cod
is whether Confirm on Delivery reports are required.

snd_rt_expiry
is whether Expiry reports are required.

reply_q is the name of the reply queue.

reply_qmgr
is the name of the reply Queue Manager.

rcv_truncatedmsg
is whether truncated messages are accepted.

rcv_convert
is whether the message is code page converted by the message transport when received.

rcv_poisonmsg
is whether poison message handling is enabled.

rcv_openshared
is whether the queue is opened as a shared queue.

rcv_wait_intrvl
is a period of time (in milliseconds) that the receive waits for a message to be available.

pub_suppressreg
is whether implicit registration of the publisher is suppressed. (This attribute is ignored for WebSphere MQ Integrator Version 2.)

pub_anonymous
is whether the publisher registers anonymously.

pub_publocal
is whether the publication is only sent to subscribers that are local to the broker.

pub_direct
is whether the publisher should accept direct requests from subscribers.

pub_correlasid
is whether the Correl ID is used by the broker as part of the publisher's identity.

pub_retain
is whether the publication is retained by the broker.

pub_othersonly
is whether the publication is not sent to the publisher if it has subscribed to the same topic (used for conference-type applications).

sub_anonymous
is whether the subscriber registers anonymously.

sub_sublocal
is whether the subscriber is sent publications that were published with the `Publish Locally` option, at the local broker only.

sub_newpubsonly
is whether the subscriber is not sent existing retained publications when it registers.

sub_pubonreqonly
is whether the subscriber is not sent retained publications, unless it requests them by using Request Update.

sub_correlasid
is the broker as part of the subscriber's identity.

sub_informifret
is whether the broker informs the subscriber if a publication is retained.

sub_unsuball
is whether all topics for this subscriber are to be deregistered.

Default Table Values

Most of the MQ DataBlade functions have an optional *policy* and *service* parameter. If the parameter is not passed, the default value is used. Table 6-1 lists the default value that MQ DataBlade module will use.

Table 6-1. Default Values

Type	Name	Resources	Notes
Service	IDS.DEFAULT.SERVICE	IDS.DEFAULT.QUEUE	created
Service	IDS.DEFAULT.SUBSCRIBER	SYSTEM.BROKER.CONTROL.QUEUE	system default
Service	IDS.DEFAULT.PUBLISHER	SYSTEM.BROKER.DEFAULT.STREAM	system default
Service	IDS.DEFAULT.SUBSCRIBER.RECEIVER	IDS.DEFAULT.SUBSCRIBER.RECEIVER.QUEUE	created
Policy	IDS.DEFAULT.POLICY	*connection name :default queuemanager*	system default
Publisher	IDS.DEFAULT.PUBLISHER	sender:IDS.DEFAULT.PUBLISHER	system default
Subscriber	IDS.DEFAULT.SUBSCRIBER	sender:IDS.DEFAULT.SUBSCRIBER receiver: IDS.DEFAULT.SUBSCRIBER.RECEIVER	system default

Each service definition includes a queue specification, as listed in Table 6-1 above. The service can be mapped any queue. For testing purposes, you can create the following queues using the script **idsdefault.tst**:

- IDS.DEFAULT.QUEUE queue for the IDS.DEFAULT.SERVICE
- IDS.DEFAULT.SUBSCRIBER.RECIVER.QUEUE queue for the IDS.DEFAULT.SUBSCRIBER

The script **idsdefault.tst** is located in the **MQBLADE** directory. Use the **runmqsc** utility to execute commands in **idsdefault.tst**.

If the QueueManager is not a default queue manager, you must update the **queuemanager** column of the **informix.mqiservice** table by updating **servicename** to IDS.DEFAULT.SERVICE, IDS.DEFAULT.PUBLISHER, IDS.DEFAULT.SUBSCRIBER and IDS.DEFAULT.SUBSCRIBER.RECEIVER.

During registration of the MQ DataBlade module, the following default values are inserted into the "**informix**".**mqi*** tables :

```
INSERT INTO ""informix"".mqiservice(servicename, queuemanager, queuename)
       VALUES('IDS.DEFAULT.SERVICE', '', 'IDS.DEFAULT.QUEUE');

INSERT INTO ""informix"".mqiservice(servicename, queuemanager, queuename)
       VALUES('IDS.DEFAULT.PUBLISHER', '', 'SYSTEM.BROKER.DEFAULT.STREAM');

INSERT INTO ""informix"".mqiservice(servicename, queuemanager, queuename)
       VALUES('IDS.DEFAULT.SUBSCRIBER', '', 'SYSTEM.BROKER.CONTROL.QUEUE');

INSERT INTO ""informix"".mqiservicesservicename, queuemanager, queuename)
       VALUES('IDS.DEFAULT.SUBSCRIBER.RECEIVER', '',
              'IDS.DEFAULT.SUBSCRIBER.RECEIVER.QUEUE');

INSERT INTO ""informix"".mqipubsub(pubsubname, servicebroker, receiver,
                                   psstream, pubsubtype)
```

```
        VALUES('IDS.DEFAULT.SUBSCRIBER', 'IDS.DEFAULT.SUBSCRIBER',
               'IDS.DEFAULT.SUBSCRIBER.RECEIVER',
               'SYSTEM.BROKER.DEFAULT.STREAM', 'Subscriber');

INSERT INTO ""informix"".mqipubsub(pubsubname, servicebroker, receiver,
                                   psstream, pubsubtype)
       VALUES('IDS.DEFAULT.PUBLISHER', 'IDS.DEFAULT.PUBLISHER', '', '',
              'Publisher');

INSERT INTO ""informix"".mqipolicy(policyname)
       VALUES('IDS.DEFAULT.POLICY');

INSERT INTO ""informix"".mqipolicy(policyname)
       VALUES('IDS.DEFAULT.PUB.SUB.POLICY');
```

Registering the MQ DataBlade Module

The MQ DataBlade module is distributed with IBM Informix Dynamic Server Versions 10.00.xC3 and above. Use BladeManager to register the MQ DataBlade module, 2.0 in each database from which your want to access WMQ. See the *IBM Informix: DataBlade Module Installation and Registration Guide* for more information.

Note: Information on how to unregister your datablade is also available in the *IBM Informix: DataBlade Module Installation and Registration Guide*. During unregistration, the "**informix**"**.mqi*** tables are dropped. You must unload the table data and re-load the tables again.

Verification

This section includes sample code to enable you to verify MQ DataBlade module functionality. For more information about all of the functions used below, see Chapter 8, "MQ DataBlade Functions."

The following actions are described in this section:

- Inserting data into a queue
- Reading an entry from a queue
- Receiving an entry from a queue
- Publishing and subscribing to a queue

MQ DataBlade functions must be used within a transaction. For MQ DataBlade functions using the EXECUTE statement, you must explicitly start the transaction with a BEGIN WORK statement. For MQ DataBlade functions using the SELECT, UPDATE, DELETE, or INSERT statements, you do not need to use a BEGIN WORK statement.

Inserting Data into a Queue

The service IDS.DEFAULT.SERVICE specifies the IDS.DEFAULT.QUEUE. Before inserting data into the queue, you should check the size of the queue. After inserting the data, you should check the queue to confirm that the data was added.

```
BEGIN WORK;

EXECUTE FUNCTION MQSend('IDS.DEFAULT.SERVICE', 'IDS.DEFAULT.POLICY', 'hello queue');

(expression)          1
1 row(s) retrieved.

COMMIT WORK;
```

Reading an Entry from a Queue

The **MQRead()** function reads a message from the queue but does not remove it. After reading the message, the queue has not been changed:

```
BEGIN WORK;

EXECUTE FUNCTION MQRead('IDS.DEFAULT.SERVICE', 'IDS.DEFAULT.POLICY');

(expression) hello queue

1 row(s) retrieved.

COMMIT WORK;
```

The following example reads a message from the queue and inserts it into a database table:

```
INSERT into msgtable values (MQRead('IDS.DEFAULT.SERVICE', 'IDS.DEFAULT.POLICY'));

1 row(s) inserted.

SELECT * from msgtable;

msg hello queue

1 row(s) retrieved.

COMMIT WORK;
```

Receiving an Entry from a Queue

The **MQReceive()** function removes the message from the queue, as in the following example:

```
BEGIN WORK;

EXECUTE FUNCTION MQReceive('IDS.DEFAULT.SERVICE', 'IDS.DEFAULT.POLICY');

(expression)  hello queue

1 row(s) retrieved.

COMMIT WORK;
```

Publishing and Subscribing to a Queue

Publishing and subscribing to a queue is an effective way of exchanging information between multiple users. The MQ DataBlade module interacts directly with the WMQ Publish/Subscribe component. The component allows a message to be sent to multiple subscribers based on a topic. Users subscribe to a topic, and when a publisher inserts a message with that topic into the queue, the WMQ broker routes the messages to all of the queues of each specified subscriber. Then, the subscriber retrieves the message from the queue.

The following actions are described in this section:

- Subscribing to a queue
- Unsubscribing from a queue
- Publishing to a queue

Subscribing To a Queue

To subscribe to a queue, use the **MQSubscribe()** function. The following example shows how a database application subscribes to a queue to receive messages for a topic named **Weather**:

```
   --- before subscribe
   Topic: MQ/TIMESERIES.QUEUE.MANAGER          /StreamSupport
   Topic: MQ/S/TIMESERIES.QUEUE.MANAGER        /Subscribers/Identities/*
   Topic: MQ/S/TIMESERIES.QUEUE.MANAGER        /Subscribers/Identities/*
   Topic: MQ/S/TIMESERIES.QUEUE.MANAGER        /Subscribers/Identities/*

BEGIN WORK;

EXECUTE FUNCTION MQSubscribe('AMT.SAMPLE.SUBSCRIBER', 'AMT.SAMPLE.PUB.SUB.POLICY',
'Weather');

(expression)          1

1 row(s) retrieved.

   --- after subscribe
   Topic: MQ/TIMESERIES.QUEUE.MANAGER          /StreamSupport
   Topic: MQ/S/TIMESERIES.QUEUE.MANAGER        /Subscribers/Identities/*
   Topic: Weather
   Topic: MQ/S/TIMESERIES.QUEUE.MANAGER        /Subscribers/Identities/*
   Topic: MQ/S/TIMESERIES.QUEUE.MANAGER        /Subscribers/Identities/*

COMMIT WORK;
```

Unsubscribing From a Queue

To unsubscribe from a queue, use the **MQUnsubscribe()** function, as in the following example:

```
BEGIN WORK;

EXECUTE FUNCTION MQUnsubscribe('AMT.SAMPLE.SUBSCRIBER', 'AMT.SAMPLE.PUB.SUB.POLICY',
'Weather');(

expression)          1

1 row(s) retrieved.
   Topic: MQ/TIMESERIES.QUEUE.MANAGER            /StreamSupport
   Topic: MQ/S/TIMESERIES.QUEUE.MANAGER          /Subscribers/Identities/*
   Topic: MQ/S/TIMESERIES.QUEUE.MANAGER          /Subscribers/Identities/*
   Topic: MQ/S/TIMESERIES.QUEUE.MANAGER          /Subscribers/Identities/*

COMMIT WORK;
```

Publishing to a Queue

To publish to a queue, use the **MQPublish()** function, as in the following example:

```
BEGIN WORK;

EXECUTE FUNCTION MQPublish('IDS.DEFAULT.SERVICE', 'IDS.DEFAULT.POLICY', 'Weather');

(expression)          1

COMMIT WORK;
```

Chapter 7. MQ DataBlade Tables

In This Chapter 7-1
MQ DataBlade Tables 7-1
Working with Tables 7-1
Schema Mapping 7-1
General Table Behavior 7-2
Creating and Binding a Table 7-2
Using INSERT and SELECT. 7-2
Retrieving the Queue Element 7-3
Special Considerations 7-3
Table Errors 7-4

In This Chapter

This chapter describes how to map WMQ queues to Dynamic Server relational tables and how to use those tables.

MQ DataBlade Tables

The MQ DataBlade module provides the IBM Informix Virtual-Table Interface (VTI) access method to access WMQ queues using Dynamic Server table semantics. VTI binds tables to WMQ queues, creating transparent access to WMQ objects and enabling users to access the queue as if it were a table. For more information on VTI, see the *IBM Informix: Virtual-Table Interface Programmer's Guide*.

Working with Tables

This section describes how to work with tables:

- Schema mapping
- General table behavior
- Creating and binding a table
- Using INSERT and SELECT statements
- Retrieving the queue element
- Special considerations
- Table errors

Schema Mapping

When a table is bound to a WMQ queue, the schema is mapped directly to WMQ objects. The following table shows the mapping of schema to WMQ objects.

Table 7-1. Schema Mapping to WMQ Objects

Name	Type	Description
msg	lvarchar(*maxMessage*)	The message being sent or received. The default size is 4,000; the limit is 32,739.
correlid	varchar(24)	The correlation ID, which can be used as a qualifier
topic	varchar(40)	The topic used with publisher or subscriber, which can be used as a qualifier
qname	varchar(48)	The name of the queue

© Copyright IBM Corp. 2005

Table 7-1. Schema Mapping to WMQ Objects (continued)

Name	Type	Description
msgid	varchar(12)	The message ID
msgformat	varchar(8)	The message format

General Table Behavior

For every table created, the following applies:

- The PUBLIC group is limited to SELECT privileges. Only the database administrator and the table creater have INSERT privileges.
- When a function is first invoked in each user session, WMQ metadata tables are read and their values are cached in the PER_SESSION memory. The cache is not refreshed until the session closes or the database is closed and reopened.

Creating and Binding a Table

Use the **MQCreateVtiReceive()** function to create a table and bind it to a queue. The following example creates a table named **vtimq**, and binds it to the queue defined by service **IDS.DEFAULT.SERVICE** and policy **IDS.DEFAULT.POLICY**.

```
BEGIN WORK;

EXECUTE FUNCTION MQICreateVtiReceive ("VtiMQ",
               "IDS.DEFAULT.SERVICE",  "IDS.DEFAULT.POLICY");
```

Using a SELECT statement on a table created with **MQCreateVtiReceive()**, results in a message is received from the table, which is the equivalent of calling the **MQReceive()** function on the queue. For both functions, the messages selected are removed from the quueue.

To browse the messages on the queue without removing the messages from the queue, use the **MQCreateVtiRead()** function. to bind the table **vtimq** to a queue. In the following example, **MQCreateVtiRead()** binds the table **vtimq** to a queue:

```
BEGIN WORK;

EXECUTE FUNCTION MQCreateVtiRead (vtimq, read-service, policy, maxMessage)
```

For complete information on the **MQCreateVtiRead()** or **MQCreateVtiReceive()** functions, see Chapter 8, "MQ DataBlade Functions."

Using INSERT and SELECT

After a table is bound to a queue, use INSERT to insert items into the WMQ queue, and SELECT to retrieve WMQ messages.

Using the example with table **vtimq** above, the following example inserts a message into the **msg** column of **VtiMQ** and into the queue described by IDS.DEFAULT.SERVICE service and policy IDS.DEFAULT.POLICY:

```
INSERT into VtiMQ (msg) values ('PUT on queue with SQL INSERT');
1 row(s) inserted.
```

Use a SELECT statement to display the message:

```
SELECT * from VtiMQ;
msg        PUT on queue with SQL INSERT
correlid
```

```
topic
qname       IDS.DEFAULT.QUEUE
msgid       AMQ
msgformat   MQSTR
```

Retrieving the Queue Element

Use the **MQRead()** function to retrieve the queue element:

```
BEGIN WORK;

EXECUTE FUNCTION MQRead('IDS.DEFAULT.SERVICE', 'IDS.DEFAULT.POLICY');
(expression)  PUT on queue with SQL INSERT
1 row(s) retrieved.
COMMIT WORK
```

Special Considerations

Binding a table to a queue creates a useful interface between the queue and the database. However, due to the inherent limitations of a queue, not all database functionality can be used.

When a message is fetched from a queue, the default database processing is to dequeue, or remove, it. Every time a queue is read by the database, the data within the queue changes. This behavior differs from a standard read by a database, in which the data does not change. Supplying only a mapping that enables users to browse, where reading does not remove the queue, eliminates a major queue functionality. Enabling both processing models provides more options and requires corresponding responsibility.

By default, the top element is removed when a message is fetched from a queue. WMQ allows messages to be retrieved based upon a *correlid*. A *correlid* is a correlation identifier that can be used as a key, for example, to correlate a response message to a request message. If the *correlid* of the message matches the *correlid* of a request, the message is returned. If the VTI table is qualified with the *correlid* column, the *correlid* qualifier is passed into the WMQ request to fetch a value.

In the following example, a queue has three messages and only the second message contains a *correlid*, which is named **'fred'**. The following statement removes all three messages from the queue and places them in a table named **flounder**:

```
INSERT into flounder (deQueuedMsg) values (SELECT msg from vtimq);
```

When execution completes, no messages remain on the queue and three new rows appear in the **flounder** table.

The following example qualifies the **vtimq** table:

```
INSERT into flounder (deQueuedMsg) values (SELECT msg from vtimq where
correlid = 'fred');
```

The above statement creates two groups of messages:

- Messages that failed the ***correlid* = 'fred'** qualification
- Messages that passed the ***correlid* = 'fred'** qualification. The one message that passed the qualification is located in the **flounder** table.

Statements including qualifiers other than equality (=) or NULL return an error. Statements including NULL return unexpected results.

Table Errors

Tables that are mapped to WMQ can generate non-database errors if the underlying WMQ request fails. In the example below, a VTI mapping was established using a bad service definition, and the error was not recognized until a SELECT statement was executed against the table.

```
BEGIN WORK;
EXECUTE FUNCTION MQCreateVtiReceive('vtiTable',"BAD.SERVICE");
SELECT * from vtitable;

(MQ015) - FUNCTION:MqiGetServicePolicy, SERVICE:BAD.SERVICE,
POLICY:IDS.DEFAULT.POLICY ::
BAD.SERVICE is not present in the database "informix".MQISERVICE table.
Error in line 1
Near character position 23
```

Chapter 8. MQ DataBlade Functions

In This Chapter . . . 8-2
MQ DataBlade Functions Overview . . . 8-2
MQ DataBlade Functions . . . 8-2
MQCreateVtiRead() . . . 8-2
Syntax . . . 8-2
Usage . . . 8-3
Return Codes . . . 8-4
Example . . . 8-4
MQCreateVtiReceive() . . . 8-4
Syntax . . . 8-4
Usage . . . 8-5
Return Codes . . . 8-6
Example . . . 8-6
MQPublish() . . . 8-6
Syntax . . . 8-6
Usage . . . 8-7
Return Codes . . . 8-8
Examples . . . 8-8
MQRead() . . . 8-10
Syntax . . . 8-10
Usage. . . . 8-10
Return Codes . . . 8-11
Examples . . . 8-11
MQReadClob() . . . 8-12
Syntax . . . 8-12
Usage. . . . 8-13
Return Codes . . . 8-13
Examples . . . 8-13
MQReceive() . . . 8-14
Syntax . . . 8-14
Usage. . . . 8-15
Return Codes . . . 8-15
Examples . . . 8-15
MQReceiveClob() . . . 8-17
Syntax . . . 8-17
Usage. . . . 8-17
Return Codes . . . 8-18
Examples . . . 8-18
MQSend() . . . 8-19
Syntax . . . 8-19
Usage. . . . 8-20
Return Codes . . . 8-20
Examples . . . 8-20
MQSendClob() . . . 8-21
Syntax . . . 8-21
Usage. . . . 8-22
Return Codes . . . 8-22
Examples . . . 8-22
MQSubscribe() . . . 8-23
Syntax . . . 8-23
Usage. . . . 8-24
Return Codes . . . 8-24
Examples . . . 8-24
MQTrace() . . . 8-25
Syntax . . . 8-26

© Copyright IBM Corp. 2005

Examples 8-26
MQUnsubscribe() 8-27
Syntax 8-27
Usage. 8-27
Return Codes 8-28
Examples 8-28
MQVersion() 8-28
Syntax 8-28
Example 8-28

In This Chapter

This chapter describes each MQ DataBlade function and provides detailed information about each function's syntax and usage.

MQ DataBlade Functions Overview

The MQ DataBlade module provides functions to enable Dynamic Server applications to exchange data directly between the application and Websphere MQ. These MQ DataBlade functions can be executed within SQL statements and should have an explicit or implicit transactional context. Examples of SQL code using MQ DataBlade functions are included in the next section.

MQ DataBlade Functions

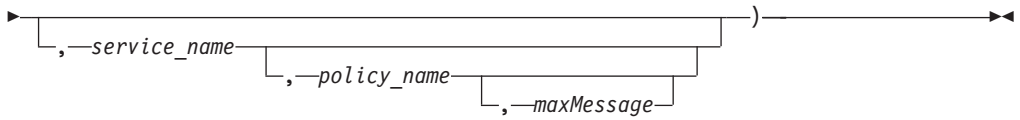
When you register the MQ DataBlade module in a database, the following functions become available:

- MQCreateVtiRead()
- MQCreateVtiReceive()
- MQPublish()
- MQRead()
- MQReadClob()
- MQReceive()
- MQReceiveClob()
- MQSend()
- MQSendClob()
- MQSubscribe()
- MQTrace()
- MQUnsubscribe()
- MQVersion()

MQCreateVtiRead()

The **MQCreateVtiRead()** function creates a table and maps it to a queue managed by WMQ.

Syntax



table_name
Required parameter. Specifies the name of the table to be created. The queue pointed to by the *service_name* parameter is mapped to this table.

service_name
Optional parameter. Refers to the value in the **servicename** column of the "**informix**".**mqiservice** table. If *service_name* is not specified, IDS.DEFAULT.SERVICE is used as the service. The maximum size of *service_name* is 48 bytes.

policy_name
Optional parameter. Refers to the value in the **policyname** column of the "**informix**".**mqipolicy** table. If *policy_name* is not specified, IDS.DEFAULT.POLICY is used as the policy. The maximum size of *policy_name* is 48 bytes.

maxMessage
Optional parameter. Specifies the maximum length of the message to be sent or received. The default value is `4000`; the maximum allowable size is `32,739`.

Usage

The **MQCreateVtiRead()** function creates a table bound to a queue specified by *service_name*, using the quality of service policy defined in *policy_name*. Selecting from the table created by this function returns all the committed messages in the queue, but does not remove the messages from the queue. If no messages are available to be returned, the SELECT statement returns no rows. An insert to the bound table puts a message into the queue.

The table created has the following schema and uses the "**informix**".**mq** access method:

```
create table table_name (
       msg lvarchar(maxMessage),
       correlid varchar(24),
       topic varchar(40),
       qname varchar(48),
       msgid varchar(12),
       msgformat varchar(8));
             using "informix".mq (SERVICE = service_name,
                                  POLICY = policy_name,
                                  ACCESS = "READ");
```

The mapping for a table bound to a queue requires translation of operation. Actions on specific columns within the table are translated into specific operations within the queue, as outlined here:

- An insert operation inserts the following into the mapped table column:
  - **msg.** The message text that will be inserted onto the queue.
  - **correlid.** The message will be sent with the specified correlation identifier.
- A select operation maps these in the following way to a WMQ queue:
  - **msg.** The message is retrieved from the queue

– **correlid.** Within the WHERE clause, is the value passed to the queue manager to qualify messages (the correlation identifier). The only operator that should be used when qualifying is equals (=).

The following table describes how the arguments for the **MQCreateVtiRead()** function are interpreted.

Table 8-1. MQCreateVtiRead() argument interpretation

Usage	Argument Interpretation
MQCreateVtiRead(arg1)	arg1 = *table_name*
MQCreateVtiRead(arg1, arg2)	arg1 = *table_name* arg2 = *service_name*
MQCreateVtiRead(arg1, arg2, arg3)	arg1 = *table_name* arg2 = *service_name* arg3 = *policy_name*
MQCreateVtiRead(arg1, arg2, arg3, arg4)	arg1 = *table_name* arg2 = *service_name* arg3 = *policy_name* arg4 = *maxMessage*

Return Codes

't' The operation was successful.

'f' The operation was unsuccessful.

Example

Create a table called **VtiReadTest** using the default service name and policy name:

```
begin;
EXECUTE FUNCTION MQCreateVtiRead('VtiReadTest');
commit;
```

Insert a message into the queue:

```
INSERT INTO VtiReadTest(msg) values ('QMessage');
```

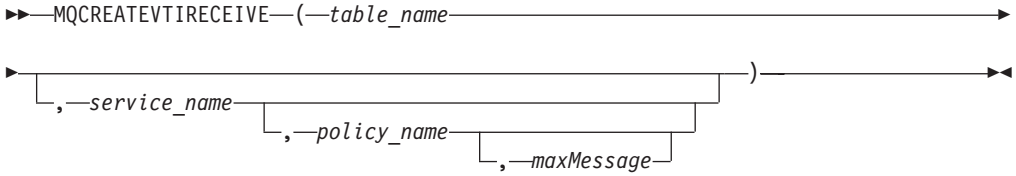
Read a message from the queue:

```
select * from VtiReadTest;
```

MQCreateVtiReceive()

The **MQCreateVtiReceive()** function creates a table and maps it to a queue managed by WMQ.

Syntax



table_name
Required parameter. Specifies the name of the table to be created. The queue pointed to by the *service_name* parameter is mapped to this table.

service_name
Optional parameter. Refers to the value in the **servicename** column of the "**informix**".**mqiservice** table. If *service_name* is not specified, IDS.DEFAULT.SERVICE is used as the service. The maximum size of *service_name* is 48 bytes.

policy_name
Optional parameter. Refers to the value in the **policyname** column of the "**informix**".**mqipolicy** table. If *policy_name* is not specified, IDS.DEFAULT.POLICY is used as the policy. The maximum size of *policy_name* is 48 bytes.

maxMessage
Optional parameter. Specifies the maximum length of the message to be sent or received. The default value is `4000`; the maximum allowable size is `32,739`.

Usage

The **MQCreateVtiReceive()** function creates a *table_name* bound to a queue specified by *service_name*, using the quality of service policy defined in *policy_name*. Selecting from this table returns all the available messages in the queue and also removes the messages from the queue. If no messages are available to be returned, the no rows are returned. An insert into the bound table puts messages in the queue.

The table created has the following schema and uses the "**informix**".**mq** access method:

```
create table table_name (
       msg lvarchar(maxMessage),
       correlid varchar(24),
       topic varchar(40),
       qname varchar(48),
       msgid varchar(12),
       msgformat varchar(8));
              using "informix".mq (SERVICE = service_name,
                                   POLICY = policy_name,
                                   ACCESS = "RECEIVE");
```

The mapping between a table bound to a queue requires translation of operation. Actions on specific columns within the table are translated into specific operations within the queue, as outlined here:

- An insert operation maps the following columns to the MQ manager:
  - **msg.** The text that will be inserted onto the queue
  - **correlid.** The key recognized by queue manager to get messages from the queue
- A select operation maps the following columns to the MQ manager:
  - **msg.** The message is removed from the queue.
  - **correlid.** Within the WHERE clause, is the value passed to the queue manager to qualify messages (the correlation identifier). The only operator that should be used when qualifying is equals (=).

The following table describes how the arguments for the **MQCreateVtiReceive()** function are interpreted.

Table 8-2. MQCreateVtiReceive() argument interpretation

Usage	Argument Interpretation
MQCreateVtiReceive(arg1)	arg1 = *table_name*
MQCreateVtiReceive(arg1, arg2)	arg1 = *table_name* arg2 = *service_name*
MQCreateVtiReceive(arg1, arg2, arg3)	arg1 = *table_name* arg2 = *service_name* arg3 = *policy_name*
MQCreateVtiReceive(arg1, arg2, arg3, arg4)	arg1 = *table_name* arg2 = *service_name* arg3 = *policy_name* arg4 = *maxMessage*

Return Codes

't' The operation was successful.

'f' The operation was unsuccessful.

Example

Create the table **VtiReceiveTest** using the default service name and policy name:

```
begin;
EXECUTE FUNCTION MQCreateVtiRead('VtiReceiveTest');
commit;
```

Insert a message to the queue:

```
INSERT INTO VtiReceiveTest(msg) values ('QMessage');
```

Read a message from the queue:

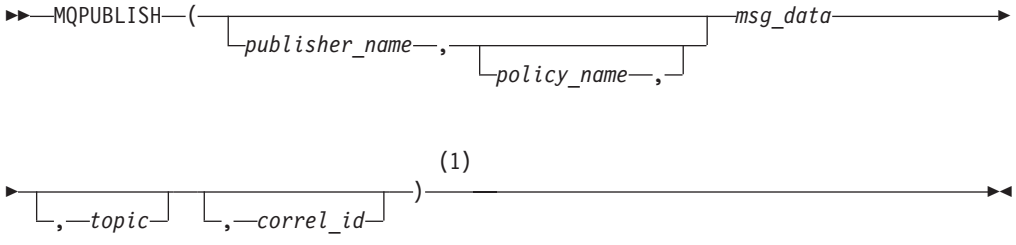
```
select * from VtiReceiveTest;
```

Attempting to read the queue a second time results in returning no rows because the table was created using the **MQCreateVtiReceive()** function, which removes entries as they are read.

MQPublish()

The **MQPublish()** function publishes a message on one or more topics to a queue managed by WMQ.

Syntax



Notes:

1 See the Usage section for argument interpretation.

publisher_name
Optional parameter. Refers to the value in the **pubsubname** column of the "**informix**".**mqipubsub** table. If *publisher_name* is not specified, IDS.DEFAULT.PUBLISHER is used as the publisher. The maximum length of *publisher_name* is 48 bytes.

policy_name
Optional parameter. Refers to the value in the **policyname** column of the "**informix**".**mqipolicy** table. If *policy_name* is not specified, IDS.DEFAULT.PUB.SUB.POLICY is used as the policy. The maximum size of *policy_name* is 48 bytes.

msg_data
Required parameter. A string containing the data to be sent by WMQ. The maximum size of the string is defined by the LVARCHAR data type.

topic Optional parameter. A string containing the topic for the message publication. The maximum size of a topic is 40 bytes. Multiple topics can be specified in one string (up to 40 characters long). Each topic must be separated by a colon. For example, "t1:t2:the third topic" indicates that the message is associated with all three topics: t1, t2, and the third topic. If no topic is specified, none are associated with the message.

correl_id
Optional parameter. A string containing a correlation identifier to be associated with this message. The *correl_id* is often specified in request and reply scenarios to associate requests with replies. The maximum size of *correl_id* is 24 bytes. If not specified, no correlation ID is added to the message.

Usage

The **MQPublish()** function publishes data to WMQ. It requires the installation of the WMQ Publish/Subscribe component of WMQ, and that the Message Broker is running.

The **MQPublish()** function publishes the data contained in *msg_data* to the WMQ publisher specified in *publisher_name*, using the quality of service policy defined by *policy_name*.

The following table describes how the arguments for the **MQPublish()** function are interpreted.

Table 8-3. MQPublish() argument interpretation

Usage	Argument Interpretation
MQPublish(arg1)	arg1 = *msg_data*
MQPublish(arg1, arg2)	arg1 = *msg_data* arg2 = *topic*
MQPublish(arg1, arg2, arg3)	arg1 = *publisher_name* arg2 = *msg_data* arg3 = *topic*
MQPublish(arg1, arg2, arg3, arg4)	arg1 = *publisher_name* arg2 = *policy_name* arg3 = *msg_data* arg4 = *topic*

Table 8-3. MQPublish() argument interpretation (continued)

Usage	Argument Interpretation
MQPublish(arg1, arg2, arg3, arg4, arg5)	arg1 = *publisher_name* arg2 = *policy_name* arg3 = *msg_data* arg4 = *topic* arg5 = *correl_id*

Return Codes

1 The operation was successful.

Error The operation was unsuccessful.

Examples

Example 1:

```
begin;
EXECUTE FUNCTION MQPublish('Testing 123');
commit:
```

This example publishes the message with the following parameters:

- *publisher_name*: default publisher
- *policy_name*: default policy
- *msg_data*: "Testing 123"
- *topic*: None
- *correl_id*: None

Example 2:

```
begin;
EXECUTE FUNCTION MQPublish('MYPUBLISHER','Testing 345','TESTTOPIC');
commit;
```

This example publishes the message with the following parameters:

- *publisher_name*: "MYPUBLISHER"
- *policy_name*: default policy
- *msg_data*: "Testing 345"
- *topic*: "TESTTOPIC"
- *correl_id*: None

Example 3:

```
begin;
EXECUTE FUNCTION MQPublish('MYPUBLISHER','MYPOLICY','Testing 678','TESTTOPIC',
'TEST1');
commit;
```

This example publishes the message with the following parameters:

- *publisher_name*: "MYPUBLISHER"
- *policy_name*: "MYPOLICY"
- *msg_data*: "Testing 678"
- *topic*: "TESTTOPIC"
- *correl_id*: "TEST1"

Example 4:

```
begin;
EXECUTE FUNCTION MQPublish('Testing 901','TESTS');
commit;
```

This example publishes the message with the following parameters:

- *publisher_name*: default publisher
- *policy_name*: default policy
- *msg_data*: "Testing 901"
- *topic*: "TESTS"
- *correl_id*: None

Example 5:

```
begin;
EXECUTE FUNCTION MQPublish('SEND.MESSAGE', 'emergency', 'CODE BLUE', 'expedite');
commit;
```

This example publishes the message with the following parameters:

- *publisher_name*: "SEND.MESSAGE"
- *policy_name*: "emergency"
- *msg_data*: "CODE BLUE"
- *topic*: "expedite"
- *correl_id*: None

Example 6: The following table contains sample rows and columns in the "**informix**".**mqipubsub** table.

	pubsubname column	**receiver column**	**pubsubtype column**
Sample row 1	'IDS.DEFAULT. PUBLISHER'	' '	'Publisher'
Sample row 2	'IDS.DEFAULT. SUBSCRIBER'	'IDS.DEFAULT. SUBSCRIBER.RECEIVER'	'Subscriber'

```
begin;
 EXECUTE FUNCTION
    MQSubscribe('IDS.DEFAULT.SUBSCRIBER',
                'IDS.DEFAULT.PUB.SUB.POLICY', 'Weather');
commit;
```

This statement demonstrates a subscriber registering an interest in messages containing the topic "Weather," with the following parameters:

- *subscriber_name*: "IDS.DEFAULT.SUBSCRIBER"
- *policy_name*: "IDS.DEFAULT.PUB.SUB.POLICY"
- *topic*: "Weather"

```
begin;
   EXECUTE FUNCTION MQPublish('IDS.DEFAULT.PUBLISHER',
                              'IDS.DEFAULT.PUB.SUB.POLICY', 'Rain', 'Weather');
commit;
```

This statement publishes the message with the following parameters:

- *publisher_name*: "IDS.DEFAULT.PUBLISHER"
- *policy_name*: "IDS.DEFAULT.PUB.SUB.POLICY"

correl_id

Optional parameter. A string containing a correlation identifier to be associated with this message. The *correl_id* is often specified in request and reply scenarios to associate requests with replies. The maximum size of *correl_id* is 24 bytes. If not specified, no correlation ID is added to the message.

Usage

The **MQSendClob()** function puts the data contained in *clob_data* to the WMQ queue specified by *send_service*, using the quality of service policy defined by *policy_name*. If *correl_id* is specified, then the message is sent with a correlation identifier. If *correl_id* is not specified, then no correlation ID is sent with the message.

The following table describes how the arguments for the **MQSendClob()** function are interpreted.

Table 8-9. MQSendClob() argument interpretation

Usage	Argument Interpretation
MQSendClob(arg1)	arg1 = *clob_data*
MQSendClob(arg1, arg2)	arg1 = *service_name* arg2 = *clob_data*
MQSendClob(arg1, arg2, arg3)	arg1 = *service_name* arg2 = *policy_name* arg3 = *clob_data*
MQSendClob(arg1, arg2, arg3, arg4)	arg1 = *service_name* arg2 = *policy_name* arg3 = *clob_data* arg4 = *correl_id*

Return Codes

1	The operation was successful.
0 or Error	The operation was unsuccessful.

Examples

Example 1:

```
begin;
EXECUTE FUNCTION MQSendClob(filetoclob("/work/mydata", "client"));
commit;
```

This example sends a CLOB to the WMQ with the following parameters:

- *service_name*: default service name
- *policy_name*: default policy
- *msg_data*: filetoclob(″/work/mydata″, ″client″)
- *correl_id*: none

Example 2:

```
begin;
EXECUTE FUNCTION MQSendClob('MYSERVICE', filetoclob("/work/mydata", "client"));
commit;
```

This example sends a CLOB to the WMQ with the following parameters:

- *service_name*: "MYSERVICE"
- *policy_name*: default policy
- *msg_data*: filetoclob("/work/mydata", "client")
- *correl_id*: none

Example 3:

```
begin;
EXECUTE FUNCTION MQSendClob('MYSERVICE', 'MYPOLICY',
filetoclob("/work/mydata", "client"));
commit;
```

This example sends a CLOB to the WMQ with the following parameters:

- *service_name*: "MYSERVICE"
- *policy_name*: "MYPOLICY"
- *msg_data*: filetoclob("/work/mydata", "client")
- *correl_id*: none

Example 4:

```
begin;
EXECUTE FUNCTION MQSendClob('MYSERVICE', 'MYPOLICY',
filetoclob("/work/mydata", "client"), 'TEST3');
commit;
```

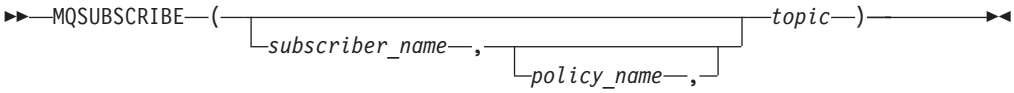
This example sends a CLOB to the WMQ with the following parameters:

- *service_name*: "MYSERVICE"
- *policy_name*: "MYPOLICY"
- *msg_data*: filetoclob("/work/mydata", "client")
- *correl_id*: "TEST3"

MQSubscribe()

The **MQSubscribe()** function is used to register interest in WMQ messages published on one or more topics.

Syntax



subscriber_name
Optional parameter. Refers to the value in the **pubsubname** column of the "**informix**".**mqiservice** table. If *subscriber_name* is not specified, IDS.DEFAULT.SUBSCRIBER is used as the subscriber. The maximum size of *subscriber_name* is 48 bytes.

policy_name
Optional parameter. Refers to the value in the **policyname** column of the "**informix**".**mqipolicy** table. If *policy_name* is not specified, IDS.DEFAULT.PUB.SUB.POLICY is used as the policy. The maximum size of *policy_name* is 48 bytes.

topic Required parameter. A string containing the topic for the message

publication. The maximum size of a topic is 40 bytes. Multiple topics can be specified in one string (up to 40 characters long). Each topic must be separated by a colon. For example, ″t1:t2:the third topic″ indicates that the message is associated with all three topics: t1, t2, and the third topic. If no topic is specified, none are associated with the message.

Usage

The **MQSubscribe()** function is used to register interest in WMQ messages published on a specified topic. The *subscriber_name* specifies a logical destination for messages that match the specified topic. Messages published on the topic are placed on the queue referred by the service pointed to by the **receiver** column for the subscriber (*subscriber_name* parameter). These messages can be read or received through subsequent calls to the **MQRead()** and **MQReceive()** functions on the receiver service.,

This function requires the installation of the WMQ Publish/Subscribe Component of WMQ and that the Message Broker must be running.

The following table describes how the arguments for the **MQSubscribe()** function are interpreted.

Table 8-10. MQSubscribe() argument interpretation

Usage	Argument Interpretation
MQSubscribe(arg1)	arg1 = *topic*
MQSubscribe(arg1, arg2)	arg1 = *service_name* arg2 = *topic*
MQSubscribe(arg1, arg2, arg3)	arg1 = *service_name* arg2 = *policy_name* arg3 = *topic*

Return Codes

1	The operation was successful.
Error	The operation was unsuccessful.

Examples

Example 1: The following table contains sample rows and columns in the ″**informix**″**.mqipubsub** table.

	pubsubname column	**receiver column**	**pubsubtype column**
Sample row 1	'IDS.DEFAULT. PUBLISHER'	''	'Publisher'
Sample row 2	'IDS.DEFAULT. SUBSCRIBER'	'IDS.DEFAULT. SUBSCRIBER.RECEIVER'	'Subscriber'

```
begin;
EXECUTE FUNCTION MQSubscribe('IDS.DEFAULT.SUBSCRIBER',
   'IDS.DEFAULT.PUB.SUB.POLICY', 'Weather');
commit;
```

The above statement demonstrates a subscriber registering an interest in messages containing the topic ″Weather″ with the following parameters:

- *subscriber_name*: "IDS.DEFAULT.SUBSCRIBER"
- *policy_name*: "IDS.DEFAULT.PUB.SUB.POLICY"
- *topic*: "Weather"

```
begin;
EXECUTE FUNCTION MQPublish('IDS.DEFAULT.PUBLISHER',
'IDS.DEFAULT.PUB.SUB.POLICY', 'Rain', 'Weather');
commit;
```

The above statement publishes the message with the following parameters:

- *publisher_name*: "IDS.DEFAULT.PUBLISHER"
- *policy_name*: "IDS.DEFAULT.PUB.SUB.POLICY"
- *msg_data*: "Rain"
- *topic*: "Weather"
- *correl_id*: none

```
begin;
EXECUTE FUNCTION MQReceive('IDS.DEFAULT.SUBSCRIBER.RECEIVER',
'IDS.DEFAULT.PUB.SUB.POLICY');
commit;
```

The above statement receives the message with the following parameters (it returns "Rain"):

- *service_name*: "IDS.DEFAULT.SUBSCRIBER.RECEIVER"
- *policy_name*: "IDS.DEFAULT.PUB.SUB.POLICY"

Example 2:

```
begin;
EXECUTE FUNCTION MQSubscribe('Weather');
commit;
```

This example demonstrates a subscriber registering an interest in messages containing the topic "Weather" with the following parameters:

- *subscriber_name*: default subscriber
- *policy_name*: default policy
- *topic*: "Weather"

Example 3:

```
begin;
EXECUTE FUNCTION MQSubscribe('PORTFOLIO-UPDATES', 'BASIC-POLICY', 'Stocks:Bonds');
commit;
```

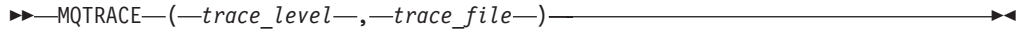
This example demonstrates a subscriber registering an interest in messages containing the topics "Stocks" and "Bonds" with the following parameters:

- *subscriber_name*: "PORTFOLIO-UPDATES"
- *policy_name*: "BASIC-POLICY"
- *topic*: "Stocks", "Bonds"

MQTrace()

The **MQTrace()** procedure specifies the level of tracing and the location to which the trace file is written.

Syntax



trace_level
Required parameter. Integer value specifying the trace level, currently only a value of greater than 50 results in output.

trace_file
Required parameter. The full path and name of the file to which trace information is appended. The file must be writable by user **informix**.

To enable tracing, you must first create a trace class by inserting a record into the **systemtraceclasses** system catalog:

```
insert into informix.systraceclasses(name) values ('idsmq')
```

For more details regarding tracing, see the *IBM Informix: Guide to SQL Reference*.

Examples

Example 1:

Enable tracing at a level of 50 with an output file of **/tmp/trace.log**:

```
EXECUTE PROCEDURE MQTrace(50, '/tmp/trace.log');
```

Execute a request:

```
begin;
EXECUTE FUNCTION MQSend('IDS');
commit;
```

Look at the trace output:

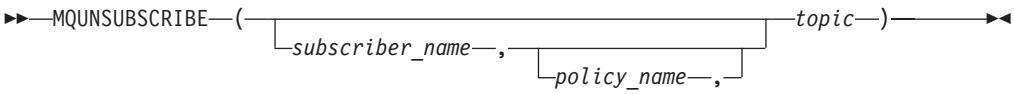
```
14:19:38 Trace ON level : 50
14:19:47  >>ENTER : mqSend<<
14:19:47    status:corrid is null
14:19:47  >>ENTER : MqOpen<<
14:19:47   status:MqOpen @ build_get_mq_cache()
14:19:47  >>ENTER : build_get_mq_cache<<
14:19:47   status:build_get_mq_cache @ mi_get_database_info()
14:19:47   status:build_get_mq_cache @ build_mq_service_cache()
14:19:47  >>ENTER : build_mq_service_cache<<
14:19:47  <<EXIT : build_mq_service_cache>>
14:19:47   status:build_get_mq_cache @ build_mq_policy_cache()
14:19:47  >>ENTER : build_mq_policy_cache<<
14:19:47  <<EXIT : build_mq_policy_cache>>
14:19:47   status:build_get_mq_cache @ build_mq_pubsub_cache()
14:19:47  >>ENTER : build_mq_pubsub_cache<<
14:19:47  <<EXIT : build_mq_pubsub_cache>>
14:19:47  <<EXIT : build_get_mq_cache>>
14:19:47   status:MqOpen @ MqiGetServicePolicy()
14:19:47  >>ENTER : MqiGetServicePolicy<<
14:19:47  <<EXIT : MqiGetServicePolicy>>
14:19:47    MQI:MqOpen @ MQCONNX()
14:19:47   status:MqOpen @ MqXadsRegister()
14:19:47  >>ENTER : MqXadsRegister<<
14:19:47   status:MqXadsRegister @ ax_reg()
14:19:47  <<EXIT : MqXadsRegister>>
14:19:47   status:MqOpen @ MqGetMqiContext()
14:19:47  >>ENTER : MqGetMqiContext<<
14:19:47    MQI:MqGetMqiContext @ MQOPEN()
14:19:47  <<EXIT : MqGetMqiContext>>
14:19:47  <<EXIT : MqOpen>>
```

```
14:19:47  >>ENTER : MqTransmit<<
14:19:47  >>ENTER : MqBuildMQPMO<<
14:19:47  <<EXIT : MqBuildMQPMO>>
14:19:47  >>ENTER : MqBuildMQMDSend<<
14:19:47  <<EXIT : MqBuildMQMDSend>>
14:19:47    MQI:MqTransmit @ MQPUT()
14:19:47  <<EXIT : MqTransmit>>
14:19:47  <<EXIT : mqSend>>
14:19:47  >>ENTER : MqEndTran<<
14:19:47    MQI:MqEndTran @ MQCMIT()
14:19:47   status:MqEndTran @ MqShut()
14:19:47  >>ENTER : MqShut<<
14:19:47   status:MqEndTran @ MQDISC
14:19:47  <<EXIT : MqEndTran>>:
```

MQUnsubscribe()

The **MQUnsubscribe()** function is used to unregister interest in WMQ messages published on one or more topics.

Syntax



subscriber_name
Optional parameter. Refers to the value in the **pubsubname** column of the "**informix**".**mqiservice** table. If *subscriber_name* is not specified, IDS.DEFAULT.SUBSCRIBER is used as the subscriber. The maximum size of *subscriber_name* is 48 bytes.

policy_name
Optional parameter. Refers to the value in the **policyname** column of the "**informix**".**mqipolicy** table. If *policy_name* is not specified, IDS.DEFAULT.PUB.SUB.POLICY is used as the policy. The maximum size of *policy_name* is 48 bytes.

topic Required parameter. A string containing the topic for the message publication. The maximum size of a topic is 40 bytes. Multiple topics can be specified in one string (up to 40 characters long). Each topic must be separated by a colon. For example, "t1:t2:the third topic" indicates that the message is associated with all three topics: t1, t2, and the third topic. If no topic is specified, none are associated with the message.

Usage

The **MQUnsubscribe()** function is used to unregister interest in WMQ messages subscription on a specified topic. The *subscriber_name* specifies a logical destination for messages that match the specified topic.

This function requires the installation of the WMQ Publish/Subscribe Component of WMQ and that the Message Broker must be running.

The following table describes how the arguments for the **MQUnsubscribe()** function are interpreted.

Table 8-11. MQUnsubscribe() argument interpretation

Usage	Argument Interpretation
MQUnsubscribe(arg1)	arg1 = *topic*
MQUnsubscribe(arg1, arg2)	arg1 = *service_name* arg2 = *topic*
MQUnsubscribe(arg1, arg2, arg3)	arg1 = *service_name* arg2 = *policy_name* arg3 = *topic*

Return Codes

1	The operation was successful.
Error	The operation was unsuccessful.

Examples

Example 1:

```
begin;
EXECUTE FUNCTION MQUnsubscribe('Weather');
commit;
```

This example demonstrates unsubscribing an interest in messages containing the topic ″Weather″ with the following parameters:

- *subscriber_name*: default subscriber
- *policy_name*: default policy
- *topic*: ″Weather″

Example 2:

```
begin;
EXECUTE FUNCTION MQUnsubscribe('PORTFOLIO-UPDATES','BASIC-POLICY',
      'Stocks:Bonds');
commit;
```

This example demonstrates unsubscribing an interest in messages containing the topic ″Weather″ with the following parameters:

- *subscriber_name*: ″PORTFOLIO-UPDATES″
- *policy_name*: ″BASIC-POLICY″
- *topic*: ″Stocks″, ″Bonds″

MQVersion()

The **MQVersion()** function returns the version of the MQ DataBlade module.

Syntax

Example

Show the version:

```
EXECUTE FUNCTION MQVersion();
OutPut of the MQVersion() function: MQBLADE 2.0 on 29-MAR-2005
```

Chapter 9. MQ DataBlade Module Error Handling

In This Chapter 9-1
Error Codes 9-1

In This Chapter

This chapter provides information on MQ DataBlade error codes.

Error Codes

This section describes MQ DataBlade error codes.

SQL State	Description
MQ000	Memory allocation failure in %FUNC%.
MQPOL	MQOPEN Policy : %POLICY%
MQSES	MQOPEN Session : %SESSION%
MQRCV	Read %BYTES% from the queue.
MQNMS	No data read/received, queue empty.
MQSUB	Subscribing to %SUBSCRIBE%.
MQVNV	VTI Table definition parameter NAME:%NAME% VALUE:%VALUE%.
MQNPL	VTI No policy defined for table mapped to MQ. Must define table with policy attribute.
MQNSV	VTI No service defined for table mapped to MQ. Must define table with service attribute.
MQNAC	VTI No access defined for table mapped to MQ. Must define table with access attribute.
MQBAC	VTI Invalid Access specification FOUND:%VALUE%, possible values%VALONE% or %VALTWO%.
MQVCN	VTI Qualified : Column 'correlid' cannot be qualified with NULL.
MQVTB	Table missing required 'message' column. Message column is bound to the queue, it is mandatory.
MQVSP	VTI mapped Queue did not include the POLICY and SESSION columns.
MQVIA	VTI table definition invalid access type (%VALUE%), valid access types are %READ% or %RECEIVE%.
MQVMS	VTI mapped queue missing SERVICE specification.
MQVMA	VTI mapped QUEUE creation did not include ACCESS definition.
MQVMP	VTI mapped QUEUE creation did not include POLICY specification.
MQVQC	VTI queue mapping, Column '%COLUMN%' must be qualified with a constant.
MQVQN	VTI queue mapping, Column '%COLUMN%' cannot be qualified with NULL.
MQVQE	VTI queue mapping, Column '%COLUMN%' can only use equality operator.
MQVQF	VTI queue mapping, column '%COLUMN%' - failed to fetch field.
MQSUN	Invalid selector '%IDX%' found, path not possible.
MQERX	Extended error : '%FUNC%', code:%CODE% explain: %EXPLAIN%, refer to MQSeries manual for further description.
MQGEN	%FUNC% encountered error %ERR% with accompanying message : %MSG%
MQTNL	Topic cannot be NULL.
MQCNL	Internal error encountered NULL context.
MQNLM	Cannot send NULL message.

© Copyright IBM Corp. 2005

SQL State	Description
MQVNQ	MQSeries underlying qualification system does not support negation.
MQVDQ	Qualifications cannot bridge between MQSeries and database.
MQEDN	MQ Transport error, service '%NAME%' underlying queue manager may not be activated.
MQEPL	Policy '%POLICY%' could not be found in the repository.
MQRLN	Error during read, expected %EXPECT%, received:%READ%.
MQELO	Error attempting to fetch CLOB, function:%NAME% returned %CODE%.
MQRDA	MQ Transport error, service '%NAME%' underlying transpost layer not enabled to receive requests
MQSDA	MQ Transport error, service '%NAME%' underlying transpost layer not enabled to send requests
MQVQM	MQSeries : Cannot have multiple qualifies for the same column (%COLUMN%).
MQRFQ	Retrieved entries from queue, at least one entry failed qualification - data lost.
MQQCI	Qualification column invalid, only can qualify on 'topic' and 'correlid'.
MQGER	MQ Error : %MSG%
MQGVT	MQ VTI Error : %MSG%
MQZCO	Correlation value found to be zero length, invalid value for MQSeries.
MQVTN	Must supply name of VTI table.
MQ018	FUNCTION:%NAME%, SERVICE:%SERVICE%, POLICY:%POLICY% :: The specified (sender, receiver, distribution list, publisher, or subscriber) service was not found, so the request was not carried out.
MQ020	FUNCTION:%NAME%, SERVICE:%SERVICE%, POLICY:%POLICY% :: The specified policy was not found, so the request was not carried out.
MQT40	Topic exceeded forty character maximum.
MQINX	Input too large, maximum:%len% found:%txt%
MQITM	Invalid table 'msg' column size %len%, valid range (1-%max%)
MQEXT	AMRC_TRANPORT_ERR, fetched secondary error at:%NAME%, MQI error :%ERR%
MQXAR	Xadatasource (%XADS%) registration error : FUNCTION: %FUNCTION%, RETURN VALUE: %VALUE%
MQ010	FUNCTION:%NAME%: Unable to obtain database information.
MQ011	FUNCTION:%NAME%: Error while querying table:%TABNAME%
MQ012	FUNCTION:%NAME%: Unexpected NULL value while querying the table:%TABNAME%
MQ013	FUNCTION:%NAME%: Unexpected return value from mi function while querying table:%TABNAME%
MQ014	FUNCTION:%NAME%: Unexpected failure opening mi connection while querying table:%TABNAME%
MQMQI	FUNCTION:%FNAME%, SERVICE:%SERVICE%, POLICY:%POLICY% :: MQI Error generated by %MQINAME% with CompCode=%CCODE%, Reason=%REASON%.
MQ015	FUNCTION:%FNAME%, SERVICE:%SERVICE%, POLICY:%POLICY% :: %NAME% is not present in the database %TABNAME% table.
MQ016	FUNCTION:%FNAME%, SERVICE:%SERVICE%, POLICY:%POLICY% :: Connection to Multiple QueueManager's are not allowed in the same transaction.
MQ019	FUNCTION:%FNAME%, SERVICE:%SERVICE%, POLICY:%POLICY% :: Internal Error. not able to switch to the virtual processor where the MQCONNX() is invoked.
MQ017	FUNCTION:%FNAME%, SERVICE:%SERVICE%, POLICY:%POLICY% :: Internal Error. The Virtual processor class not the same as ""MQ""

Part 3. Appendixes

© Copyright IBM Corp. 2005

Appendix. Accessibility

The syntax diagrams in the HTML version of this manual are available in dotted decimal syntax format, which is an accessible format that is available only if you are using a screen reader.

Dotted Decimal Syntax Diagrams

In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), the elements can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that your screen reader is set to read punctuation. All syntax elements that have the same dotted decimal number (for example, all syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, the word or symbol is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is read as 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol that provides information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, this identifies a reference that is defined elsewhere. The string following the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you should refer to a separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

? Specifies an optional syntax element. A dotted decimal number followed

© Copyright IBM Corp. 2005

by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element (for example, 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.

! Specifies a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP only applies to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

***** Specifies a syntax element that can be repeated zero or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data-area, you know that you can include more than one data area or you can include none. If you hear the lines 3*, 3 HOST, and 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you could write HOST STATE, but you could not write HOST HOST.
3. The * symbol is equivalent to a loop-back line in a railroad syntax diagram.

+ Specifies a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times. For example, if you hear the line 6.1+ data-area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. As for the * symbol, you can only repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Notices

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created

© Copyright IBM Corp. 2005

programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

> © (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. (enter the year or years). All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

AIX; DB2; DB2 Universal Database; Distributed Relational Database Architecture; NUMA-Q; OS/2, OS/390, and OS/400; IBM Informix®; C-ISAM®; Foundation.2000™; IBM Informix ® 4GL; IBM Informix®DataBlade®Module; Client SDK™; Cloudscape™; Cloudsync™; IBM Informix®Connect; IBM Informix®Driver for JDBC; Dynamic Connect™; IBM Informix®Dynamic Scalable Architecture™(DSA); IBM Informix®Dynamic Server™; IBM Informix®Enterprise Gateway Manager (Enterprise Gateway Manager); IBM Informix®Extended Parallel Server™; i.Financial Services™; J/Foundation™; MaxConnect™; Object Translator™; Red Brick™; IBM Informix® SE; IBM Informix® SQL; InformiXML™; RedBack®; SystemBuilder™; U2™; UniData®; UniVerse®; wintegrate®are trademarks or registered trademarks of International Business Machines Corporation.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Windows, Windows NT, and Excel are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

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

Other company, product, and service names used in this publication may be trademarks or service marks of others.

Index

Special characters

"informix".mqipolicy table 6-4
"informix".mqipubsub table 6-4
"informix".mqiservice table 6-4

A

Accessibility xvi
  dotted decimal format of syntax diagrams A-1
  syntax diagrams, reading in a screen reader A-1
API interface 3-2
  using 4-7, 4-10
Attribute
  flags
    client file settings 3-22
    large object settings 3-10

B

Binary data
  determining for lld_lob data type 3-31, 4-3
  inserting into table 4-1
  specifying with lld_lob data type 2-2
BladeManager
  registering Large Object Locator 1-3
  registering MQ 6-9
BLOB data type
  casting to lld_lob data type 2-3
    explicitly 4-2
    implicitly 4-1
Boldface type ix

C

Callback function
  registering 5-1
Casting
  BLOB data type to lld_lob data type 2-3
    explicitly 4-2
    implicitly 4-1
  CLOB data type to lld_lob data type 2-3
    explicitly 4-2
    implicitly 4-1
  lld_lob data type to BLOB and CLOB data types 2-3, 4-1, 4-2
Character data
  determining for lld_lob data type 3-31, 4-3
  inserting into table 4-1
  specifying with lld_lob data type 2-2
Client files
  attribute flags 3-22
  copying
    to a large object 3-20, 3-21, 3-24
    to a large object, example 4-1, 4-5, 4-6
  creating 3-18
  deleting 3-19
  functions 3-17, 3-24
  opening 3-22, 3-23
CLOB data type
  casting to lld_lob data type 2-3
    explicitly 4-2
    implicitly 4-1
Code, sample, conventions for xii
Command-line conventions
  how to read xi
  sample diagram xi
Compliance
  with industry standards xviii
Concurrent access, how to limit 1-3
Contact information xviii
Conventions
  command-line xi
  documentation viii
  functions, naming 3-1
  sample-code xii
  syntax diagrams ix
  syntax notation ix
  typographical viii

D

Data type
  lld_lob
    casting to BLOB and CLOB data types 2-3, 4-1, 4-2
    defined 2-2
    determining type of data 3-31, 4-3
    introduced 1-2
    using 4-1, 4-3
    using to insert binary and character data into table 4-1
  lld_locator
    defined 2-1
    introduced 1-2
    using 4-4, 4-7
    using to insert row into table 4-4
    using to reference smart large object, example 4-4
Default table values
  MQ DataBlade 6-8
Disabilities, visual
  reading syntax diagrams A-1
Documentation conventions viii
Documentation Notes xiv
Documentation set of all manuals xvi
Documentation, types of xiii
  machine notes xiv
  online manuals xv
  printed manuals xvi
Dotted decimal format of syntax diagrams A-1
Dynamic Server
  configuring for MQ Datablade 6-3

E

Environment variables ix
Error code
  argument 5-1
Error codes
  MQ DataBlade 9-1
Error messages xv

© Copyright IBM Corp. 2005

Errors
callback functions, registering for 5-1
codes listed 5-2
codes listed, MQ DataBlade 9-1
error code argument for 5-1
exceptions, generating for 3-26
exceptions, handling for 5-1
handling 5-1
example of 4-10
functions for 3-25, 3-27
MQ DataBlade 9-1
SQL 5-1
status of, and function return value 5-1
translating to SQL states 3-27
ESQL/C
interface 3-2
Exceptions
generating 3-26
handling 5-1

F

File
client.
See Client files.
copying smart large objects to 3-30
creating, example 4-6
deleting, example 4-6
Fixed and Known Defects File xiv
Function
basic large object 3-2, 3-16
client file support 3-17, 3-24
error code argument 5-1
error utility 3-25, 3-27
introduced 1-2
lld_close() 3-4
using 4-7, 4-10
lld_copy() 3-5, 3-6
using 4-5, 4-6
lld_create 3-7, 3-8, 4-4
lld_create_client() 3-18
lld_delete_client() 3-19
lld_delete() 3-9
lld_error_raise() 3-26
lld_from_client() 3-20, 3-21
using 4-5
LLD_LobType 3-31, 4-3
lld_open_client 3-22, 3-23
lld_open() 3-10, 3-11
using 4-7, 4-10
lld_read() 3-12, 4-7, 4-10
lld_sqlstate 3-27
lld_tell() 3-15
lld_to_client() 3-24, 4-6
lld_write() 3-16, 4-7, 4-10
LOCopy 3-29
LOToFile 3-30
MQCreateVtiRead() 8-2
MQCreateVtiReceive() 8-4
MQPublish() 8-6
MQRead() 8-10
MQReadClob() 8-12
MQReceive() 8-14
MQReceiveClob() 8-17
MQSend() 8-19
MQSendClob() 8-21
MQSubscribe() 8-23
Function *(continued)*
MQTrace() 8-25
MQUnsubscribe() 8-27
MQVersion() 8-28
naming conventions 3-1
return value and error status 5-1
smart large object copy 3-28, 3-29

H

Help xvi

I

Industry standards, compliance with xviii
Informix Dynamic Server documentation set xvi
Installation Guides xiii
Installing
Large Object Locator 1-3
MQ DataBlade 6-9
Interfaces 3-1
API 3-2
using 4-7, 4-10
ESQL/C 3-2
naming conventions 3-1
SQL 3-2
using 4-1, 4-7

K

Keywords
in syntax diagrams xii

L

Large Object Locator 1-1
functions
See Function or individual function name.
installing 1-3
registering 1-3
types of users viii
Large objects
accessing 1-1
appending data to 3-16
attribute flags of 3-10
basic functions for 3-2, 3-16
closing 3-4
copying
client files to 3-20, 3-21
function for 3-5, 3-6
to client files 3-24
to large objects, example 4-5
copying to client files, example 4-6
creating 3-7, 3-8
defined 1-1
deleting 3-9
limiting concurrent access 1-3
offset
returning for 3-15
setting in 3-13
opening 3-10, 3-11
protocols, listed 2-1
reading from 3-12
referencing 2-1
setting read and write position in 3-13, 3-14

Large objects *(continued)*
tracking open 5-2
writing to 3-16
Libraries
API 3-2
ESQL/C 3-2
SQL 3-2
Limitations
MQ DataBlade 6-2
lld_close() function 3-4
using 4-7, 4-10
lld_copy() function 3-5, 3-6
using 4-5, 4-6
lld_create_client() function 3-18
lld_create() function 3-7, 3-8
using 4-4
lld_delete_client() function 3-19
lld_delete() function 3-9
lld_error_raise() function 3-26
lld_from_client() function 3-20, 3-21
using 4-5
LLD_IO structure 3-11, 3-23
lld_lob data type
casting to BLOB and CLOB data types 2-3, 4-1
explicitly 4-2
defined 2-2
determining type of data in 3-31, 4-3
inserting binary data into table 4-1
inserting character data into table 4-1
introduced 1-2
using 4-1, 4-3
LLD_LobType function 3-31
using 4-3
lld_locator data type
defined 2-1
inserting a row into a table 4-4
introduced 1-2
referencing a smart large object 4-4
using 4-4, 4-7
lld_open_client() function 3-22, 3-23
attribute flags 3-22
lld_open() function 3-10, 3-11
attribute flags 3-10
flags 3-10
using 4-7, 4-10
lld_read() function 3-12
using 4-7, 4-10
lld_sqlstate() function 3-27
lld_tell() function 3-15
lld_to_client() function 3-24
using 4-6
lld_write() function 3-16
using 4-7, 4-10
LOCopy function 3-29
LOToFile function 3-30

M

Machine notes xiv
Messages
receiving from a queue 6-2
sending to a queue 6-2
Messaging
WMQ 6-1
MQ DataBlade
configuring Dynamic Server for 6-3
default table values 6-8
MQ DataBlade *(continued)*
error codes 9-1
error codes listed 9-1
error handling 9-1
functions 6-1
binding a table 7-2
creating a table 7-2
MQCreateVtiRead() 8-2
MQCreateVtiReceive() 8-4
MQPublish() 8-6
MQRead() 8-10
MQReadClob() 8-12
MQReceive() 8-14
MQReceiveClob() 8-17
MQSend() 8-19
MQSendClob() 8-21
MQSubscribe() 8-23
MQTrace() 8-25
MQUnsubscribe() 8-27
MQVersion() 8-28
overview 8-2
retrieving a queue element 7-3
inserting data into queue 6-9
installing 6-9
installing WMQ 6-2
limitations 6-2
preparing 6-2
publishing to queue 6-10, 6-11
reading entry from queue 6-10
receiving entry from queue 6-10
registering 6-9
Registration 6-3
requirements 6-2
subscribing to queue 6-10
tables 6-1
unsubscribing from queue 6-11
verifying functionality 6-9
mq virtual processor class, creating 6-3
MQCreateVtiRead() function
defined 8-2
MQCreateVtiReceive() function
defined 8-4
MQPublish() function
defined 8-6
MQRead() function
defined 8-10
MQReadClob() function
defined 8-12
MQReceive() function
defined 8-14
MQReceiveClob() function
defined 8-17
MQSend() function
defined 8-19
MQSendClob() function
defined 8-21
MQSubscribe() function
defined 8-23
MQTrace() function
defined 8-25
MQUnsubscribe() function
defined 8-27
MQVersion() function
defined 8-28

N

Naming conventions 3-1

O

Offset
  in large objects
    returning 3-15
    setting 3-13
Online help xvi
Online manuals xv
Online notes xiii, xiv

P

Parameters, storage
  specifying for smart large objects 3-8, 3-21
Printed manuals xvi
Protocol
  list, for large objects 2-1

R

Registering
  Large Object Locator 1-3
  MQ DataBlade 6-9
Release Notes xiv
Requirements
  MQ DataBlade 6-2
Resources
  cleaning up 1-3
  reclaiming client file 3-18
Rollback
  limits on with Large Object Locator 1-3

S

Sample-code conventions xii
sbspace storage parameter, specifying for smart large objects 3-8, 3-21
Schema mapping to WMQ objects 7-1
Screen reader
  reading syntax diagrams A-1
Smart large object
  copying client files to 3-21
  copying to 3-6
  copying to a file 3-30
  copying to a smart large object 3-29
  creating 3-7
  creating with lld_copy() function 3-6
  creating, example 4-4
  deleting 3-9
  functions for copying 3-28, 3-29
  referencing with lld_lob data type 2-2
  referencing, example 4-1
  storage parameter defaults 3-8, 3-21
SQL
  errors 5-1
  interface 3-2
    using 4-1, 4-7
  states, translating from error codes 3-27
SQL code xii
Storage parameters
  specifying for smart large objects 3-8, 3-21
Syntax diagrams
  conventions for ix
  keywords in xii
  reading in a screen reader A-1
  variables in xii
Syntax segment xi

T

Table values
  default
    MQ DataBlade 6-8
TOC Notes xiv
Transaction rollback
  creating client files and 3-18
  creating large objects and 3-7
  limits on with Large Object Locator 1-3
Types.
  See Data type.
Typographical conventions viii

U

User informix, adding to mqm group 6-3
User-defined routine
  calling API functions from 3-2
  example 4-7, 4-11

V

Variables, in syntax diagrams xii
Virtual-Table Interface
  accessing WMQ queues 7-1
Visual disabilities
  reading syntax diagrams A-1
VPCLASS parameter
  MQ DataBlade 6-3
VTI
  accessing WMQ queues 7-1

W

WMQ
  messages
    SELECT 7-2
  messaging 6-1
  metadata table behavior 7-2
  objects
    schema mapping to 7-1
  product documentation 6-3
  queues
    accessing 7-1
    configuring 6-2
    INSERTitems into 7-2
    mapping to tables 7-1
  tables mapped to
    generating errors 7-4

Printed in USA

G251-2770-00