IBM Informix

Version 11.50

IBM

IBM Informix DataBlade API Programmer's Guide

IBM Informix

Version 11.50

IBM

IBM Informix DataBlade API Programmer's Guide

Contents

Part 2. Data Manipulation

Part 4. Creating User-Defined Routines

Part 5. Appendixes

Appendix A. Writing a Client LIBMI Application A-1

Appendix B. Accessibility B-1

Notices C-1

Index X-1

Introduction

In This Introduction

This introduction provides an overview of the information in this publication and describes the conventions it uses.

About This Publication

This publication contains information on the DataBlade API, the C-language application programming interface (API) provided with IBM Informix Dynamic Server (IDS). You can use the DataBlade API to develop client LIBMI applications and C user-defined routines (UDRs) that access data in an IBM Informix Dynamic Server (IDS) database.

This publication explains how to use the DataBlade API functions. The companion publication, the *IBM Informix DataBlade API Function Reference*, describes the functions in alphabetical order.

This section discusses the intended audience, the software that you need to use the DataBlade API, localization, and demonstration databases.

Types of Users

This publication is for the following users:

- Database-application programmers
- DataBlade® developers
- Developers of C UDRs

To understand this publication, you need to have the following background:

- A working knowledge of your computer, your operating system, and the utilities that your operating system provides
- Some experience working with relational databases or exposure to database concepts
- Some experience with computer programming in the C programming language
- Some experience with database design and the optimization of database queries

If you have limited experience with relational databases, SQL, or your operating system, see the *IBM Informix Dynamic Server Getting Started Guide* for your database server for a list of supplementary titles.

Software Dependencies

This publication is based on the assumption that you are using Version 11.50 of IBM Informix Dynamic Server (IDS).

Assumptions About Your Locale

IBM Informix products can support many languages, cultures, and code sets. All the information related to character set, collation, and representation of numeric data, currency, date, and time is brought together in a single environment, called a Global Language Support (GLS) locale.

The examples in this publication are for the default locale, **en_us.8859-1**. This locale supports U.S. English format conventions for date, time, and currency. In addition, this locale supports the ISO 8859-1 code set, which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

If you plan to use nondefault characters in your data or your SQL identifiers, or if you want to conform to the nondefault collation rules of character data, you need to specify the appropriate nondefault locale.

For instructions on how to specify a nondefault locale, additional syntax, and other considerations related to GLS locales, see the *IBM Informix GLS User's Guide*.

Demonstration Databases

The DB–Access utility, which is provided with your Informix® database server products, includes one or more of the following demonstration databases:

- The **stores_demo** database illustrates a relational schema with information about a fictitious wholesale sporting-goods distributor. Many examples in IBM Informix publications are based on the **stores_demo** database.
- The **superstores_demo** database illustrates an object-relational schema. The **superstores_demo** database contains examples of extended data types, type and table inheritance, and user-defined routines.

For information about how to create and populate the demonstration databases, see the *IBM Informix DB–Access User's Guide*. For descriptions of the databases and their contents, see the *IBM Informix Guide to SQL: Reference*.

The scripts that you use to install the demonstration databases reside in the **$INFORMIXDIR/bin** directory on UNIX® or Linux and in the **%INFORMIXDIR%\bin** directory on Windows.

Function Syntax Conventions

This guide uses the following conventions to specify DataBlade API function syntax:

- Brackets ([ ]) surround optional items.
- Braces ({ }) surround items that can be repeated.
- A vertical line (|) separates alternatives.
- Function parameters are italicized; arguments that you must specify as shown are not italicized.

DataBlade API Module Code Conventions

This publication includes sample code for DataBlade API modules. These samples follow C-language coding conventions for indentation and use C ANSI format for parameters in function declarations.

Note: 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.

Documentation Conventions

This section describes the following conventions, which are used in the product documentation for IBM® Informix Dynamic Server:

- Typographical conventions
- Feature, product, and platform conventions
- Example code conventions

Typographical Conventions

This publication 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 names, path names, 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.

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 operating system

```
└───────────────────── 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 (Windows)

Example Code Conventions

Examples of SQL code occur throughout this publication. 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 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. If you are using DB–Access, you must delimit multiple statements with semicolons.

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 documentation for your product.

Additional Documentation

You can view, search, and print all of the product documentation from the IBM Informix Dynamic Server information center on the Web at http://publib.boulder.ibm.com/infocenter/idshelp/v115/index.jsp.

For additional documentation about IBM Informix Dynamic Server and related products, including release notes, machine notes, and documentation notes, go to the online product library page at http://www.ibm.com/software/data/informix/

pubs/library/. Alternatively, you can access or install the product documentation from the Quick Start CD that is shipped with the product.

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.

How to Provide Documentation Feedback

You are encouraged to send your comments about IBM Informix user documentation by using one of the following methods:

* Send e-mail to docinf@us.ibm.com.
* Go to the Information Center at http://publib.boulder.ibm.com/infocenter/ idshelp/v115/index.jsp and open the topic that you want to comment on. Click the feedback link at the bottom of the page, fill out the form, and submit your feedback.

Feedback from both methods is monitored by those who maintain the user documentation of Dynamic Server. The feedback methods are 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 Web site at http://www.ibm.com/planetwide/.

We appreciate your suggestions.

Part 1. DataBlade API Overview

Chapter 1. Using the DataBlade API

In This Chapter

The IBM Informix DataBlade API is the application programming interface (API) for IBM Informix Dynamic Server (IDS). You can use DataBlade API functions in DataBlade modules to access data stored in a Dynamic Server database.

This chapter provides the following information:

- A description of the different kinds of DataBlade API modules you can write with the DataBlade API
- A summary of the basic parts of the DataBlade API

For information about how to develop DataBlade modules, see the *IBM Informix DataBlade Developers Kit User's Guide*.

DataBlade API Module

A *DataBlade API module* is a C-language module that uses the functions of the DataBlade API to communicate with Dynamic Server. You can use the DataBlade API in either of the following kinds of DataBlade API modules:

- A C UDR: a user-defined routine that is written in C
- A client LIBMI application: a client application written in C

Tip: This publication uses the term "DataBlade API module" generically to refer to either a client LIBMI application or a user-defined routine (UDR).

To provide portability for applications, most of the DataBlade API functions behave identically in a UDR and a client LIBMI application. In cases where syntax or

semantics differ, this publication uses qualifying paragraphs to distinguish between server-side and client-side behavior of the DataBlade API.

If neither the server-specific or client-specific qualifying paragraphs appear, you can assume that the functionality is the same in both the server-side and client-side implementations of the DataBlade API. For more information, see "Feature, Product, and Platform Markup" on page xiii of the introduction.

You can dynamically determine the kind of DataBlade API module with the **mi_client( )** function.

User-Defined Routine (Server)

A *user-defined routine* (UDR) is a routine that you can invoke within an SQL statement or another UDR. UDRs are building blocks for the development of DataBlade modules. Possible uses for a UDR follow:

- Support function for an opaque data type
- Cast function to cast data from one data type to another
- End-user routine for use in SQL statements
- Operator function to implement an operation on a particular data type

For a more complete list, see "Uses of a C UDR" on page 12-3.

When you write a UDR in an external language (a language other than SPL), the UDR is called an *external routine*. An external routine that is written in the C language is called a *C UDR*. A C UDR uses the server-side implementation of the DataBlade API to communicate with the database server.

This section provides the following information about C UDRs. For general information about UDRs, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Types of UDRs

You can write the following types of C UDRs.

Type of UDR	Description	C Implementation

User-defined *function*	Returns one or more values and therefore can be used in SQL expressions
	
	For example, the following query returns the results of a UDR named **area( )** as part of the query results:
	
	`SELECT diameter, area(diameter)`
	`        FROM shapes`
	`        WHERE diameter > 6;`
		A C function that returns some data type other than **void** (usually a DataBlade API data type)

User-defined *procedure*	Does *not* return any values and cannot be used in SQL expressions because it does not return a value You can call a user-defined procedure directly, however, as the following example shows: `EXECUTE PROCEDURE myproc(1, 5);`	A C function that returns **void**

Differences between C UDRs and UDRs Written in SPL

Advantages of C UDRs over UDRs written in the SPL language:

- Performance, efficiency, and flexibility of C code

  C UDRs are compiled to machine code. You can use the C programming language to manipulate data at the level of bytes and bits and access data in efficient data structures such as array, hash, linked list, or tree.

- Access to the DataBlade API (DAPI) and other C libraries

  DAPI provides many functions that are not available in SPL or SQL, including the ESQL/C function library for manipulating data in C. Any C library that follows the guidelines of the DataBlade API can also be included. For example, a C UDR has random access to data within a smart large object.

- Greater dynamic SQL support in C routines

  C UDRs can dynamically build arbitrary SQL query strings at runtime and execute them. In SPL, the CLOSE, DECLARE, EXECUTE IMMEDIATE, FETCH, FREE, OPEN, and PREPARE statements support runtime replacement of question mark (?) placeholders with specific input parameter values, but some dynamic SQL syntax features and some cursor management statements of ESQL/C are not supported in SPL. For example, IDS 11.50 only supports sequential cursors. C UDRS can have other types of cursors such as scroll and hold. The FOREACH statement of SPL declares a direct cursor, but its associated SQL statement must have hard-coded names of database objects, such as tables, columns, and functions because SPL variables can only represent values, not SQL identifiers. (The EXEC Bladelet also supports some dynamic SQL features in SPL routines, but its programming interface is more complex and less intuitive than when SPL is used directly.)

- Extending the server

  You can use C UDRs to define user-defined data types (UDTs), user-defined aggregates, and user-defined access methods (for example, to access stream data outside IDS) to return data on the selectivity and cost of another UDR to the optimizer and to access data of a ROW type that was unknown at compile time.

Advantages of UDRs written in SPL over C UDRs:

- SPL routines typically require less coding SPL is a higher-level language than C and can therefore accomplish a given task in fewer lines of code

  For example, in SPL it takes only a few lines to execute SQL and fetch results in a loop. In C, it takes many lines to define and prepare the statement, execute it with a cursor, fetch rows, fetch columns, close the cursor, close the statement, and check for errors during the process.

- All SQL statements in SPL routines are automatically prepared

  In SPL, any embedded SQL statements are parsed, prepared, and optimized when the SPL routine is created and compiled. In a C UDR, if you want to execute SQL repeatedly and efficiently, you must prepare it explicitly. (SPL can only use the PREPARE statement to prepare a query or a call to a routine, but it can then use EXECUTE IMMEDIATE to execute the prepared statement.)

- SPL routines are easier to write

  A C UDR must follow the documented guidelines of the DataBlade API in areas that include yielding the processor, allocating memory and variables, performing I/O, and making system calls that block. Failure to follow the guidelines can cause problems for the IDS instance, although you can mitigate this risk by registering the C UDR to run on a user-defined VP class.

- Support for non-cursor EXECUTE...INTO statement

Beginning with IDS 11.50, SPL supports only an EXECUTE IMMEDIATE non-cursor statement that does not return any row. However, ESQL/C also supports the non-cursor EXECUTE ... INTO statement. The query in this statement can return a single row that is assigned to the SPL variables listed after the INTO clause. Although SPL in IDS 11.50, or later does not support multiple statements within the non-cursor EXECUTE IMMEDIATE statement, this restriction reduces the risk of the insertion of unwanted SQL statements.

Using UDRs

You can write a UDR in C by using the DataBlade API functions to communicate with the database server. You can also write subroutines in C that a UDR calls as it executes. These subroutines must follow the same rules as the UDR with respect to the use of DataBlade API functions.

Tip: Because of the subject matter of this publication, the publication uses the terms "C UDR" and "UDR" interchangeably.

You compile UDRs into shared-object files. You then register the UDR in the system catalog tables so that the database server can locate the code at runtime. The database server dynamically loads the shared-object files into memory when the UDR executes.

For more information on how to create C UDRs, see the following chapters of this publication:
* Chapter 12, "Developing a User-Defined Routine," on page 12-1, provides an overview to the development process, including information on compilation, registration, execution, and debugging.
* Chapter 13, "Writing a User-Defined Routine," on page 13-1, describes specific features and tasks of a C UDR.
* Chapter 14, "Managing Memory," on page 14-1, describes how to manage memory allocation within a C UDR.
* Chapter 15, "Creating Special-Purpose UDRs," on page 15-1, describes how to create special kinds of UDRs, such as iterator functions, user-defined aggregates, and optimization functions.

Client LIBMI Application

A *client LIBMI application* is a stand-alone client application that uses the client-side implementation of the DataBlade API to communicate with the database server. The application might be written in C, C++, or Visual Basic.

Important: Support is provided for client LIBMI applications for backward compatibility with existing applications. For the development of new C client applications, use another IBM Informix C-language product such as IBM Informix ODBC.

Compatibility of Client and Server DataBlade API Modules

You can execute a UDR from an SQL statement as well as from a client application with little or no modification to the code. Any function that does not require interactive input from the client application can be written as a UDR. However, not all application code should be in a C UDR. You must balance the load between the client and the database server to achieve optimal performance.

To avoid interfering with the operation of the database server, you can develop functions on the client side even if they are intended to run from the server

process eventually. When you develop a C UDR on a client computer, you can use the same DataBlade API functions on the client and the server computers, in most cases, without changing the code. Almost all of the DataBlade API functions behave identically in a client LIBMI application and a C UDR to provide portability for DataBlade API modules. If you are writing code that might execute in either a C UDR or a client LIBMI application, you can use the **mi_client( )** function to determine at runtime where the code is running.

DataBlade API Components

The DataBlade API contains the following components for the development of DataBlade API modules:
- Header files
- Public data type structures
- Public functions

Header Files

The following categories of header files are provided for use in a DataBlade API module:
- *DataBlade API header files* define DataBlade API data types and functions.
- IBM Informix ESQL/C *header files* define the IBM Informix ESQL/C library functions and data types.
- The IBM Informix GLS header file provides the ability to internationalize your DataBlade API module.
- Private header files, which you create, can support the DataBlade API module.

DataBlade API Header Files
The DataBlade API header files begin with the **mi** prefix. The DataBlade API provides the following public header files for use in DataBlade API modules.

Header File	Description
mi.h	Is the main DataBlade API header file
	It includes other DataBlade API public header files: **milib.h**, **milo.h**, and **mitrace.h**.
	The **mi.h** header file does not automatically include **mistrmtype.h**. To use the stream I/O functions of the DataBlade API, you must explicitly include **mistrmtype.h**.
milib.h	Defines function prototypes for the public entry points and public declarations of required data type structures and related macros
	The **mi.h** header file automatically includes **milib.h**.
mitypes.h	Defines all DataBlade API simple data types, accessor macros for these data types, and directly related value macros
	The **mitypes.h** header file automatically includes the Informix ESQL/C header files: **datetime.h**, **decimal.h**, and **int8.h**.
	The **milib.h** header file automatically includes **mitypes.h**.
milo.h	Defines the data type structures, values, and function prototypes for the smart-large-object interface (functions that have names starting with **mi_lo_**)
	The **mi.h** header file automatically includes **milo.h**.

Header File	Description
mistream.h	Contains definitions for stream data structures, error constants, and generic stream I/O functions
	The **mistrmtype.h** and **mistrmutil.h** header files automatically include **mistream.h**.
mistrmtype.h	Contains definitions for the type-specific stream-open functions that the DataBlade API provides
	The **mistrmtype.h** header file automatically includes **mistream.h**; however, the **mi.h** header file does not include **mistrmtype.h**. You must explicitly include **mistrmtype.h** to use the stream I/O functions of the DataBlade API.
mistrmutil.h	Contains definitions for the stream-conversion functions that the DataBlade API provides for use in **streamwrite( )** and **streamread( )** opaque-type support functions
	The **mistrmutil.h** header file automatically includes **mistream.h**; however, the **mi.h** header file does not include **mistrmutil.h**. You must explicitly include **mistrmutil.h** to use the stream-conversion functions of the DataBlade API.
mitrace.h	Defines the data type structures, values, and function prototypes for the DataBlade API trace facility
	The **mi.h** header file automatically includes **mitrace.h**.
miconv.h	Contains convention definitions, including on/off switches based on architecture, compiler type, and so on
	Other parts of the code use these switches to define data types correctly.
	The **mitypes.h** header file automatically includes **miconv.h**.
memdur.h	Contains the definition of the **MI_MEMORY_DURATION** data type, which enumerates valid public memory durations
	The **milib.h** header file automatically includes **memdur.h**.

The **mi.h** header file provides access to most of the DataBlade API header files in the preceding table. Include this header file in your DataBlade API module to obtain declarations of most DataBlade API functions and data types.

The DataBlade API provides the following advanced header files for the use of advanced features in C UDRs.

Header File	Description
minmmem.h	Includes the **minmdur.h** and **minmprot.h** header files, which are necessary for access to advanced memory durations and memory-management functions
	Neither the **mi.h** nor **milib.h** header file automatically includes **minmmem.h**. You must explicitly include **minmmem.h** to use advanced memory durations or memory-management functions.
minmdur.h	Contains definitions for the advanced memory durations
	The **minmmem.h** header file automatically includes **minmdur.h**. You must explicitly include **minmmem.h** to use advanced memory durations.

Header File	Description
minmprot.h	Contains definitions for the advanced DataBlade API functions
	The **minmmem.h** header file automatically includes **minmdur.h**. You must explicitly include **minmmem.h** to use advanced functions.

Neither **mi.h** nor **milib.h** provides access to the advanced header files. To use the advanced features, include the **minmmem.h** header file in your DataBlade API module to obtain declarations of DataBlade API functions and data types.

Tip: For a complete list of header files, check the **incl/public** subdirectory of the **INFORMIXDIR** directory.

ESQL/C Header Files

The following header files are provided to support some of the functions and data types of the IBM Informix ESQL/C library.

Header File	Contents
datetime.h	Structure and macro definitions for DATETIME and INTERVAL data types
decimal.h	Structure and macro definitions for DECIMAL and MONEY data types
int8.h	Declarations for structure and Informix ESQL/C library functions for the INT8 data type
sqlca.h	Structure definition that Informix ESQL/C uses to store error-status codes
	This structure enables you to check for the success or failure of SQL statements.
sqlda.h	Structure definition for value pointers and descriptions of dynamically defined variables
sqlhdr.h	Function prototypes of all Informix ESQL/C library functions
sqlstype.h	Definitions of strings that correspond to SQL statements
	Informix ESQL/C uses these strings when your program contains a DESCRIBE statement.
sqltypes.h	Integer constants that correspond to Informix ESQL/C language and SQL data types
	ESQL/C uses these constants when your program contains a DESCRIBE statement.
sqlxtype.h	Integer constants that correspond to C language and SQL data types that Informix ESQL/C uses in X/Open mode, when your program contains a DESCRIBE statement
varchar.h	Macros that you can use with the VARCHAR data type

Important: The **mitypes.h** header file automatically includes the **datetime.h**, **decimal.h**, and **int8.h** header files. In turn, the **milib.h** header file automatically includes **mitypes.h**, and **mi.h** automatically includes **milib.h**. Therefore, you automatically have access to the information in these Informix ESQL/C header files when you include **mi.h** in your DataBlade API module.

For additional information about the use of these Informix ESQL/C header files, see the following sections of this publication.

Header File	More Information
datetime.h	"The datetime.h Header File" on page 4-9
decimal.h	"The decimal.h Header File" on page 3-11
int8.h	"The int8.h Header File" on page 3-6

IBM Informix GLS Header File

A header file is provided to support the IBM Informix GLS library. If you use the IBM Informix GLS library in your DataBlade API module, include its header file, **ifxgls.h**, in your source code. For more information on the IBM Informix GLS library and how to use it in a DataBlade API module, see "Internationalization of DataBlade API Modules (GLS)" on page 1-19.

Private Header Files

If you define any opaque data types, you must include their header file in your DataBlade API source code. An opaque-type header file usually contains the declaration of the internal format for the opaque data type. For more information, see "Creating an Opaque Data Type" on page 16-1.

Public Data Types

The DataBlade API provides support for the following public data types:

- *DataBlade API data types*, which provide support for standard C, IBM Informix ESQL/C, and SQL data types
- DataBlade API *support data types*, which provide support for functions of the DataBlade API
- *DataBlade API data type structures*, which provide access to information that functions of the DataBlade API use

DataBlade API Data Types

To ensure portability across dissimilar computer architectures, the DataBlade API provides a set of data types, which Table 1-1 on page 1-8 shows. These data types begin with the **mi_** prefix. Most of these data types correspond to common SQL or C-language data types.

Table 1-1. DataBlade API, C, and SQL Data Types

DataBlade API Data Type	Standard C or ESQL/C Data Type	SQL Data Type
Character Data Types:		
mi_char	C: **char**	CHAR, VARCHAR, IDSSECURITYLABEL, GLS: NCHAR, NVARCHAR
mi_char1	C: **char**	CHAR(1)
mi_unsigned_char1	C: unsigned char	None
mi_wchar (deprecated)	C: unsigned two-byte integer	None
mi_string	C: **char ***	CHAR, VARCHAR, GLS: NCHAR, NVARCHAR

Table 1-1. DataBlade API, C, and SQL Data Types (continued)

DataBlade API Data Type	Standard C or ESQL/C Data Type	SQL Data Type
mi_lvarchar	Informix ESQL/C: **lvarchar** (though **lvarchar** *is* null-terminated and **mi_lvarchar** is *not*)	LVARCHAR Within C UDRs: for CHAR, NCHAR, TEXT, VARCHAR, and NVARCHAR arguments and return value
Integer Numeric Data Types:		
mi_sint1	C: signed one-byte integer	None
mi_int1	C: unsigned one-byte integer, char	None
mi_smallint	C: signed two-byte integer (**short integer** on many systems)	SMALLINT
mi_unsigned_smallint	C: unsigned two-byte integer	None
mi_integer	C: signed four-byte integer (**long integer** on many systems)	INTEGER, SERIAL
mi_unsigned_integer	C: unsigned four-byte integer	None
mi_int8	C: signed eight-byte integer; Informix ESQL/C: **int8**, **ifx_int8_t**	INT8, SERIAL8
mi_unsigned_int8	C: unsigned eight-byte integer; Informix ESQL/C: **int8**, **ifx_int8_t**	None
Fixed-Point Numeric Data Types:		
mi_decimal, mi_numeric	Informix ESQL/C: **decimal**, **dec_t**	DECIMAL(p,s) (fixed-point)
mi_money	Informix ESQL/C: **decimal**, **dec_t**	MONEY
Floating-Point Numeric Data Types:		
mi_decimal	Informix ESQL/C: **decimal**, **dec_t**	DECIMAL(p) (floating-point)
mi_real	C: **float**	SMALLFLOAT, REAL
mi_double_precision	C: **double**	FLOAT, DOUBLE PRECISION
Date and Time Data Types:		
mi_date	C: four-byte integerInformix ESQL/C: **date**	DATE
mi_datetime	Informix ESQL/C: **datetime**, **dtime_t**	DATETIME
mi_interval	Informix ESQL/C: **interval**, **intrvl_t**	INTERVAL
Varying-Length Data Types:		
mi_lvarchar	C: **void *** Informix ESQL/C: **lvarchar** (though **lvarchar** *is* null-terminated and **mi_lvarchar** is *not*)	LVARCHAR, Opaque types Within C UDRs: for CHAR, NCHAR, TEXT, VARCHAR, and NVARCHAR arguments and return value
mi_sendrecv	C: **void ***	SENDRECV, opaque-type support functions: send, receive
mi_impexp	C: **void ***	IMPEXP, opaque-type support functions: import, export
mi_impexpbin	C: **void ***	IMPEXPBIN, opaque-type support functions: importbin, exportbin

Table 1-1. DataBlade API, C, and SQL Data Types (continued)

DataBlade API Data Type	Standard C or ESQL/C Data Type	SQL Data Type
mi_bitvarying	C: **void ***	BITVARYING
Complex Data Types:		
MI_COLLECTION	C: **void ***	SET, LIST, MULTISET
MI_ROW	C: **void ***	ROW (unnamed row type), Named row type
Other Data Types:		
mi_boolean	C: **char**Informix ESQL/C: **boolean**	BOOLEAN
mi_pointer	C: **void ***	POINTER
MI_LO_HANDLE	None	CLOB, BLOB
		Smart large objects

Important: To make your DataBlade API module portable, it is recommended that you use the DataBlade API platform-independent data types (such as **mi_integer**, **mi_smallint**, **mi_real**, **mi_boolean**, and **mi_double_precision**) instead of their C-language counterparts. These data types handle the different sizes of numeric values across computer architectures.

—————————————— **Server Only** ——————————————

Table 1-1 on page 1-8 lists the DataBlade API data types and SQL data types. However, when you pass some of these data types to and from C UDRs, you must pass them as pointers rather than as actual values. For more information, see "Passing Mechanism for MI_DATUM Values" on page 12-22.

—————————————— **End of Server Only** ——————————————

Table 1-2 shows where you can find information about how DataBlade API data types correspond to SQL data types.

Table 1-2. Correspondence of SQL Data Types to DataBlade API Data Types

SQL Data Type	Information on Corresponding DataBlade API Data Types
BITVARYING	"The mi_bitvarying Data Type" on page 2-28
BLOB	Chapter 6, "Using Smart Large Objects," on page 6-1
BOOLEAN	"Boolean Data Types" on page 2-30
BYTE	"Simple Large Objects" on page 2-32
CHAR	"Character Data Types" on page 2-7
CLOB	Chapter 6, "Using Smart Large Objects," on page 6-1
DATE	Chapter 4, "Using Date and Time Data Types," on page 4-1
DATETIME	Chapter 4, "Using Date and Time Data Types," on page 4-1
DECIMAL	Chapter 3, "Using Numeric Data Types," on page 3-1
Distinct	Chapter 16, "Extending Data Types," on page 16-1
FLOAT	Chapter 3, "Using Numeric Data Types," on page 3-1
INT8	Chapter 3, "Using Numeric Data Types," on page 3-1

Table 1-2. Correspondence of SQL Data Types to DataBlade API Data Types (continued)

SQL Data Type	Information on Corresponding DataBlade API Data Types
INTEGER	Chapter 3, "Using Numeric Data Types," on page 3-1
INTERVAL	Chapter 4, "Using Date and Time Data Types," on page 4-1
LIST	Chapter 5, "Using Complex Data Types," on page 5-1
LVARCHAR	"Varying-Length Data Type Structures" on page 2-13
MONEY	Chapter 3, "Using Numeric Data Types," on page 3-1
MULTISET	Chapter 5, "Using Complex Data Types," on page 5-1
NCHAR	"Character Data Types" on page 2-7
NVARCHAR	"Character Data Types" on page 2-7
Opaque	Chapter 16, "Extending Data Types," on page 16-1
POINTER	"Pointer Data Types (Server)" on page 2-31
ROW	Chapter 5, "Using Complex Data Types," on page 5-1
SERIAL	Chapter 3, "Using Numeric Data Types," on page 3-1
SERIAL8	Chapter 3, "Using Numeric Data Types," on page 3-1
SET	Chapter 3, "Using Numeric Data Types," on page 3-1
SMALLFLOAT	Chapter 3, "Using Numeric Data Types," on page 3-1
SMALLINT	Chapter 3, "Using Numeric Data Types," on page 3-1
TEXT	"Simple Large Objects" on page 2-32
VARCHAR	"Character Data Types" on page 2-7

DataBlade API Support Data Types

The DataBlade API provides additional data types that DataBlade API functions use. These data types are usually enumerated data types that restrict valid values for an argument or return value of a DataBlade API function. Most of these data types, which Table 1-3 lists, start with the **MI_** prefix.

Table 1-3. DataBlade API Support Data Types

Support Data Type	Purpose	Location of Description
MI_CALLBACK_STATUS	Enumerates valid return values of a callback function	"Return Value of a Callback Function" on page 10-13
MI_CURSOR_ACTION	Enumerates movements through a cursor	"Positioning the Cursor" on page 5-6 "Fetching Rows Into a Cursor" on page 8-23
MI_EVENT_TYPE	Classifies an event	"DataBlade API Event Types" on page 10-2
MI_FUNCARG	Enumerates kinds of arguments that a companion UDR might receive	"MI_FUNCARG Data Type" on page 15-56
mi_funcid	Holds a routine identifier	"Routine Resolution" on page 12-19
MI_ID	Enumerates the kinds of identifiers that the **mi_get_id( )** function can obtain	Description of **mi_get_id( )** in the *IBM Informix DataBlade API Function Reference*
MI_SETREQUEST	Enumerates values of the iterator-status constant, which the database server can return to a UDR through the **mi_fp_request( )** function	"Writing an Iterator Function" on page 15-3

Table 1-3. DataBlade API Support Data Types (continued)

Support Data Type	Purpose	Location of Description
MI_TRANSITION_TYPE	Enumerates types of state transitions in a transition descriptor	"Understanding State-Transition Events" on page 10-49
MI_UDR_TYPE	Enumerates the kind of UDR for which the **mi_routine_get_by_typeid( )** function obtains a function descriptor	Description of **mi_routine_get_by_typeid( )** in the *IBM Informix DataBlade API Function Reference*

DataBlade API Data Type Structures

Many DataBlade API functions provide information for DataBlade API modules in special data type structures. The names of these data type structures begin with the **MI_** prefix. Table 1-4 lists these data type structures, their purposes, and where you can find detailed descriptions of them.

Table 1-4. DataBlade API Data Type Structures

DataBlade API Data Type Structure	Purpose	More Information
MI_COLL_DESC	*Collection descriptor*, which describes the structure of a collection	"Using a Collection Descriptor" on page 5-3
MI_COLLECTION	*Collection structure*, which contains the elements of a collection	"Using a Collection Structure" on page 5-3
MI_CONNECTION	*Connection descriptor*, which contains the execution context for a connection	"Establishing a Connection" on page 7-11
MI_CONNECTION_INFO	*Connection-information descriptor*, which contains connection parameters for an open connection	"Using Connection Parameters" on page 7-4
MI_DATABASE_INFO	*Database-information descriptor*, which contains database parameters for an open connection	"Using Database Parameters" on page 7-6
MI_DATUM	*Datum*, which provides a transport mechanism to pass data of an SQL data type by value or by reference	"The MI_DATUM Data Type" on page 2-32
MI_ERROR_DESC	*Error descriptor*, which describes an exception	"Event Information" on page 10-17
MI_FPARAM	*Function-parameter structure*, which holds information about a UDR that the routine can access during its execution	"Accessing MI_FPARAM Routine-State Information" on page 9-2
MI_FUNCARG	*Function-argument structure*, which holds information about the argument of a companion UDR	"MI_FUNCARG Data Type" on page 15-56
MI_FUNC_DESC	*Function descriptor*, which describes a UDR that is to be invoked with the Fastpath interface	"Obtaining a Function Descriptor" on page 9-17
MI_LO_FD	*LO file descriptor*, which describes an open smart large object	"Obtaining an LO File Descriptor" on page 6-41
MI_LO_HANDLE	*LO handle*, which identifies the location of a smart large object in its sbspace	"Obtaining an LO Handle" on page 6-40
MI_LO_SPEC	*LO-specification structure*, which contains storage characteristics for a smart large object	"Obtaining the LO-Specification Structure" on page 6-25

Table 1-4. DataBlade API Data Type Structures (continued)

DataBlade API Data Type Structure	Purpose	More Information
MI_LO_STAT	*LO-status structure*, which contains status information for a smart large object	"Obtaining Status Information for a Smart Large Object" on page 6-52
MI_PARAMETER_INFO	*Parameter-information descriptor*, which specifies whether callbacks are enabled or disabled and whether pointers are checked in client LIBMI applications	"Using Session Parameters" on page 7-8
MI_ROW	*Row* (or *row structure*), which contains either the column values of a table row or field values of a row type	"Retrieving Rows" on page 8-41 "Using a Row Structure" on page 5-32
MI_ROW_DESC	*Row descriptor*, which describes the structure of a row	"Obtaining Row Information" on page 8-40 "Using a Row Descriptor" on page 5-29
MI_SAVE_SET	*Save-set descriptor*, which describes a save set	"Creating a Save Set" on page 8-60
MI_STATEMENT	*Statement descriptor*, which describes a prepared SQL statement	"Executing Prepared SQL Statements" on page 8-11
mi_statret	*Statistics-return structure* (C language structure), which holds the collected statistics for a user-defined data type	"SET_END in statcollect( )" on page 16-45
MI_STREAM	*Stream descriptor*, which describes an open stream A *stream* is an object that can be written to or read from. The DataBlade API has functions for the following predefined stream classes: • File stream • String stream • Varying-length-data stream	
MI_TRANSITION_DESC	*Transition descriptor*, which describes a state transition	"Understanding State-Transition Events" on page 10-49
MI_TYPEID	*Type identifier*, which uniquely identifies a data type within a database	"Type Identifiers" on page 2-2
MI_TYPE_DESC	*Type descriptor*, which provides information about a data type	"Type Descriptors" on page 2-3

The DataBlade API provides constructor and destructor functions for most of these public data type structures. These functions handle memory allocation of these data type structures, as follows:

• The constructor function for a DataBlade API data type structure creates a new instance of the data type structure.

  A constructor function usually returns a pointer to the DataBlade API data type structure and allocates memory for the structure.

─────────── **Server Only** ───────────
The memory allocation is in the current memory duration, which is PER_ROUTINE by default. For more information, see "Choosing the Memory

Duration" on page 14-4.

─── End of Server Only ───

- The destructor function for a DataBlade API data type structure frees the instance of the data type structure.

  You specify a pointer to the DataBlade API data type structure to the destructor function. The destructor function deallocates memory for the specified data type structure. Call destructor functions only for DataBlade API data type structures that you explicitly allocated with the corresponding constructor function.

Regular Public Functions

The DataBlade API provides support for the following kinds of functions in a DataBlade API module.

Kind of Functions	Purpose
DataBlade API functions	Provide access to the database server
IBM Informix ESQL/C functions	Provide operations on certain data types
IBM Informix GLS functions	Provide the ability to internationalize your DataBlade API module

DataBlade API Functions

The DataBlade API functions begin with the **mi_** prefix. The **milib.h** header file declares most of these DataBlade API functions. The **mi.h** header file automatically includes **milib.h**. You must include **mi.h** in any DataBlade API module that uses a DataBlade API function.

The functions of the DataBlade API function library can be divided into the following categories.

Category of DataBlade API Functions	More Information
Data handling:	
Obtaining type information	"Type Identifiers" on page 2-2
	"Type Descriptors" on page 2-3
Transferring data types between computers (database server only)	"Conversion of Opaque-Type Data with Computer-Specific Data Types" on page 16-21
Converting data to a different data type	"DataBlade API Functions for Date Conversion" on page 4-3
	"DataBlade API Functions for Date-Time or Interval Conversion" on page 4-13
	"DataBlade API Functions for Decimal Conversion" on page 3-14
	"DataBlade API Functions for String Conversion" on page 2-11
Handling collections: sets, multisets, and lists	"Collections" on page 5-2
Converting between code sets (database server only)	"Internationalization of DataBlade API Modules (GLS)" on page 1-19
Handling collections	"Collections" on page 5-2

Category of DataBlade API Functions	More Information
Managing varying-length structures	"Varying-Length Data Type Structures" on page 2-13
Obtaining SERIAL values	"Processing Insert Results" on page 8-59
Handling NULL values	"SQL NULL Value" on page 2-36
Session, thread, and transaction management:	
Obtaining connection information	"Using Connection Parameters" on page 7-4
	"Using Database Parameters" on page 7-6
	"Using Session Parameters" on page 7-8
Establishing a connection	"Establishing a Connection" on page 7-11
Initializing the DataBlade API	"Initializing the DataBlade API" on page 7-17
Managing Informix threads (database server only)	"Yielding the CPU VP" on page 13-19
	"Managing Stack Usage" on page 14-35
Obtaining transaction and server-processing state changes	"Using a Transition Descriptor" on page 10-19
SQL statement processing:	
Sending SQL statements	"Executing Basic SQL Statements" on page 8-6
	"Executing Prepared SQL Statements" on page 8-11
Obtaining statement information	"Returning a Statement Descriptor" on page 8-14
	"Obtaining Input-Parameter Information" on page 8-15
Obtaining result information	"Processing Statement Results" on page 8-33
Retrieving rows and row data (also row types and row-type data)	"Obtaining Row Information" on page 8-40
	"Retrieving Rows" on page 8-41
Retrieving columns	"Obtaining Column Information" on page 8-41
	"Obtaining Column Values" on page 8-42
Using save sets	"Using Save Sets" on page 8-59
Executing user-defined-routines:	
Accessing an **MI_FPARAM** structure	"Accessing MI_FPARAM Routine-State Information" on page 9-2
Allocating an **MI_FPARAM** structure	"Using a User-Allocated MI_FPARAM Structure" on page 9-36
Using the Fastpath interface	"Calling UDRs with the Fastpath Interface" on page 9-14
Accessing a function descriptor	"Obtaining Information from a Function Descriptor" on page 9-23
Executing selectivity and cost functions:	"Writing Selectivity and Cost Functions" on page 15-54
Memory management:	
Managing user memory	"Managing User Memory" on page 14-20

Category of DataBlade API Functions	More Information
Managing named memory (database server only)	"Managing Named Memory" on page 14-24
Exception handling:	
Raising a database exception	"Raising an Exception" on page 10-40
Accessing an error descriptor	"Using an Error Descriptor" on page 10-17, "Handling Multiple Exceptions" on page 10-38
Using callback functions	"Invoking a Callback" on page 10-3
Smart-large-object interface:	
Creating a smart large object	"Functions That Create a Smart Large Object" on page 6-19
Performing I/O on a smart large object	"Functions That Perform Input and Output on a Smart Large Object" on page 6-20
Moving smart large objects to and from operating-system files	"Functions That Move Smart Large Objects to and from Operating-System Files" on page 6-24
Manipulating LO handles	"Functions That Manipulate an LO Handle" on page 6-21
Handling LO-specification structures	"Functions That Access an LO-Specification Structure" on page 6-22
Handling smart-large-object status	"Functions That Access an LO-Status Structure" on page 6-23
Operating-system file interface:	"Access to Operating-System Files" on page 13-52
Tracing (database server):	"Using Tracing" on page 12-28

For a complete list of DataBlade API functions in each of these categories, see the *IBM Informix DataBlade API Function Reference*, which provides descriptions of the regular public and advanced functions, in alphabetical order. For more information on advanced functions of the DataBlade API, see "Advanced Features (Server)" on page 1-18.

If an error occurs while a DataBlade API function executes, the function usually indicates the error with one of the following return values.

Way to Indicate an Error	More Information
Functions that return a pointer return the NULL-valued pointer	"NULL-Valued Pointer" on page 2-37
Functions that return an **mi_integer** value (or other integer) return the MI_ERROR status code	"Handling Errors from DataBlade API Functions" on page 10-26
Functions that raise an exception	"Handling Errors from DataBlade API Functions" on page 10-26

IBM Informix ESQL/C Functions
In a DataBlade API module, you can use some of the functions in the IBM Informix ESQL/C library functions to perform conversions and operations on

different data types. The Informix ESQL/C functions do *not* begin with the **mi_** prefix. Various header files declare these functions. For more information, see "ESQL/C Header Files" on page 1-7.

The functions of the Informix ESQL/C function library that are valid in a DataBlade API module can be divided into the following categories.

Category of DataBlade API Function	More Information
Byte handling	"Manipulating Byte Data" on page 2-29
Character processing	"ESQL/C Functions for String Conversion" on page 2-12
	"Operations on Character Values" on page 2-12
DECIMAL-type and MONEY-type processing	"ESQL/C Functions for Decimal Conversion" on page 3-15
	"Performing Operations on Decimal Data" on page 3-16
DATE-type processing	"ESQL/C Functions for Date Conversion" on page 4-4
	"Operations on Date Data" on page 4-5
DATETIME-type processing and INTERVAL-type processing	"ESQL/C Functions for Date, Time, and Interval Conversion" on page 4-13
	"Operations on Date and Time Data" on page 4-15
INT8-byte processing	"Converting INT8 Values" on page 3-7
	"Performing Operations on Eight-Byte Values" on page 3-8
Processing for other C-language data types	"Formatting Numeric Strings" on page 3-20

For a complete list of Informix ESQL/C functions in each of these categories, see the *IBM Informix DataBlade API Function Reference*, which provides descriptions of these public functions, in alphabetical order.

IBM Informix GLS Functions

The IBM Informix GLS library is an API that lets developers of DataBlade API modules create internationalized applications. This library is a threadsafe library. The macros and functions of IBM Informix GLS provide access to the GLS locales, which contain culture-specific information.

The IBM Informix GLS library contains functions that provide the following capabilities:

- Process single-byte and multibyte characters

  These functions are useful for processing character data in the NCHAR and NVARCHAR data types, which can contain locale-specific information.
- Format date, time, and numeric data to locale-specific formats

  These functions provide the ability to handle end-user formats for the DATE, DATETIME, DECIMAL, and MONEY data types.

The **mi.h** header file does *not* automatically include the IBM Informix GLS library. For more information on the IBM Informix GLS library and how to use it in a DataBlade API module, see "Internationalization of DataBlade API Modules (GLS)" on page 1-19.

Advanced Features (Server)

The DataBlade API provides a set of advanced features to handle specialized needs of a UDR or DataBlade module that the regular public features cannot handle. Table 1-5 lists the advanced DataBlade API features.

Table 1-5. Advanced Features of the DataBlade API

Advanced Feature	Description	More Information
Named memory	Enables a UDR to obtain a memory address through a name assigned to the memory block	"Managing Named Memory" on page 14-24
Memory durations	Provides a UDR with memory durations that exceed its lifetime	"Advanced Memory Durations" on page 14-13
Session-duration connection descriptor	Enables a UDR to cache connection information for the length of a session	"Obtaining a Session-Duration Connection Descriptor" on page 7-13
Session-duration function descriptor	Enables a UDR to cache function descriptors in named memory so that many UDRs can execute the same UDR through Fastpath	"Reusing a Function Descriptor" on page 9-30
Controlling the VP environment	Enables a UDR to obtain dynamically information about the VP and VP class in which it executes and to make some changes to this environment	"Controlling the VP Environment" on page 13-38
Setting the row and column identifier in the **MI_FPARAM** structure of a UDR	Enables a UDR to change the row associated with a UDR	Descriptions of **mi_fp_setcolid( )** and **mi_fp_setrow( )** in the *IBM Informix DataBlade API Function Reference*
Obtaining the current **MI_FPARAM** address	Enables a UDR to obtain dynamically the address of its own **MI_FPARAM** structure	Description of **mi_fparam_get_current( )** in the *IBM Informix DataBlade API Function Reference*
Microseconds component of last-modification time for a smart large object	Enables UDRs to maintain the microseconds component of last-modification time, which the database server does not maintain	Description of **mi_lo_utimes( )** in the *IBM Informix DataBlade API Function Reference*

Warning: These DataBlade API features can adversely affect your UDR if you use them incorrectly. Use them only when the public DataBlade API features cannot perform the tasks you need done.

Internationalization of DataBlade API Modules (GLS)

For your DataBlade API module to work in any IBM Informix locale, you must implement your DataBlade API module so that it is *internationalized*. That is, the module must not make any assumptions about the locale in which it will execute.

Server Only

A C UDR inherits the server-processing locale as its current processing locale. The database server dynamically creates a server-processing locale for a particular session when a client application establishes a connection. The database server uses the client locale, database locale, the server locale, and information from the client application to determine the server-processing locale. For more information on how the database server determines the server-processing locale, see the *IBM Informix GLS User's Guide*.

End of Server Only

Client Only

A client LIBMI application performs its I/O tasks in the client locale. Any database requests that the application makes execute on the database server in the server-processing locale.

End of Client Only

This section provides the following information about how to internationalize a C UDR and the support that the DataBlade API provides for internationalized UDRs.

An internationalized C UDR must handle the following GLS considerations.

GLS Consideration for an Internationalized UDR	DataBlade API Function
What considerations must the C UDR take when copying character data?	None
How can the C UDR access GLS locales?	IBM Informix GLS function library
How does the UDR handle code-set conversion?	**mi_get_string( )** **mi_put_string( )** IBM Informix GLS function library
How does the UDR handle locale-specific end-user formats?	**mi_date_to_string( )**, **mi_decimal_to_string( )**, **mi_interval_to_string( )**, **mi_money_to_string( )**, **mi_string_to_date( )**, **mi_string_to_decimal( )**, **mi_string_to_interval( )**, **mi_string_to_money( )**
How can the C UDR access internationalized exception messages?	**mi_db_error_raise( )**
How can the C UDR access internationalized tracing messages?	GL_DPRINTF, **gl_tprintf( )**
How do opaque-type support functions handle locale-sensitive data?	**mi_get_string( )**, **mi_put_string( )**

GLS Consideration for an Internationalized UDR	DataBlade API Function
How to you obtain names of the different locales from within a C UDR?	**mi_client_locale( )**, **mi_get_connection_info( )**

For more information on how to handle these GLS considerations within a C UDR, see the chapter on database servers in the *IBM Informix GLS User's Guide*.

Chapter 2. Accessing SQL Data Types

In This Chapter

This chapter provides an overview of the data types that the DataBlade API supports. It also describes DataBlade API support for the following types of data:

- Text and strings
- Varying-length structures
- Byte data
- Miscellaneous SQL data types: POINTER, BOOLEAN, and simple large objects
- The **MI_DATUM** structure
- The NULL constant

For references to discussions of different SQL data types in this publication, see Table 1-2 on page 1-10.

Table 1-1 on page 1-8 lists the correspondences between SQL and DataBlade API data types. To declare a variable for an SQL data type, use the appropriate DataBlade API predefined data type or structure for the variable. The **mi.h** header file includes the header files for the definitions of all DataBlade API data types. Include **mi.h** in all DataBlade API modules that use DataBlade API data types.

The DataBlade API represents the SQL data type of a column value with the following data type structures:

- A short name, called the *type identifier*, which identifies only the data type
- A long name, called the *type descriptor*, which provides the data type and information about this type

---
Server Only

Type descriptors and type identifiers do not have an associated memory duration. The DataBlade API allocates them from a special data type cache.

End of Server Only

Type Identifiers

A *type identifier*, **MI_TYPEID**, is a DataBlade API data type structure that identifies a data type uniquely. For extended data types, the type identifier is database-dependent; that is, the same type identifier might identify different data types for different databases. You can determine the data type that a type identifier represents with the following DataBlade API functions.

Type-Identifier Check	DataBlade API Function
Are two type identifiers equal?	**mi_typeid_equals( )**
Does the type identifier represent a *built-in* data type?	**mi_typeid_is_builtin( )**
Does the type identifier represent a *collection* (SET, MULTISET, LIST) data type?	**mi_typeid_is_collection( )**
Does the type identifier represent a *complex* data type (row type or collection)?	**mi_typeid_is_complex( )**
Does the type identifier represent a *distinct* data type?	**mi_typeid_is_distinct( )**
Does the type identifier represent a *LIST* data type?	**mi_typeid_is_list( )**

Type-Identifier Check	DataBlade API Function
Does the type identifier represent a *MULTISET* data type?	**mi_typeid_is_multiset( )**
Does the type identifier represent a *row* type (named or unnamed)?	**mi_typeid_is_row( )**
Does the type identifier represent a *SET* data type?	**mi_typeid_is_set( )**

Important: To a DataBlade API module, the type identifier (**MI_TYPEID**) is an opaque C data structure. Do not access its internal fields directly. The internal structure of a type identifier may change in future releases. Therefore, to create portable code, always use the accessor functions for this structure to determine data type.

The DataBlade API uses type identifiers in the following situations.

Type Identifier Usage	DataBlade API Function	More Information
To indicate a column type in a row descriptor	**mi_column_type_id( )**	"Obtaining Column Information" on page 8-41
To indicate data type of arguments in a user-defined routine (UDR)	**mi_fp_argtype( )**, **mi_fp_setargtype( )**	"Determining the Data Type of UDR Arguments" on page 9-3
To indicate data type of return type of a UDR	**mi_fp_rettype( )**, **mi_fp_setrettype( )**	"Determining the Data Type of UDR Return Values" on page 9-6
To indicate data type of a column with which an input parameter in a prepared statement is associated	**mi_parameter_type_id( )**	"Obtaining Input-Parameter Information" on page 8-15
To identify a UDR by the data types of its arguments to generate its function descriptor	**mi_routine_get_by_typeid( )**	"Looking Up UDRs" on page 9-18
To identify a cast function by the source and target data types to generate its function descriptor	**mi_cast_get( )**	"Looking Up Cast Functions" on page 9-20
To identify the element type of a collection	**mi_collection_create( )**	"Creating a Collection" on page 5-3

Type Descriptors

A type descriptor, **MI_TYPE_DESC**, is a DataBlade API data type structure that contains information about an SQL data type. For built-in data types, this information comes from the **syscolumns** table. For extended data types, it contains the information in the **sysxtdtypes** table. Table 2-1 lists the DataBlade API accessor functions that obtain information from a type descriptor.

Table 2-1. Data Type Information in a Type Descriptor

Data Type Information	DataBlade API Accessor Function
The *alignment*, in number of bytes, of the data type	**mi_type_align( )**

Table 2-1. Data Type Information in a Type Descriptor (continued)

Data Type Information	DataBlade API Accessor Function
Whether a value of the data type is *passed by reference* or *passed by value*	**mi_type_byvalue( )**
A type descriptor for the element type of a collection data type	**mi_type_element_typedesc( )**
The *full name* (*owner.type_name*) of the data type	**mi_type_full_name( )**
The *length* of the data type	**mi_type_length( )**
The maximum length of the data type	**mi_type_maxlength( )**
The *owner* of the data type	**mi_type_owner( )**
The *precision* (total number of digits) of the data type	**mi_type_precision( )**
The *qualifier* of a DATETIME or INTERVAL data type	**mi_type_qualifier( )**
The *scale* of a data type	**mi_type_scale( )**
The short *name* (no *owner*) of the data type	**mi_type_typedesc( )**
The *type identifier* for the data type	**mi_typedesc_typeid( )**

Important: To a DataBlade API module, the type descriptor (**MI_TYPE_DESC**) is an opaque C data structure. Do not access its internal fields directly. The internal structure of **MI_TYPE_DESC** may change in future releases. Therefore, to create portable code, always use the accessor functions in Table 2-1 to obtain values from this structure.

The DataBlade API uses type descriptors in the following situations.

Type Descriptor Usage	More Information
To indicate a column type in a row descriptor	Description of **mi_column_typedesc( )** in the *IBM Informix DataBlade API Function Reference*
	"Obtaining Column Information" on page 8-41
To obtain the source type of a distinct type	Description of **mi_get_type_source_type( )** in the *IBM Informix DataBlade API Function Reference*
To process returned row data, especially when not all the rows returned by a query have the same size and structure	Description of **mi_get_row_desc_from_type_desc( )** in the *IBM Informix DataBlade API Function Reference*
To identify a cast function by the source and target data types to generate its function descriptor	Description of **mi_td_cast_get( )** in the *IBM Informix DataBlade API Function Reference*
	"Looking Up Cast Functions" on page 9-20

Type-Structure Conversion

You can use the following DataBlade API functions to obtain a type descriptor or type identifier.

Convert from	Convert to	DataBlade API Function
Type identifier	Type descriptor	Description of **mi_type_typedesc( )** in the *IBM Informix DataBlade API Function Reference*

Convert from	Convert to	DataBlade API Function
Type descriptor	Type identifier	Description of **mi_typedesc_typeid( )** in the *IBM Informix DataBlade API Function Reference*
Type name (as **mi_lvarchar**)	Type identifier	Description of **mi_typename_to_id( )** in the *IBM Informix DataBlade API Function Reference*
Type name (as **mi_lvarchar**)	Type descriptor	Description of **mi_typename_to_typedesc( )** in the *IBM Informix DataBlade API Function Reference*
Type name (as string: **char ***)	Type identifier	Description of **mi_typestring_to_id( )** in the *IBM Informix DataBlade API Function Reference*
Type name (as string: **char ***)	Type descriptor	Description of **mi_typestring_to_typedesc( )** in the *IBM Informix DataBlade API Function Reference*

Data Type Descriptors and Column Type Descriptors

A type descriptor for a data type and a type descriptor for a column use the same accessor functions and share the same underlying data type structures. These descriptors differ, however, in the handling of parameterized data types (such as DATETIME, INTERVAL, DECIMAL, and money), as follows:

- A data type descriptor holds *unparameterized* information, which is general information about the data type.
- A column type descriptor holds *parameterized* information, which is the information for the data type of a particular column.

Table 2-1 on page 2-3 lists the DataBlade API accessor functions that obtain information from a type descriptor. When you use type-descriptor accessor functions on parameterized data types, the results depend on which kind of type descriptor you pass into the accessor function.

For example, Figure 2-1 shows a named row type with fields that have parameterized data types.

```
CREATE ROW TYPE row_type
   (time_fld DATETIME YEAR TO SECOND,
    dec_fld DECIMAL(6,3));
```

Figure 2-1. Sample Named Row Type with Parameterized Fields

Figure 2-2 shows a code fragment that obtains a data type descriptor and a column type descriptor for the *first* field (**time_fld**) from the row descriptor (**row_desc**) for the **row_type** row type.

```
type_id = mi_column_type_id(row_desc, 0);
type_desc = mi_type_typedesc(conn, type_id);
col_type_desc = mi_column_type_desc(row_desc, 0);
```

Figure 2-2. Type Descriptor and Column Type Descriptor for DATETIME Field

For the DATETIME data type of the **time_fld** column, the type-descriptor accessor functions obtain different qualifier information for each kind of type descriptor, as follows:

- The data type descriptor, **type_desc**, stores the *unparameterized* type information for the DATETIME data type.

   The following code fragment calls the **mi_type_typename( )** and **mi_type_qualifier( )** accessor functions on the **type_desc** type descriptor (which Figure 2-2 defines):

   ```
   type_string = mi_type_typename(type_desc);
   type_scale = mi_type_qualifier(type_desc);
   ```

   The call to **mi_type_typename( )** returns the string "datetime" as the unparameterized name of the data type. The call to **mi_type_qualifier( )** returns zero (0) as the type qualifier.

- The column type descriptor, **col_type_desc**, stores the *parameterized* type information for the DATETIME field of **row_type**.

   The following code fragment calls the **mi_type_typename( )** and **mi_type_qualifier( )** accessor functions on the **col_type_desc** type descriptor (which Figure 2-2 defines):

   ```
   type_string = mi_type_typename(col_type_desc);
   type_scale = mi_type_qualifier(col_type_desc);
   ```

   The call to **mi_type_typename( )** returns the string "datetime year to second" as the parameterized name of the data type. The call to **mi_type_qualifier**( ) returns the actual DATETIME qualifier of 3594, which is the encoded qualifier value for:

   ```
   TU_DTENCODE(TU_YEAR, TU_SECOND)
   ```

Similarly, for DECIMAL and MONEY data types, the type-descriptor accessor functions can obtain scale and precision information from a column type descriptor but not a data type descriptor. Figure 2-3 shows a code fragment that obtains a data type descriptor and a column type descriptor for the second field (**dec_fld**) from the row descriptor (**row_desc**) for the **row_type** row type.

```
type_id2 = mi_column_type_id( row_desc, 1 );
type_desc2 = mi_type_typedesc( conn, type_id2 );
col_type_desc2 = mi_column_type_desc( row_desc, 1 );
```

Figure 2-3. Type Descriptor and Column Type Descriptor for DECIMAL Field

For the DECIMAL data type of the **dec_fld** column, the results from the type-descriptor accessor functions depend on which type descriptor you pass into the accessor function, as follows:

- The data type descriptor, **type_desc2**, stores the *unparameterized* type information for DECIMAL.

   The following code fragment calls the **mi_type_precision( )** and **mi_type_scale( )** accessor functions on the **type_desc2** type descriptor (which Figure 2-3 defines):

```
type_prec = mi_type_precision(type_desc2);
type_scale = mi_type_scale(type_desc2);
```

Both the **mi_type_precision( )** and **mi_type_scale( )** functions return zero (0) for the precision and scale.

- The column type descriptor, **col_type_desc**, stores the *parameterized* type information for the DECIMAL field of **row_type**.

  The following code fragment calls the **mi_type_precision( )** and **mi_type_scale( )** accessor functions on the **col_type_desc2** type descriptor (which Figure 2-3 defines):

```
type_prec = mi_type_precision(col_type_desc2);
type_scale = mi_type_scale(col_type_desc2);
```

  The **mi_type_precision( )** and **mi_type_scale( )** functions return the actual precision and scale of the DECIMAL column, 6 and 3, respectively.

Character Data Types

The DataBlade API supports the following data types that can hold character data in a DataBlade API module.

DataBlade API Character Data Type	Description	SQL Character Data Type
mi_char1	One-byte character	None
mi_unsigned_char1	Unsigned one-byte character	None
mi_char, **mi_string**	Character string or array	CHAR, VARCHAR, NCHAR, NVARCHAR, IDSSECURITYLABEL
mi_lvarchar	Varying-length structure to hold varying-length character data	LVARCHAR
MI_LO_HANDLE	LO handle to a smart large object that holds character data	CLOB

Tip: The database server also supports the TEXT data type for character data. It stores TEXT character data as a simple large object. However, the DataBlade API does not directly support simple large objects. For more information, see "Simple Large Objects" on page 2-32.

The mi_char1 and mi_unsigned_char1 Data Types

The **mi_char1** and **mi_unsigned_char1** data types hold a single-byte character. These data types can also hold an integer quantity within C code so you can also use **mi_unsigned_char1** to hold unsigned one-byte integer values.

Important: To make your DataBlade API module portable, It is recommended that you use the DataBlade API data type **mi_char1** for single-character values instead of the native C-language counterpart, **char**. The **mi_char1** data type ensures a consistent size across computer architectures.

─────────────── **Global Language Support** ───────────────

The **mi_char1** and **mi_unsigned_char1** data types assume that one character uses one byte of storage. Therefore, do *not* use these data types to hold multibyte characters (which can require up to four bytes of storage). Instead, use the **mi_char**, **mi_string**, or **mi_lvarchar** data type. For more information on multibyte

characters, see the *IBM Informix GLS User's Guide*.

_____ **End of Global Language Support** _____

_____ **Server Only** _____

The **mi_char1** and **mi_unsigned_char1** data types are guaranteed to be one byte on *all* computer architectures. Therefore, they *can* fit into an **MI_DATUM** structure and can be passed by value in C UDRs.

_____ **End of Server Only** _____

_____ **Client Only** _____

All data types, including **mi_char1** and **mi_unsigned_char1**, must be passed by reference in client LIBMI applications.

_____ **End of Client Only** _____

The mi_char and mi_string Data Types

The **mi_char** and **mi_string** data types are the DataBlade API equivalents of the **char** C-language data type. These two data types are exactly the same in both storage and functionality. Use them to declare character strings in your DataBlade API module.

You can use the **mi_char** or **mi_string** data type to hold CHAR, VARCHAR, or IDSSECURITYLABEL data, as long as this data is not an argument or return value of a C UDR. For more information, see "Character Data in C UDRs (Server)" on page 2-10.

_____ **Global Language Support** _____

You can use the **mi_char** and **mi_string** data types to store multibyte characters (NCHAR and NVARCHAR columns). However, your code must track how many bytes each character contains. You can use the IBM Informix GLS interface to assist with this process. For more information on multibyte characters, see the *IBM Informix GLS User's Guide*.

_____ **End of Global Language Support** _____

_____ **Server Only** _____

The **mi_char** and **mi_string** data types *cannot* fit into an **MI_DATUM** structure. They must be passed by reference within C UDRs.

_____ **End of Server Only** _____

_____ **Client Only** _____

All data types, including **mi_char** and **mi_string**, must be passed by reference within client LIBMI applications.

_____ **End of Client Only** _____

The mi_lvarchar Data Type

The **mi_lvarchar** data type has the following uses:

- Holds data of an LVARCHAR column
- Holds character data that is passed to or received from an SQL statement when the query is in binary mode

---
Server Only

- Holds data for character arguments and return values of C UDRs
- Holds the external format of an opaque data type

End of Server Only

The following sections summarize each of these uses of an **mi_lvarchar**. For information about the structure of the **mi_lvarchar** data type, see "Varying-Length Data Type Structures" on page 2-13.

The SQL LVARCHAR Data Type

The LVARCHAR data type of SQL stores variable-length character strings whose length can be up to 32,739 bytes. LVARCHAR is a built-in opaque data type that is valid in distributed queries of tables, views, and synonyms of databases outside the local server. The DataBlade API supports the LVARCHAR data type with the **mi_lvarchar** data type, which is implemented in the DataBlade API as a varying-length structure.

Tip: The SQL data type LVARCHAR and the DataBlade API data type **mi_lvarchar** are not the same. Although you use **mi_lvarchar** to hold LVARCHAR data, **mi_lvarchar** is also used for other purposes.

If you declare no maximum size for an LVARCHAR column, the default size is two kilobytes. The maximum valid size is 32,739 bytes, but the maximum row size in a database table is limited to 32 kilobytes. (In addition, no more than 195 columns in the same database table can be of varying-length data types, named or unnamed ROW data types, collection data types, or simple large object data types, regardless of the declared size of individual columns.)

If you attempt to insert more than the declared maximum size into an LVARCHAR column, the result depends on the data type of the data:

- If the value comes from a built-in type (such as CHAR or VARCHAR), the database server truncates the data to the declared column size.
- The database server does *not* truncate data strings that come from an **mi_lvarchar** structure, but the database server does return an error.

Tip: If you need to store more than 32,739 bytes of text data in a database of the local database server, use the CLOB data type. The CLOB data type allows you to store the text data outside the database table, in an sbspace. For more information, see Chapter 6, "Using Smart Large Objects," on page 6-1.

Character Data in Binary Mode of a Query

When the database server processes a query, it might handle character data in the following cases:

- Character data that is passed as an input parameter to an SQL statement
- Character data that an SQL statement returns (for example, as a column value)

When a query has a control mode of binary, the database server stores character data in an **mi_lvarchar** varying-length structure. For more information on the control modes of a query, see "Control Modes for Query Data" on page 8-8.

Character Data in C UDRs (Server)

You must use the **mi_lvarchar** data type if your UDR expects any of the SQL character data types as an argument or a return value. Within an **MI_DATUM** structure, the routine manager passes character data to and from a C UDR as a pointer to an **mi_lvarchar** varying-length structure. Therefore, a C UDR must handle text data as **mi_lvarchar** values when it receives arguments or returns data of an SQL character data type, as the following table describes.

Handling Character Data	More Information
If the C UDR receives an argument of an SQL character data type, it must declare its corresponding parameter as a pointer to an **mi_lvarchar** data type.	"Handling Character Arguments" on page 13-6
If a C UDR returns a value of an SQL character data type, it must return a pointer to an **mi_lvarchar** data type.	"Returning Character Values" on page 13-13

External Representation of an Opaque Data Type (Server)

The database server stores the external representation of an opaque data type in an **mi_lvarchar** varying-length structure. The external representation is a text representation of the opaque-type data. Therefore, the input and output support functions of an opaque type handle the external representation as an **mi_lvarchar**. For more information, see "Input and Output Support Functions" on page 16-11.

Character Data in a Smart Large Object

You can use a smart large object to store very large amounts of character data. The **MI_LO_HANDLE** data type has a structure, called an LO *handle,* that identifies the location of smart-large-object data in a separate database partition, called an sbspace. For smart-large-object data that is character data, use the SQL CLOB data type. The CLOB data type allows you to store varying-length character data that is potentially larger than 32 kilobytes. The CLOB data type is a predefined opaque type (an opaque data type that Informix defines). For more information, see Chapter 6, "Using Smart Large Objects," on page 6-1.

Character Processing

The DataBlade API library provides the following functions to process character data:

- Transfer functions
- Conversion functions
- Operation functions

Global Language Support

You can use these character-processing functions on NCHAR and NVARCHAR data. You can also use the character processing that the IBM Informix GLS interface provides to handle multibyte characters.

End of Global Language Support

Transferring Character Data (Server)

To transfer character data between different computer architectures, the DataBlade API provides the following functions that handle type alignment.

DataBlade API Function	Description
mi_get_string( )	Copies a character string, converting any difference in alignment on the client computer to that of the server computer
mi_put_string( )	Copies a character string, converting any difference in alignment on the server computer to that of the client computer

The **mi_get_string( )** and **mi_put_string( )** functions are useful in the send and receive support function of an opaque data type that contains character data (such as **mi_string** or **mi_char**). They ensure that character data remains aligned when transferred to and from client applications. For more information, see "Conversion of Opaque-Type Data with Computer-Specific Data Types" on page 16-21.

Converting Character Data

Both the DataBlade API library and the Informix ESQL/C library provide functions that convert between the binary and text representation of values.

DataBlade API Functions for String Conversion: Many DataBlade API functions expect to manipulate character data as an **mi_lvarchar** value. In addition, all SQL character data types are passed into a C UDR as an **mi_lvarchar** value. The DataBlade API provides the following functions to allow for conversion between a text (null-terminated string) representation of character data and its binary (internal) equivalent. The binary representation of character data is a varying-length structure (**mi_lvarchar**) equivalent.

DataBlade API Function	Description
mi_lvarchar_to_string( )	Creates a null-terminated string from the data in a varying-length structure
mi_string_to_lvarchar( )	Creates a varying-length structure to hold a string

The **mi_lvarchar_to_string( )** and **mi_string_to_lvarchar( )** functions are useful for converting between null-terminated strings and varying-length structures (whose data is *not* null-terminated).

─────────────────────── Server Only ───────────────────────

The **mi_lvarchar_to_string( )** and **mi_string_to_lvarchar( )** functions are also useful in the input and output support functions of an opaque data type that contains **mi_lvarchar** values. They allow you to convert a string between its external format (text) and its internal format (**mi_lvarchar**) when transferred to and from client applications. For more information, see "Conversion of Opaque-Type Data Between Text and Binary Representations" on page 16-16.

─────────────────────── End of Server Only ───────────────────────

For more information on the structure of an **mi_lvarchar** value, see "Varying-Length Data Type Structures" on page 2-13.

In addition, the DataBlade API library provides the following functions to convert text representation of values to their binary representations.

Type of String	More Information
Decimal strings	"DataBlade API Functions for Decimal Conversion" on page 3-14
Date strings	"DataBlade API Functions for Date Conversion" on page 4-3
Date and time strings, Interval strings	"DataBlade API Functions for Date-Time or Interval Conversion" on page 4-13

ESQL/C Functions for String Conversion: The Informix ESQL/C function library provides the following functions that facilitate conversion of values in character data types (such as **mi_string** or **mi_char**) to and from some C-language data types.

Function Name	Description
rstod( )	Converts a string to a **double** type
rstoi( )	Converts a null-terminated string to a two-byte integer (**int2**)
rstol( )	Converts a string to a four-byte integer (**int4**)

In addition, the Informix ESQL/C library provides the following functions to convert text representation of values to their binary representation.

Type of String	More Information
INT8 strings	"Converting INT8 Values" on page 3-7
Decimal strings	"ESQL/C Functions for Decimal Conversion" on page 3-15
Date strings	"ESQL/C Functions for Date Conversion" on page 4-4
Date and time strings	"ESQL/C Functions for Date, Time, and Interval Conversion" on page 4-13

Operations on Character Values

The Informix ESQL/C function library provides the following functions to perform operations on null-terminated strings.

Function Name	Description
ldchar( )	Copies a fixed-length string to a null-terminated string
rdownshift( )	Converts all letters to lowercase
rupshift( )	Converts all letters to uppercase
stcat( )	Concatenates one null-terminated string to another
stchar( )	Copies a null-terminated string to a fixed-length string
stcmpr( )	Compares two null-terminated strings
stcopy( )	Copies one null-terminated string to another string
stleng( )	Counts the number of bytes in a null-terminated string

Character Type Information

The DataBlade API provides functions to obtain the following information about a character (CHAR, VARCHAR, and IDSSECURITYLABEL) data type:

• The data type: its type name (string), type descriptor, or type identifier

- The precision: the maximum number of characters in the data type

The DataBlade API provides the following functions to obtain the type and precision of a character data type.

	DataBlade API Functions	
Source	Type Name, Type Identifier, or Type Descriptor	Precision
For a basic data type	**mi_type_typedesc( )**, **mi_type_typename( )**	**mi_type_precision( )**
For a UDR argument	**mi_fp_argtype( )**, **mi_fp_setargtype( )**	**mi_fp_argprec( )**, **mi_fp_setargprec( )**
For a UDR return value	**mi_fp_rettype( )**, **mi_fp_setrettype( )**	**mi_fp_retprec( )**, **mi_fp_setretprec( )**
For a column	**mi_column_type_id( )**, **mi_column_typedesc( )**	**mi_column_precision( )**
For an input parameter in a prepared statement	**mi_parameter_type_id( )**, **mi_parameter_type_name( )**	**mi_parameter_precision( )**

Varying-Length Data Type Structures

A *varying-length data type structure* can hold data whose length varies from one instance to the next. The database server uses varying-length structures extensively to manage data transfer for DataBlade API modules.

This section provides the following information about varying-length data type structures:
- How to use a varying-length structure
- How to manage memory for a varying-length structure
- How to access data in a varying-length structure

Using a Varying-Length Structure

The DataBlade API provides the following data types to support varying-length data.

DataBlade API Data Type	SQL Varying-Length Data Type	More Information
mi_lvarchar	LVARCHAR	"The mi_lvarchar Data Type" on page 2-9 "Input and Output Support Functions" on page 16-11
mi_bitvarying	BITVARYING	"The mi_bitvarying Data Type" on page 2-28
mi_sendrecv	SENDRECV	"Send and Receive Support Functions" on page 16-17
mi_impexp	IMPEXP	"External Unload Representation" on page 16-22
mi_impexpbin	IMPEXPBIN	"Internal Unload Representation" on page 16-29

All these DataBlade API data types have the same underlying structure. For more information about the structure of a varying-length data type, see "Creating a

Varying-Length Structure" on page 2-14.

┌──────────────── **Informix SE** ────────────────┐

These varying-length data types (**mi_lvarchar**, **mi_bitvarying**, **mi_sendrecv**, **mi_impexp**, **mi_impexpbin**, and varying-length opaque types) *cannot* fit into an **MI_DATUM** structure. Therefore, they must be passed by reference to and from C UDRs.

└──────────────── **End of Informix SE** ────────────────┘

┌──────────────── **Client Only** ────────────────┐

All data types, including **mi_lvarchar**, must be passed by reference within client LIBMI applications.

└──────────────── **End of Client Only** ────────────────┘

Managing Memory for a Varying-Length Structure

The following table summarizes the memory operations for a varying-length structure.

Memory Duration	Memory Operation	Function Name
Current memory duration	Constructor	**mi_new_var( )**, **mi_streamread_lvarchar( )**, **mi_string_to_lvarchar( )**, **mi_var_copy( )**
	Destructor	**mi_var_free( )**

This section describes the DataBlade API functions that allocate and deallocate a varying-length structure.

Important: Do not use either the DataBlade API memory-management functions (such as **mi_alloc( )** and **mi_free( )**) or the operating-system memory-management functions (such as **malloc( )** and **free( )**) to handle allocation of varying-length structures.

Creating a Varying-Length Structure

Table 2-2 lists the DataBlade API functions that create a varying-length structure. These functions are constructor functions for a varying-length structure.

Table 2-2. DataBlade API Allocation Functions for Varying-Length Structures

Accessor Function Name	Description
mi_new_var( )	Creates a new varying-length structure with a data portion of the specified size
mi_streamread_lvarchar( )	Reads a varying-length structure (**mi_lvarchar**) value from a stream and copies the value to a buffer
mi_string_to_lvarchar( )	Creates a new varying-length structure and puts the specified null-terminated string into the data portion The data does not contain a null terminator once it is copied to the data portion.

Accessor Function Name	Description
mi_var_copy( )	Allocates and creates a copy of an existing varying-length structure
	The copy contains its own data portion with the same varying-length data as the original varying-length structure.

The varying-length structure is not contiguous. The allocation functions in Table 2-2 allocate this structure in two parts:

- The varying-length *descriptor* is a fixed-length structure that stores the metadata for the varying-length data.

  The allocation functions allocate the varying-length descriptor and set the data length and the data pointer in this descriptor.

- The *data portion* contains the actual varying-length data.

  The allocation functions allocate the data portion with the length that is specified in the varying-length descriptor. They then set the data pointer in the varying-length descriptor to point to this data portion.

Important: The varying-length data itself resides in a separate structure; it does not actually reside in the varying-length descriptor.

For example, suppose you call the **mi_new_var( )** function that Figure 2-4 shows.

```
mi_lvarchar *new_lvarch;
...
new_lvarch = mi_new_var(200);
```

Figure 2-4. A Sample mi_new_var( ) Call

Figure 2-5 shows the varying-length structure that this **mi_new_var( )** call allocates. This structure consists of both a descriptor and a data portion of 200 bytes. The **mi_new_var( )** function returns a pointer to this structure, which the code in Figure 2-4 assigns to the **new_lvarch** variable.
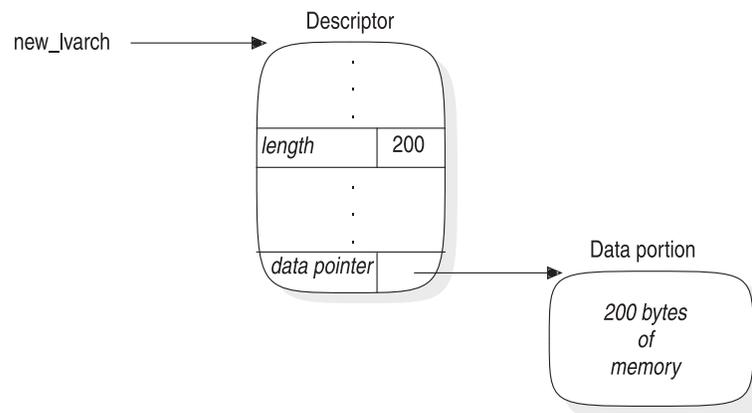

Figure 2-5. Memory Allocated for a Varying-Length Structure

The allocation functions in Table 2-2 on page 2-14 allocate the varying-length structure with the current memory duration. By default, the current memory duration is PER_ROUTINE. For PER_ROUTINE memory, the database server automatically deallocates a varying-length structure at the end of the UDR in which it was allocated. If your varying-length structure requires a longer memory duration, call the **mi_switch_mem_duration( )** function *before* the call to one of the allocation functions in Table 2-2.

—————————— End of Server Only ——————————

The allocation functions in Table 2-2 return the newly allocated varying-length structure as a pointer to an **mi_lvarchar** data type. For example, the call to **mi_new_var( )** in Figure 2-4 allocates a new **mi_lvarchar** structure with a data portion of 200 bytes.

To allocate *other* varying-length data types, cast the **mi_lvarchar** pointer that the allocation function returns to the appropriate varying-length data type. For example, the following call to **mi_new_var( )** allocates a new **mi_sendrecv** varying-length structure with a data portion of 30 bytes:

```
mi_sendrecv *new_sndrcv;
...
new_sndrcv = (mi_sendrecv *)mi_new_var(30);
```

This cast is *not* strictly required, but many compilers recommend it and it does improve clarity of purpose.

Deallocating a Varying-Length Structure

A varying-length structure has a default memory duration of the current memory duration. To conserve resources, use the **mi_var_free( )** function to explicitly deallocate the varying-length structure once your DataBlade API module no longer needs it. The **mi_var_free( )** function is the destructor function for a varying-length structure. It frees *both* parts of a varying-length structure: the varying-length descriptor and the data portion.

Important: Do not use the DataBlade API memory-management function **mi_free( )** to deallocate a varying-length structure. The **mi_free( )** function does not deallocate both parts of a varying-length structure.

Use **mi_var_free( )** to deallocate varying-length structures that you have allocated with **mi_new_var( )** or **mi_var_copy( )**. Do *not* use it to deallocate any varying-length structure that the DataBlade API has allocated.

The **mi_var_free( )** function accepts as an argument a pointer to an **mi_lvarchar** value. The following call to **mi_var_free( )** deallocates the **mi_lvarchar** varying-length structure that Figure 2-4 on page 2-15 allocates:

```
mi_var_free(new_lvarch);
```

To deallocate *other* varying-length data types, cast the **mi_lvarchar** argument of **mi_var_free( )** to the appropriate varying-length type, as the following code fragment shows:

```
mi_sendrecv *new_sndrcv;
...
new_sndrcv = (mi_sendrecv *)mi_new_var(30);
...
mi_var_free((mi_lvarchar *)new_sndrcv);
```

This cast is *not* strictly required, but many compilers recommend it and it does improve clarity of purpose.

Accessing a Varying-Length Structure

A varying-length structure contains the following information:
- Private members, which are not revealed to the DataBlade API programmer
- Public members, which you can access with DataBlade API functions

After you allocate a varying-length structure, you can access the public members of this structure with the DataBlade API accessor functions in Table 2-3.

Table 2-3. Varying-Length Accessor Functions

Accessor Function Name	Description
mi_get_varlen( )	Obtains from the varying-length descriptor the length of the varying-length data
mi_get_vardata( )	Obtains from the varying-length descriptor the data pointer to the data contained in the data portion
mi_get_vardata_align( )	Obtains from the varying-length descriptor the data pointer to the data contained in the data portion, adjusting for any initial padding required to align the data on a specified byte boundary
mi_set_varlen( )	Sets the length of the varying-length data in the varying-length descriptor
mi_set_vardata( )	Sets the data in the data portion of the varying-length structure
mi_set_vardata_align( )	Sets the data in the data portion of the varying-length structure, adding any initial padding required to align the data on a specified byte boundary
mi_set_varptr( )	Sets the data pointer in the varying-length descriptor to the location of a data portion that you allocate

Important: To a DataBlade API module, the varying-length structure is an opaque C data structure. Do not access its internal fields directly. The internal structure of the varying-length structure may change in future releases. Therefore, to create portable code, always use the accessor functions for this structure to obtain and store values.

Varying-Length Data and Null Termination

When you work with varying-length data, keep the following restrictions in mind:
- Do *not* assume that the data in a varying-length structure is null-terminated.
- Do *not* assume that you can use any DataBlade API functions or system calls that operate on a null-terminated string to operate on varying-length data.

Instead, always use the data length (which you can obtain with the **mi_get_varlen( )** function) for all operations on varying-length data.

The varying-length accessor functions in Table 2-3 on page 2-17 do *not* automatically interpret a null-terminator character. Instead, they transfer the number of bytes that the data length in the varying-length descriptor specifies, as follows:

- The **mi_set_vardata( )** and **mi_set_vardata_align( )** functions copy the number of bytes that the data length specifies from their string argument to a varying-length structure.

  For more information, see "Storing a Null-Terminated String" on page 2-20.

- The **mi_get_vardata( )** and **mi_get_vardata_align( )** functions obtain the data pointer from the varying-length descriptor. Use the data length to move through the varying-length data.

  For more information, see "Obtaining the Data Pointer" on page 2-26.

To convert between null-terminated strings and an **mi_lvarchar** structure, use the **mi_string_to_lvarchar( )** and **mi_lvarchar_to_string( )** functions. For more information, see "DataBlade API Functions for String Conversion" on page 2-11.

Storage of Varying-Length Data

This section provides the following information about how to store varying-length data:

- How to store data in a varying-length structure
- How to store a null-terminated string in a varying-length structure
- How to set the data pointer of a varying-length structure

Storing Data in a Varying-Length Structure: The **mi_set_vardata( )** and **mi_set_vardata_align( )** functions copy data into an *existing* data portion of a varying-length structure. These functions assume that the data portion is large enough to hold the data being copied. The code fragment in Figure 2-6 uses **mi_set_vardata( )** to store data in the existing data portion of the varying-length structure that **new_lvarch** references.

```
#define TEXT_LENGTH 200
...

mi_lvarchar *new_lvarch;
mi_char *local_var;
...
/* Allocate a new varying-length structure with a 200-byte
 * data portion
 */
new_lvarch = mi_new_var(TEXT_LENGTH);

/* Allocate memory for null-terminated string */
local_var = (char *)mi_alloc(TEXT_LENGTH + 1);

/* Create the varying-length data to store */
sprintf(local_var, "%s %s %s", "A varying-length structure ",
   "stores data in a data portion, which is separate from ",
   "the varying-length structure.");

/* Update the data length to reflect the string length */
mi_set_varlen(new_lvarch, stleng(local_var));

/* Store the varying-length data in the varying-length
 * structure that new_lvarch references
 */
mi_set_vardata(new_lvarch, local_var);
```

Figure 2-6. Storing Data in Existing Data Portion of a Varying-Length Structure

In Figure 2-6, the call to **mi_new_var( )** creates a new varying-length structure and sets the length field to 200. This call also allocates the 200-byte data portion (see Figure 2-5 on page 2-15).

Figure 2-7 shows the format of the varying-length structure that **new_lvarch** references after the call to **mi_set_vardata( )** successfully completes.
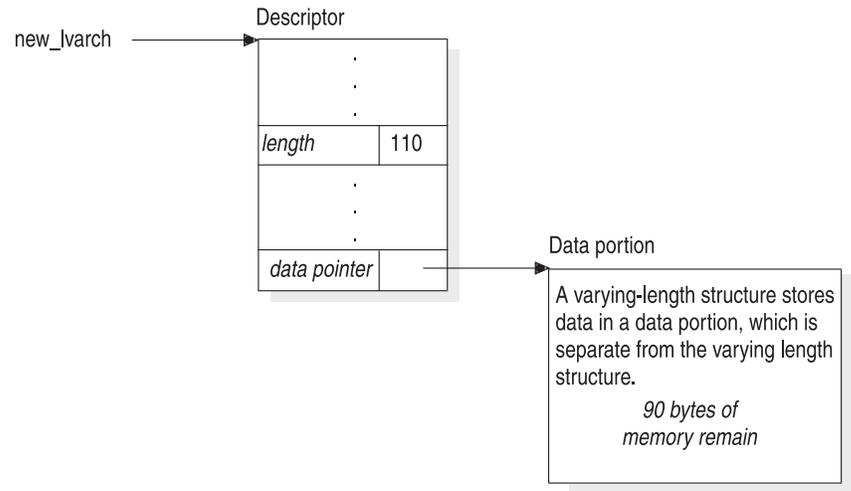

Figure 2-7. Format of a Varying-Length Structure

The **mi_set_vardata( )** function copies from the **local_var** buffer the number of bytes that the data length specifies. Your code must ensure that the data-length field contains the number of bytes you want to copy. In the code fragment in Figure 2-6 on page 2-18, the data-length field was last set by the call to **mi_set_varlen( )** to 110 bytes. However, if the **mi_set_varlen( )** function executed *after* the **mi_set_vardata( )** call, the data length would still have been 200 bytes (set by **mi_new_var( )**). In this case, **mi_set_vardata( )** would try to copy 200 bytes starting at the location of the **local_var** variable. Because the actual **local_var** data only occupies 110 bytes of memory, 90 unused bytes remain in the data portion.

The **mi_set_vardata( )** function aligns the data that it copies on four-byte boundaries. If this alignment is not appropriate for your varying-length data, use the **mi_set_vardata_align( )** function to store data on a byte boundary that you specify. For example, the following call to **mi_set_vardata_align( )** copies data into the **var_struc** varying-length structure and aligns this data on eight-byte boundaries:

```
char *buff;
mi_lvarchar *var_struc;
...
mi_set_vardata_align(var_struc, buff, 8);
```

You can determine the alignment of a data type from its type descriptor with the **mi_type_align( )** function.

Tip: You can also store data in a varying-length structure through the data pointer that you obtain with the **mi_get_vardata( )** or **mi_get_vardata_align( )** function. For more information, see "Obtaining the Data Pointer" on page 2-26.

The **mi_set_vardata_align( )** function copies the number of bytes that the data-length field specifies.

Storing a Null-Terminated String: The **mi_string_to_lvarchar( )** function copies a null-terminated string into a varying-length structure that it creates. This function performs the following steps:

1. Allocates a new varying-length structure

   The **mi_string_to_lvarchar( )** function allocates the varying-length descriptor, setting the data length and data pointer appropriately. Both the data length and the size of the data portion are the length of the null-terminated string *without* its null terminator.

   ┌─────────────────────── **Server Only** ───────────────────────┐

   The **mi_string_to_lvarchar( )** function allocates the varying-length structure that it creates with the current memory duration.

   └─────────────────────── **End of Server Only** ───────────────────────┘

2. Copies the data of the null-terminated string into the newly allocated data portion

   The **mi_string_to_lvarchar( )** function does *not* copy the null terminator of the string.

3. Returns a pointer to the newly allocated varying-length structure

The following code fragment uses **mi_string_to_lvarchar( )** to store a null-terminated string in the data portion of a new varying-length structure:

```
char *local_var;
mi_lvarchar *lvarch;
...
/* Allocate memory for null-terminated string */
local_var = (char *)mi_alloc(200);

/* Create the varying-length data to store */
sprintf(local_var, "%s %s %s", "A varying-length structure ",
   "stores data in a data portion, which is separate from ",
   "the varying-length structure.");

/* Store the null-terminated string as varying-length data */
lvarch = mi_string_to_lvarchar(local_var);
```

Figure 2-8 shows the format of the varying-length structure that **lvarch** references after the preceding call to **mi_string_to_lvarchar( )** successfully completes.
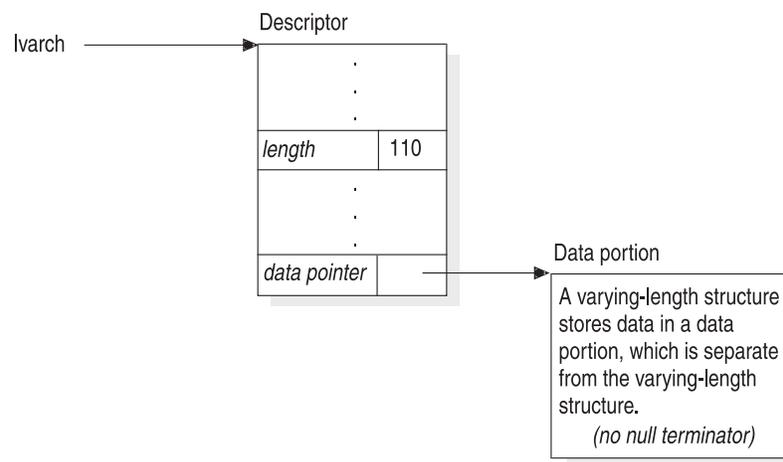

Figure 2-8. Copying a Null-Terminated String into a Varying-Length Structure

The **lvarch** varying-length structure in Figure 2-8 has a data length of 110. The null terminator is *not* included in the data length because the **mi_string_to_lvarchar( )** function does *not* copy the null terminator into the data portion.

If your DataBlade API module needs to store a null terminator as part of the varying-length data, you can take the following steps:

1. Increment the data length accordingly and save it in the varying-length descriptor with the **mi_set_varlen( )** function.
2. Copy the data, including the null terminator, into the varying-length structure with the **mi_set_vardata( )** or **mi_set_vardata_align( )** function.

   These functions copy in the null terminator because the data length includes the null-terminator byte in its count. These functions assume that the data portion is large enough to hold the string and any null terminator.

After you perform these steps, you can obtain the null terminator as part of the varying-length data.

Important: If you choose to store null terminators as part of your varying-length data, your code must keep track that this data is null-terminated. The DataBlade API functions that handle varying-length structures do not track the presence of a null terminator.

The following code fragment stores a string plus a null terminator in the varying-length structure that **lvarch** references:

```
#define TEXT_LENGTH 200
...

mi_lvarchar *lvarch;
char *var_text;
mi_integer var_len;
...
/* Allocate memory for null-terminated string */
var_text = (char *)mi_alloc(TEXT_LENGTH);

/* Create the varying-length data to store */
sprintf(var_text, "%s %s %s", "A varying-length structure ",
   "stores data in a data portion, which is separate from ",
   "the varying-length structure.");
var_len = stleng(var_text) + 1;

/* Allocate a varying-length structure to hold the
 * null-terminated string (with its null terminator)
 */
lvarch = mi_new_var(var_len);

/* Copy the number of bytes that the data length specifies
 * (which includes the null terminator) into the
 * varying-length structure
 */
mi_set_vardata(lvarch, var_text);
```

Figure 2-9 shows the format of this varying-length structure after the preceding call to **mi_set_vardata( )** successfully completes.

Descriptor

lvarch

length   111

data pointer

Data portion

A varying-length structure
stores data in a data
portion, which is separate
from the varying length
structure.
                (null terminated)
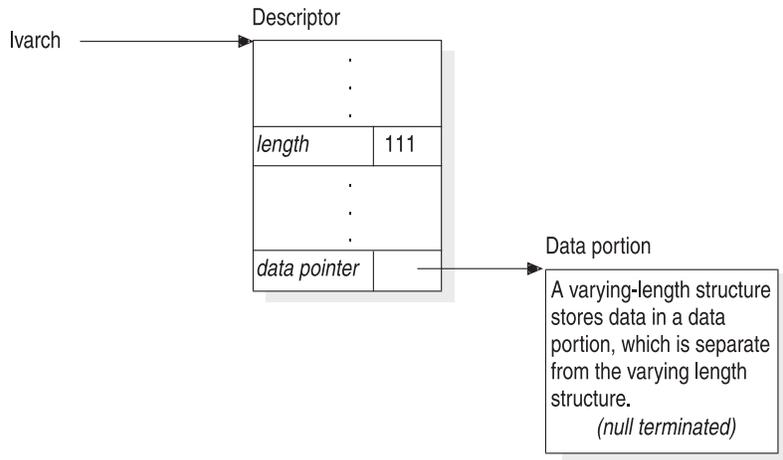
Figure 2-9. Copying a Null-Terminated String into a Varying-Length Structure

Setting the Data Pointer: The **mi_set_varptr( )** function enables you to set the data pointer in a varying-length structure to memory that you allocate. The following code fragment creates an *empty* varying-length structure, which is a varying-length structure that has no data portion allocated:

```
#define VAR_MEM_SIZE 20
...
mi_lvarchar *new_lvarch;
char *var_text;
mi_integer var_len;
...
/* Allocate PER_COMMAND memory for varying-length data */
var_text = (char *)mi_dalloc(VAR_MEM_SIZE, PER_COMMAND);

/* Allocate an empty varying-length structure */
(void)mi_switch_mem_duration(PER_COMMAND);
new_lvarch = mi_new_var(0);

/* Store the varying-length data in the var_text buffer
 * with the fill_buffer( ) function (which you have coded).
 * This function returns the actual length of the nonnull-
 * terminated string. It does NOT put a null terminator at
 * the end of the data.
 */
var_len = fill_buffer(var_text);
```

Figure 2-10 shows the format of the varying-length structure that **new_lvarch** references after the **fill_buffer( )** function successfully completes.

Descriptor

new_lvarch ⟶

. . .	
length	0
. . .	
data pointer	NULL

var_text ⟶

20 bytes of
PER_COMMAND memory
(no null terminator at end of data)
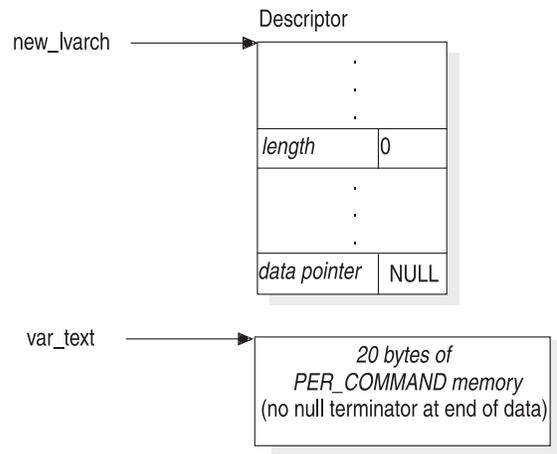
Figure 2-10. Empty Varying-Length Structure

The varying-length structure in Figure 2-10 is empty because it has the following characteristics:

- Data length of zero (0)
- NULL-valued pointer as its data pointer

After you have an empty varying-length structure, you can use the **mi_set_varptr( )** function to set the data pointer to the PER_COMMAND memory duration, as the following code fragment shows:

```
/* Set the length of the new varying-length data */
mi_set_varlen(new_lvarch, VAR_MEM_SIZE);

/* Set the pointer to the data portion of the
 * varying-length structure to the PER_COMMAND memory
 * that 'var_text' references.
 */
mi_set_varptr(new_lvarch, var_text);
```

The preceding call to **mi_set_varlen( )** updates the length in the varying-length structure to the length of 20 bytes. Figure 2-11 shows the format of the varying-length structure that **new_lvarch** references after the preceding call to **mi_set_varptr( )** successfully completes.

Descriptor

new_lvarch ⟶

. . .	
length	20
. . .	
data pointer	

Data portion

20 bytes of
PER_COMMAND memory
(no null terminator at
end of data)

var_text ⟶
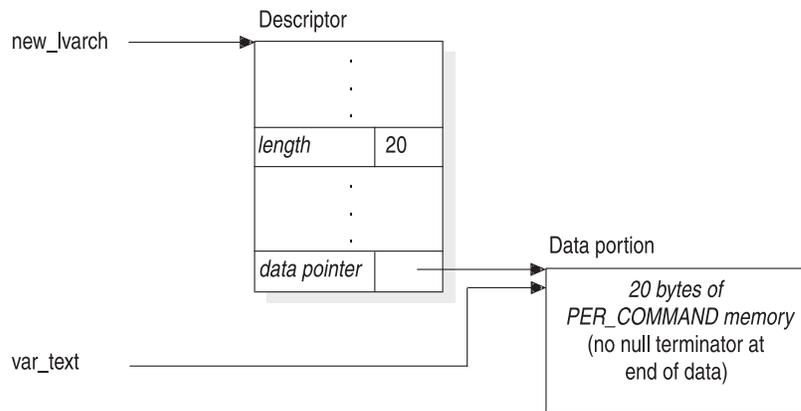
*Figure 2-11. Setting the Data-Portion Pointer in a Varying-Length Structure*

Make sure that you allocate the data-portion buffer with a memory duration appropriate to the use of the data portion.

For more information in memory allocation, see Chapter 14, "Managing Memory," on page 14-1.

Information About Varying-Length Data

Use the following DataBlade API accessor functions to obtain information about varying-length data from a varying-length structure.

Varying-Length Information	DataBlade API Accessor Function
Length of varying-length data	**mi_get_varlen( )**
Data portion	**mi_lvarchar_to_string( )**, **mi_var_to_buffer( )**, **mi_var_copy( )**
Data pointer	**mi_get_vardata( )**, **mi_get_vardata_align( )**

Obtaining the Data Length: The **mi_get_varlen( )** function returns the data length from a varying-length descriptor. Keep in mind the following restrictions about data length:

- Do *not* assume that the data in a varying-length structure is null-terminated.

  Always use the data length to determine the end of the varying-length data when you perform operations on this data.

- When you increase the length of the data with **mi_set_varlen( )**, this function does *not* automatically increase the amount of memory allocated to the data portion.

  You must ensure that there is sufficient space in the data portion to hold the varying-length data. If there is insufficient space, allocate a new data portion with a DataBlade API memory-management function (such as **mi_dalloc( )**) and assign a pointer to this new memory to the data pointer of your varying-length structure.

For the varying-length structure in Figure 2-5 on page 2-15, a call to **mi_get_varlen( )** returns 200. For the varying-length structure that Figure 2-7 on page 2-19 shows, a call to **mi_get_varlen( )** returns 110.

Obtaining Data as a Null-Terminated String: The **mi_lvarchar_to_string( )** function obtains the data from a varying-length structure and converts it to a null-terminated string. This function performs the following steps:

1. Allocates a new buffer to hold the null-terminated string

The **mi_lvarchar_to_string( )** function allocates the string that it creates with the current memory duration.

2. Copies the data in the data portion of the varying-length structure to the newly allocated buffer

The **mi_lvarchar_to_string( )** function automatically copies the number of bytes that the data length in the varying-length descriptor specifies. It then appends a null terminator to the string.

3. Returns a pointer to the newly allocated null-terminated string

Suppose you have the varying-length structure that Figure 2-8 on page 2-20 shows. The following code fragment uses the **mi_lvarchar_to_string( )** function to obtain this varying-length data as a null-terminated string:

```
mi_lvarchar *lvarch;
char *var_str;
...
var_str = mi_lvarchar_to_string(lvarch);
```

The code fragment does not need to allocate memory for the **var_str** string because the **mi_lvarchar_to_string( )** function allocates memory for the new string. After the call to **mi_lvarchar_to_string( )** completes successfully, the **var_str** variable contains the following null-terminated string:

```
A varying-length structure stores data in a data portion, which is separate
from the varying-length structure.
```

Copying Data into a User-Allocated Buffer: The **mi_var_to_buffer( )** function copies the data of an existing varying-length structure into a user-allocated buffer. The function copies data up to the data length specified in the varying-length descriptor. You can obtain the current data length with the **mi_get_varlen( )** function.

The following code fragment copies the contents of the varying-length structure in Figure 2-8 on page 2-20 into the **my_buffer** user-allocated buffer:

```
mi_lvarchar *lvarch;
char *my_buffer;
...
my_buffer = (char *)mi_alloc(mi_get_varlen(lvarch));
mi_var_to_buffer(lvarch, my_buffer);
```

After the successful completion of **mi_var_to_buffer( )**, the **my_buffer** variable points to the following string, which is *not* null terminated:

```
A varying-length structure stores data in a data portion, which is separate
from the varying-length structure.
```

Important: Do not assume that the data in the user-allocated buffer is null terminated. The **mi_var_to_buffer( )** function does not append a null terminator to the data in the character buffer.

Copying Data into a New Varying-Length Structure: The **mi_var_copy( )** function copies data from an existing varying-length structure into a new varying-length structure. This function performs the following steps:

1. Allocates a new varying-length structure

   For the new varying-length structure, the **mi_var_copy( )** function allocates a data portion whose size is that of the data in the existing varying-length structure.

   ──────────────────── Server Only ────────────────────
   The **mi_var_copy( )** function allocates the varying-length structure that it creates with the current memory duration.
   ──────────────────── End of Server Only ────────────────────

2. Copies the data in the data portion of the existing varying-length structure to the data portion of the newly allocated varying-length structure

   The **mi_var_copy( )** function automatically copies the number of bytes that the data length in the existing varying-length descriptor specifies.

3. Returns a pointer to the newly allocated varying-length structure as a pointer to an **mi_lvarchar** value

Suppose you have the varying-length structure that Figure 2-8 on page 2-20 shows. The following code fragment uses the **mi_var_copy( )** function to create a copy of this varying-length structure:

```
mi_lvarchar *lvarch, *lvarch_copy;
...
lvarch_copy = mi_var_copy(lvarch);
```

After the call to **mi_var_copy( )** completes successfully, the **lvarch_copy** variable points to a new varying-length structure, as Figure 2-12 shows. The varying-length structure that **lvarch_copy** references is a completely separate structure from the structure that **lvarch** references.
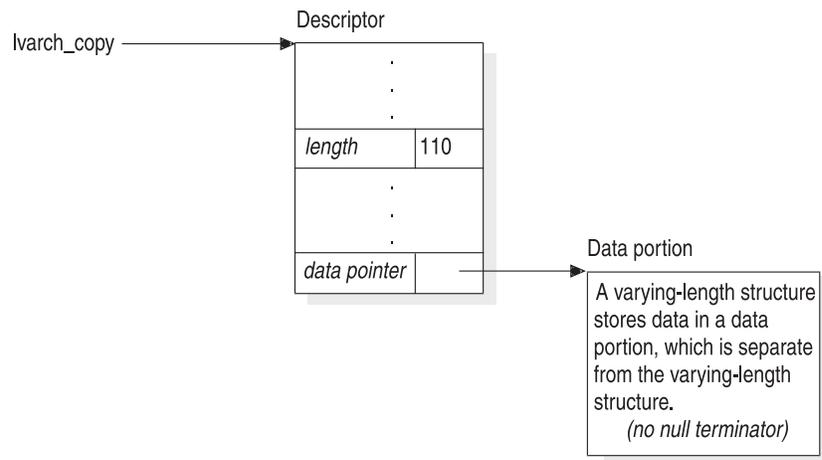

Figure 2-12. Copying a Varying-Length Structure

Obtaining the Data Pointer: The **mi_get_vardata( )** and **mi_get_vardata_align( )** functions obtain the actual data pointer from the varying-length descriptor. Through this data pointer, you can directly access the varying-length data.

The following code fragment uses the **mi_get_vardata( )** function to obtain the data pointer from the varying-length structure in Figure 2-7 on page 2-19:

```
mi_lvarchar *new_lvarch;
char *var_ptr;
...
/* Get the data pointer of the varying-length structure */
var_ptr = mi_get_vardata(new_lvarch);
```

Figure 2-13 shows the format of the varying-length structure that **new_lvarch** references after the preceding call to **mi_get_vardata( )** successfully completes.
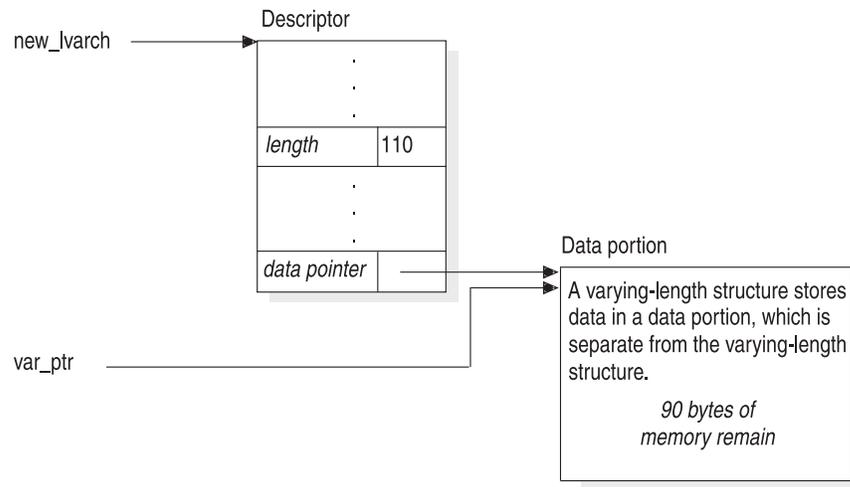
Figure 2-13. Getting the Data Pointer from a Varying-Length Structure

You can then access the data through the **var_ptr** data pointer, as the following code fragment shows:

```
mi_lvarchar *new_lvarch;
mi_integer var_len, i;
mi_char one_char;
mi_char *var_ptr;

var_ptr = mi_get_vardata(new_lvarch);
var_len = mi_get_varlen(new_lvarch);
for ( i=0; i<var_len; i++ )
   {
   one_char = var_ptr[i];
   /* process the character as needed */
   ...
   }
```

────────────────────────────── **Server Only** ──────────────────────────────

The database server passes text data to a UDR as an **mi_lvarchar** structure. Figure 13-3 on page 13-8 shows the implementation of a user-defined function named **initial_cap( )**, which ensures that the first letter of a character string is uppercase and that subsequent letters are lowercase.

The **initial_cap( )** function uses **mi_get_vardata( )** to obtain each character from the data portion of the varying-length structure. This data portion contains the character value that the function receives as an argument. The function checks each letter to ensure that it has the correct case. If the case is incorrect, **initial_cap( )** uses the data pointer to update the appropriate letter. The function then returns a new **mi_lvarchar** structure that holds the result. For more information, see "Handling Character Arguments" on page 13-6.

────────────────────────────── **End of Server Only** ──────────────────────────────

The varying-length structure aligns data on four-byte boundaries. If this alignment is not appropriate for your varying-length data, use the **mi_get_vardata_align( )** function to obtain the data aligned on a byte boundary that you specify. You can determine the alignment of a data type from its type descriptor with the **mi_type_align( )** function.

Tip: When you obtain aligned data from a varying-length structure that is associated with an extended data type, specify an alignment value to **mi_get_vardata_align( )** that is appropriate for the extended data type. For more information, see "Specifying the Memory Alignment of an Opaque Type" on page 16-6.

The **mi_get_vardata_align( )** function obtains the number of bytes that the data-length field specifies.

Byte Data Types

The DataBlade API supports the following data types that can hold byte data in a DataBlade API module.

DataBlade API Character Data Type	Description	SQL Character Data Type
mi_bitvarying	Varying-length structure to hold varying-length byte data	None
MI_LO_HANDLE	LO handle to identify a smart large object that holds byte data	BLOB

Tip: The database server also supports the BYTE data type for byte data. It stores BYTE data as a simple large object. However, the DataBlade API does not directly support simple large objects. For more information, see "Simple Large Objects" on page 2-32.

The mi_bitvarying Data Type

The SQL BITVARYING data type stores variable-length byte data that is potentially larger than 255 bytes. The BITVARYING data type is a predefined opaque type (an opaque data type that Informix defines). The DataBlade API supports the BITVARYING data type with the **mi_bitvarying** data type, which the DataBlade API implements as a varying-length structure.

Tip: The SQL data type BITVARYING and the DataBlade API data type **mi_bitvarying** are not exactly the same. Although you use the **mi_bitvarying** varying-length structure to hold BITVARYING data, you can also use a varying-length structure for other varying-length data.

For a BITVARYING column, the maximum size of the data is two kilobytes. This limitation is *not* inherent to the BITVARYING data type; however, the maximum row size in a database table is 32 kilobytes. If a BITVARYING column were to use the full supported size of 32 kilobytes, the table could contain only one column: a single BITVARYING column.

Tip: If you need to store more than two kilobytes of byte data, use the BLOB data type. The BLOB data type enables you to store the byte data outside the database table in an sbspace. For more information, see Chapter 6, "Using Smart Large Objects," on page 6-1.

You can use an **mi_bitvarying** varying-length structure to store large amounts of byte data. For more information, see "Varying-Length Data Type Structures" on page 2-13.

The routine manager uses an **mi_bitvarying** structure to hold data for an argument or return value of a C UDR when this data is a varying-length opaque type. For more information, see "Determining the Passing Mechanism for an Opaque Type" on page 16-7.

You must use the **mi_bitvarying** data type if your UDR expects any varying-length data type as an argument or a return value. Within an **MI_DATUM** structure, the routine manager passes varying-length opaque-type data to and from a C UDR as a pointer to an **mi_bitvarying** varying-length structure. Therefore, a C UDR must handle this data as **mi_bitvarying** values when it receives arguments or returns data of a varying-length opaque data type, as the following table describes.

Handling Character Data	More Information
If the C UDR receives an argument of a varying-length opaque data type, it must declare its corresponding parameter as a pointer to an **mi_bitvarying** data type.	"Handling Varying-Length Opaque-Type Arguments" on page 13-10
If a C UDR returns a value of a varying-length opaque data type, it must return a pointer to an **mi_bitvarying** data type.	"Returning Opaque-Type Values" on page 13-14

Byte Data in a Smart Large Object

You can use a smart large object to store very large amounts of byte data. The **MI_LO_HANDLE** data type holds a structure, called an LO *handle*, that identifies the location of smart-large-object data in a separate database partition, called an sbspace. For smart-large-object data that is byte data, use the SQL BLOB data type. The BLOB data type allows you to store varying-length byte data of up to four terabytes. The BLOB data type is a predefined opaque type (an opaque data type that Informix defines). For more information, see Chapter 6, "Using Smart Large Objects," on page 6-1.

Byte Processing

The DataBlade API provides the following support for byte data:
- Informix ESQL/C functions that operate on byte data
- DataBlade API functions that transfer byte data

Manipulating Byte Data

The DataBlade API supports the following byte functions from the Informix ESQL/C library to perform operations on byte data.

Function Name	Description
bycmpr( )	Compares two groups of contiguous bytes
bycopy( )	Copies bytes from one area to another
byfill( )	Fills the specified area with a character
byleng( )	Counts the number of bytes in a string

Transferring Byte Data (Server)

To transfer byte data between different computer architectures, the DataBlade API provides the following functions that handle type alignment and byte order.

DataBlade API Function	Description
mi_get_bytes( )	Copies an aligned number of bytes, converting any difference in alignment or byte order on the client computer to that of the server computer
mi_put_bytes( )	Copies an aligned number of bytes, converting any difference in alignment or byte order on the server computer to that of the client computer

The **mi_get_bytes( )** and **mi_put_bytes( )** functions are useful in the send and receive support function of an opaque data type that contains uninterpreted bytes. They ensure that byte data remain aligned when transferred to and from client applications. For more information, see "Conversion of Opaque-Type Data Between Text and Binary Representations" on page 16-16.

Boolean Data Types

Boolean data holds values to indicate two states: true and false. The DataBlade API provides support for boolean values in both their text and binary representations.

Boolean Text Representation

The DataBlade API supports a Boolean value in text representation as a character enclosed in single quotation marks, with the format that Table 2-4 shows.

Table 2-4. Text Representation of Boolean Data

Boolean Value	Text Representation
True	't' or 'T'
False	'f' or 'F'

A Boolean value in its text representation is often called a *Boolean string*.

Boolean Binary Representation

The SQL BOOLEAN data type holds the internal (binary) format of a Boolean value. This value is a single-byte representation of Boolean data, as the following table shows.

Boolean Value	Binary Representation
True	\1
False	\0

The BOOLEAN data type is a predefined opaque type (an opaque data type that Informix defines). Its external format is the Boolean text representation that Table 2-4 shows. Its internal format consists of the values that the preceding table shows. For a complete description of the SQL BOOLEAN data type, see the *IBM Informix Guide to SQL: Reference*.

Tip: The internal format of the BOOLEAN data type is often referred to as its binary representation.

The DataBlade API supports the SQL BOOLEAN data type with the **mi_boolean** data type. Therefore, the **mi_boolean** data type also holds the binary representation of a Boolean value.

Server Only

An **mi_boolean** value is one byte on *all* computer architectures; therefore, it can fit into an **MI_DATUM** structure. You can pass **mi_boolean** data by value in C UDRs.

End of Server Only

Client Only

In client LIBMI applications, you must pass all data by reference, including **mi_boolean** values.

End of Client Only

Windows Only

Because an **mi_boolean** value is smaller than the size of an **MI_DATUM** structure, the DataBlade API cast promotes the value to the size of **MI_DATUM** when you copy the value into an **MI_DATUM** structure. When you obtain the **mi_boolean** value from an **MI_DATUM** structure, you need to reverse the cast promotion to ensure that your value is correct.

```
MI_DATUM datum;
mi_boolean bool_val;
...
bool_val = (char) datum;
```

Alternatively, you can declare an **mi_integer** value to hold the Boolean value.

End of Windows Only

Pointer Data Types (Server)

The SQL POINTER data type is the SQL equivalent of a generic pointer. This data type is used in the routine registration of a UDR to indicate that some data type has no equivalent SQL data type. The DataBlade API represents the POINTER data type with the **mi_pointer** data type.

Use the **mi_pointer** data type only for communications between UDRs. The POINTER data type is a predefined opaque type (an opaque data type that Informix defines). However, no opaque-type support functions for this data type are included.

Important: Because the POINTER data type does not include opaque-type support functions, you cannot pass this type between the database server and a client application. Also, do not define columns to be of type POINTER.

The **mi_pointer** data type is guaranteed to be the size of the C type **void *** on *all* computer architectures. The C type **void *** is usually equivalent to a **long** type,

which is usually four bytes in length.

<div style="border:1px solid black; padding:10px;">

64-bit

On 64-bit platforms, **void *** is eight bytes in length, so **mi_pointer** is also eight bytes.

End of 64-bit
</div>

An **mi_pointer** value *can* fit into an **MI_DATUM** structure and can be passed by value to and from C UDRs. Keep in mind that because **mi_pointer** actually contains an address to a value, passing an **mi_pointer** by value is actually the same as passing the value to which **mi_pointer** points by reference.

Important: When you use **mi_pointer**, make sure that the value that the **mi_pointer** references is allocated with a memory duration appropriate to the use of the value. For more information, see "Choosing the Memory Duration" on page 14-4.

Simple Large Objects

The DataBlade API does *not* provide direct support for simple large objects. Therefore, it cannot directly access TEXT and BYTE columns. However, the database server provides the following cast functions between simple and smart large objects.

Type Conversion	SQL Cast Function
From the TEXT data type to the CLOB data type	**TextToClob( )**
From the BYTE data type to the BLOB data type	**ByteToBlob( )**

For more information on these SQL cast functions, see the description of the Expression segment in the *IBM Informix Guide to SQL: Syntax*.

<div style="border:1px solid black; padding:10px;">

Server Only

C UDRs can accept TEXT data as arguments because the database server passes *all* character data in the **mi_lvarchar** data type. For more information, see "Character Data in C UDRs (Server)" on page 2-10.

C UDRs can also accept BYTE data as long as they declare and handle this data as a smart large object. The database server converts the BYTE data to BLOB data when it passes this data to the UDR.

End of Server Only
</div>

The MI_DATUM Data Type

The DataBlade API handles a generic data value as an **MI_DATUM** value, also called a *datum*. A datum is stored in a chunk of memory that can fit into a computer register.

In the C language, the **void *** type is a typeless way to point to any object and should hold any integer value. This type is usually equivalent to the **long int** type and is usually four bytes in length, depending on the computer architecture.

MI_DATUM is defined as a **void *** type. The **MI_DATUM** data type is guaranteed to be the size of the C type **void *** on *all* computer architectures.

64-bit

On 64-bit platforms, **void *** is eight bytes in length, so an **MI_DATUM** value is stored in eight bytes.

End of 64-bit

This section provides the following information about the **MI_DATUM** data type:

- Contents of an **MI_DATUM** structure
- Address calculations with **MI_DATUM** values
- Uses of **MI_DATUM** structures

Contents of an MI_DATUM Structure

A datum in an **MI_DATUM** structure can describe a value of any SQL data type. You can use an **MI_DATUM** structure to transport a value of an SQL data type between the database server and the DataBlade API module.

MI_DATUM in a C UDR (Server)

In a C UDR, the contents of an **MI_DATUM** structure depend on the SQL data type of the value, as follows:

- For *most* data types, the **MI_DATUM** structure contains a pointer to the data type.

  The actual value of most data types is too large to fit within an **MI_DATUM** structure. For such data types, the DataBlade API passes the value using the *pass-by-reference* mechanism. Use the contents of the **MI_DATUM** structure as a pointer to access the actual value.

- For a few small data types, the **MI_DATUM** structure contains the actual data value.

  Table 2-5 shows the few data types whose value can *always* fit in an **MI_DATUM** structure. For these data types, the DataBlade API passes the value using the *pass-by-value* mechanism. Use the contents of the **MI_DATUM** structure as the actual data value.

Table 2-5. Types of Values That Fit in an MI_DATUM Structure (Passed by Value)

DataBlade API Data Types	Length	SQL Data Types
Data types that can hold four-byte integers, including **mi_integer** and **mi_unsigned_integer**	4	The SQL INTEGER data type
mi_date	4	The SQL DATE data type
Data types that can hold two-byte integers, including **mi_smallint** and **mi_unsigned_smallint**	2	The SQL SMALLINT data type
Data types that can hold a one-byte character, including **mi_char1** and **mi_unsigned_char1**	1	The SQL CHAR(1) data type (Multicharacter values must be passed by reference.)
mi_boolean	1	The SQL BOOLEAN data type
mi_pointer	size of (**void ***)	The SQL POINTER data type

Table 2-5. Types of Values That Fit in an MI_DATUM Structure (Passed by Value) (continued)

DataBlade API Data Types	Length	SQL Data Types
C data structure for the internal format of an opaque data type when the structure size can fit into an **MI_DATUM** structure	Depends on the size of the C data structure	An opaque data type whose CREATE OPAQUE TYPE statement specifies the PASSEDBYVALUE modifier

For all data types that Table 2-5 lists, the DataBlade API passes the value in an **MI_DATUM** structure by value unless the variable is declared as pass by reference. For example, in the following sample function signature, the **arg2** variable would be passed by reference to the **my_func( )** UDR because it is declared as a pointer:

```
mi_integer my_func(arg1, arg2)
   mi_integer arg1; /* passed by value */
   mi_integer *arg2; /* passed by reference */
```

Values of data types with sizes smaller than or equal to the size of **void *** can be passed by value because they can fit into an **MI_DATUM** structure. A value smaller than the size of **MI_DATUM** is cast promoted to the **MI_DATUM** size with whatever byte position is appropriate for the computer architecture. When you obtain a smaller passed-by-value value from an **MI_DATUM** structure, you need to reverse the cast promotion to ensure that your value is correct.

For example, an **mi_boolean** value is a one-byte value. To pass it by value, the DataBlade API performs something like the following example when it puts the **mi_boolean** value into an **MI_DATUM** structure:

```
datum = (void *((char) bool))
```

In the preceding cast promotion, *datum* is an **MI_DATUM** structure and *bool* is an **mi_boolean** value.

When you obtain the **mi_boolean** value from the **MI_DATUM** structure, reverse the cast-promotion process with something like the following example:

```
mi_boolean bool_val;
MI_DATUM datum;
...
bool_val = (char) datum;
```

To avoid the cast promotion situation, it is recommended that you declare small pass-by-value SQL types as **mi_integer**.

For all data types *not* listed in Table 2-5, the DataBlade API passes the value in an **MI_DATUM** structure by reference; that is, the **MI_DATUM** structure contains a pointer to the actual data type.

Warning: Do not assume that any data type of length 1, 2, or 4 is passed by value. Not all one-, two-, or four-byte datums are passed by value. For example, the **mi_real** data type is passed by reference. Always check the data type or use the **mi_type_byvalue( )** function to determine the passing mechanism.

UDRs store the data types of their arguments in an **MI_FPARAM** structure. You can check the type identifier of an argument to determine if it is passed by value or by reference, as the following code fragment shows:

```
my_type_id = mi_fp_argtype(my_fparam, 1);
my_type_desc = mi_type_typedesc(conn, my_type_id);
if ( mi_type_byvalue(my_type_desc) == MI_TRUE )
   {
   /* Argument is passed by value: extract one-, two-, or
    * four-byte item from argument
    */
   }
else
   {
   /* Argument is passed by reference: it contains a pointer
    * to the actual value
    */
   }
```

However, a UDR that hardcodes a type identifier in a **switch** or **if** statement to determine actions can handle only built-in data types. It cannot handle all possible user-defined types because not all of them have unique, type-specific identifiers.

MI_DATUM in a Client LIBMI Application

The preceding rules for passing values in **MI_DATUM** structures by reference and by value do *not* apply to client LIBMI applications. In client LIBMI applications, pass values of *all* data types in **MI_DATUM** structures by reference.

Address Calculations with MI_DATUM Values

In performing address calculations with datums, do not use **char** * as the type. This practice can lead to problems. Instead, calculate addresses with the **size_t** data type. To increment a datum by an arbitrary length, use the following equation:

```
void *ptr = (void *)((size_t)datum + (size_t)length )
```

In performing address calculations with an **MI_DATUM** value, it is common practice to use **char** * as an intermediate type because arithmetic operators are not allowed on the **void** * type. The ANSI C standard explicitly says that **void** * and **char** * have the same representation.

For example, the following code increments an **MI_DATUM** value by an arbitrary length:

```
MI_DATUM ptr = (MI_DATUM) ((char *)(datum) + (ptrdiff_t)(length))
```

In the preceding formula, **ptrdiff_t** is defined in the ANSI C header file, **stddef.h**, and is a signed integer data type.

Another addressing scheme follows:

```
void *ptr = ((char *)datum) + length
```

Uses of MI_DATUM Structures

An **MI_DATUM** structure holds a value that is transferred to or from the database server. DataBlade API functions handle **MI_DATUM** structures consistently. The following table lists uses of **MI_DATUM** structures.

Use of MI_DATUM Structures	Description	More Information
Routine arguments for a UDR	When a UDR is called, the routine manager passes UDR arguments as datums. The data type of each argument determines whether the routine manager passes the argument by reference or by value.	"MI_DATUM Arguments" on page 13-3

Use of MI_DATUM Structures	Description	More Information
Return value from a user-defined function	When a user-defined function exits, the routine manager passes the return value as a datum. The return-value data type determines whether the routine manager passes the return value by reference or by value.	"Returning a Value" on page 13-12
OUT parameter from a user-defined function	When a user-defined function sets an OUT parameter, the routine manager passes the parameter back as a datum. The routine manager always passes an OUT parameter by reference.	"Using an OUT Parameter" on page 13-14
Routine arguments for a UDR that you execute with the Fastpath interface	When you execute a UDR with the Fastpath interface, the **mi_routine_exec( )** function passes UDR arguments as datums. The data type of each argument determines whether this function passes the argument by reference or by value.	"Passing in Argument Values" on page 9-27
Return value from a UDR that you execute with the Fastpath interface	When a user-defined function that you execute with the Fastpath interface returns, the **mi_routine_exec( )** function passes the return value as a datum. The return-value data type determines whether this function passes the return value by reference or by value.	"Receiving the Return Value" on page 9-27
Column values returned or inserted in SQL statements	When the **mi_value( )** or **mi_value_by_name( )** function returns a column value for a query in binary representation, it returns this value as a datum. When the **mi_row_create( )** function creates a row structure, it accepts column values as datums.	"Obtaining Column Values" on page 8-42
Element values retrieved or inserted in SQL collections	When the **mi_collection_fetch( )** function fetches an element from a collection, it represents the element as a datum. When the **mi_collection_insert( )** function inserts an element from a collection, it represents the element as a datum.	"Accessing Elements of a Collection" on page 5-6
Input-parameter values in a prepared SQL statement	When the **mi_exec_prepared_statement( )** or **mi_open_prepared_statement( )** function provides input-parameter values, it represents them as datums.	"Assigning Values to Input Parameters" on page 8-27

The NULL Constant

The DataBlade API supports two different uses of a NULL constant:

- The SQL NULL value
- The NULL-valued pointer

Important: The DataBlade API NULL-valued pointer is *not* the same as the SQL NULL value.

SQL NULL Value

The SQL NULL value represents a null or empty value in a database column. The NULL value is distinct from all valid values for a given data type. For example, the INTEGER data type holds a four-byte integer. This four-byte data type can hold 2^{32} (or 4,294,967,296) values:

- zero (0)
- positive values: 1 to 2,147,483,647

- negative values: -1 to -2,147,483,647
- NULL value: 2,147,483,648 (the maximum negative number)

Because the representation of the NULL value is unique to each data type, the DataBlade API provides the following functions to assist in determining whether a value is the SQL NULL value.

Handling the SQL NULL Value	DataBlade API Function
Can a column hold NULL values? (Was the NOT NULL constraint used to defined the column?)	**mi_column_nullable( )**, **mi_parameter_nullable( )**
Does the value represent a NULL value?	**mi_fp_argisnull( )**, **mi_fp_setargisnull( )**, **mi_fp_returnisnull( )**, **mi_fp_setreturnisnull( )**
Does the UDR handle NULL arguments? (Has the UDR been registered to indicate that it contains code to handle NULL values as arguments?)	**mi_func_handlesnulls( )**
Does an expensive-UDR argument hold a NULL value?	**mi_funcarg_isnull( )**

NULL-Valued Pointer

The NULL-valued pointer, as defined in **stddef.h**, is a DataBlade API constant that represents an initialized pointer. NULL is usually represented as zero (0) for a C pointer. However zero does *not* always represent NULL. Use the keyword NULL in your DataBlade API code to initialize pointers, as the following line shows:

```
MI_ROW *row = NULL;
```

In addition, the DataBlade API uses the NULL-value pointer for the following:
- To signify a default value for arguments in many DataBlade API functions
- To indicate an unsuccessful execution of a DataBlade API function that, when successful, returns a pointer to some value

Part 2. Data Manipulation

Chapter 3. Using Numeric Data Types

In This Chapter

The DataBlade API provides support for the following numeric data types.

Numeric Data Type	DataBlade API Numeric Data Type
Integer data types	**mi_sint1**, **mi_int1**, **mi_smallint**, **mi_unsigned_smallint**, **mi_integer**, **mi_unsigned_integer**, **mi_int8**, **mi_unsigned_int8**
Fixed-point data types	**mi_decimal**, **mi_numeric**, **mi_money**
Floating-point data types	**mi_decimal**, **mi_double_precision**, **mi_real**

This chapter describes these numeric data types as well as the functions that the DataBlade API supports to process numeric data.

Integer Data

Integer data is a value with no digits to the right of the decimal point. The DataBlade API provides support for integer values in both their text and binary representations.

Integer Text Representation

The DataBlade API supports an integer value in text representation as a quoted string that contains the following characters.

Contents of Integer String	Character
Digits	0–9
Thousands separator: symbol between every three digits	, (comma)

An integer value in its text representation is often called an *integer string*. For example, the following integer string contains the value for 1,345:

"1,345"

In an integer string, the thousands separator is optional.

> **Global Language Support**
>
> A locale defines the end-user format for numeric values. The *end-user format* is the format in which data appears in a client application when the data is a literal string or character variable. The preceding integer string is the end-user format for the default locale, U.S. English. A nondefault locale can define an end-user format that is particular to a country or culture outside the U.S. For more information, see the *IBM Informix GLS User's Guide*.
>
> **End of Global Language Support**

Integer Binary Representations

The DataBlade API provides the following data types to support the binary representations of integer values.

Integer Data	DataBlade API Data Type	SQL Integer Data Type
One-byte integers	**mi_sint1**, **mi_int1**	None
Two-byte integers	**mi_smallint**, **mi_unsigned_smallint**	SMALLINT
Four-byte integers	**mi_integer**, **mi_unsigned_integer**	INTEGER, SERIAL
Eight-byte integers	**mi_int8**, **mi_unsigned_int8**	INT8, SERIAL8

Tip: The internal format of integer data types is often referred to as their binary representation.

One-Byte Integers

The DataBlade API supports the following data types for one-byte integer values.

DataBlade API One-Byte Integer	Description
mi_sint1	Signed one-byte (eight bits) value
mi_int1	Unsigned one-byte (eight bits) value

To hold unsigned one-byte integers, you can also use the **mi_unsigned_char1** data type.

Tip: The one-byte integer data types have names that are not consistent with those of other integer data types. The **mi_int1** data type is for an unsigned one-byte integer while the **mi_smallint**, **mi_integer**, and **mi_int8** data types are for the signed version of the two-, four-, and eight-byte integers, respectively. Use the **mi_sint1** data type to hold a signed one-byte integer value.

The DataBlade API ensures that these integer data types are one byte on *all* computer architectures. There is no corresponding SQL data type for one-byte integers.

─────────────────── **Server Only** ───────────────────

Values of the **mi_int1** and **mi_sint1** data types *can* fit into an **MI_DATUM** structure. They can be passed by value within C user-defined routines (UDRs).

─────────────────── **End of Server Only** ───────────────────

─────────────────── **Client Only** ───────────────────

All data types, including **mi_int1** and **mi_sint1**, must be passed by reference within client LIBMI applications.

─────────────────── **End of Client Only** ───────────────────

Two-Byte Integers

The DataBlade API supports the following data types for two-byte integer values.

DataBlade API Two-Byte Integers	Description
mi_smallint	Signed two-byte integer value
mi_unsigned_smallint	Unsigned two-byte integer value

Use these integer data types to hold values for the SQL SMALLINT data type, which stores two-byte integer numbers that range from -32,767 to 32,767. For a description of the SQL SMALLINT data type, see the *IBM Informix Guide to SQL: Reference*.

The **mi_smallint** and **mi_unsigned_smallint** data types hold the internal (binary) format of a SMALLINT value. The DataBlade API ensures that the **mi_smallint** and **mi_unsigned_smallint** data types are two bytes on *all* computer architectures. Use these integer data types instead of the native C types (such as **short int**). If you access two-byte values stored in a SMALLINT in the database, but use the C **short int** type, conversion errors might arise if the two types are not the same size.

Important: To make your DataBlade API module portable across different architectures, it is recommended that you use the DataBlade API data type **mi_smallint** for two-byte integer values instead of the native C-language counterpart. The **mi_smallint** data type handles the different sizes of integer values across computer architectures.

─────────────────── **Server Only** ───────────────────

Values of the **mi_smallint** and **mi_unsigned_smallint** data types *can* fit into an **MI_DATUM** structure. They can be passed by value within C UDRs.

All data types, including **mi_smallint** and **mi_unsigned_smallint**, must be passed by reference within client LIBMI applications.

To transfer two-byte integers between different computer architectures, the DataBlade API provides the following functions that handle type alignment and byte order.

DataBlade API Function	Description
mi_get_smallint( )	Copies an aligned two-byte integer, converting any difference in alignment or byte order on the client computer to that of the server computer
mi_put_smallint( )	Copies an aligned two-byte integer, converting any difference in alignment or byte order on the server computer to that of the client computer
mi_fix_smallint( )	Converts the specified two-byte integer to or from the type alignment and byte order of the client computer

These DataBlade API functions are useful in the send and receive support functions of an opaque data type that contains **mi_smallint** values. They ensure that two-byte integer (SMALLINT) values remain consistent when transferred to and from client applications. For more information, see "Conversion of Opaque-Type Data with Computer-Specific Data Types" on page 16-21.

Four-Byte Integers

The DataBlade API supports the following data types for four-byte integer values.

DataBlade API Four-Byte Integers	Description
mi_integer	Signed four-byte integer value
mi_unsigned_integer	Unsigned four-byte integer value

Use these integer data types to hold values for the following SQL four-byte integer data types:

- The SQL INTEGER data type can hold integer values in the range from -2,147,483,647 to 2,147,483,647.
- The SQL SERIAL data type holds four-byte integer values that the database server automatically assigns when a value is inserted in the column.

For a description of the SQL INTEGER and SERIAL data types, see the *IBM Informix Guide to SQL: Reference*.

The **mi_integer** and **mi_unsigned_integer** data types hold the internal (binary) format of an INTEGER or SERIAL value. The DataBlade API ensures that the **mi_integer** and **mi_unsigned_integer** data types are four bytes on *all* computer architectures. Use these integer data types instead of the native C types (such as

int or **long int**). If you access four-byte values stored in a INTEGER in the database, but use the C **int** type, conversion errors might arise if the two types are not the same size.

Important: To make your DataBlade API module portable across different architectures, it is recommended that you use of the DataBlade API data type **mi_integer** for four-byte integer values instead of the native C-language counterpart. The **mi_integer** data type handles the different sizes of integer values across computer architectures.

─────────────────────────────── Server Only ───────────────────────────────

Values of the **mi_integer** and **mi_unsigned_integer** data types *can* fit into an **MI_DATUM** structure. They can be passed by value within a C UDR.

─────────────────────────── End of Server Only ───────────────────────────

─────────────────────────────── Client Only ───────────────────────────────

All data types, including **mi_integer** and **mi_unsigned_integer**, must be passed by reference within client LIBMI applications.

To transfer four-byte integers between different computer architectures, the DataBlade API provides the following functions that handle type alignment and byte order.

DataBlade API Function	Description
mi_get_integer( )	Copies an aligned four-byte integer, converting any difference in alignment or byte order on the client computer to that of the server computer
mi_put_integer( )	Copies an aligned four-byte integer, converting any difference in alignment or byte order on the server computer to that of the client computer
mi_fix_integer( )	Converts the specified four-byte integer to or from the alignment and byte order of the client computer

The **mi_get_integer( )** and **mi_put_integer( )** functions are useful in the send and receive support functions of an opaque data type that contains **mi_integer** values. They ensure that four-byte integer (INTEGER) values remain consistent when transferred to and from client applications. For more information, see "Conversion of Opaque-Type Data with Computer-Specific Data Types" on page 16-21.

─────────────────────────── End of Client Only ───────────────────────────

Eight-Byte Integers
The DataBlade API supports the following data types for eight-byte integer values.

DataBlade API Eight-Byte Integers	Description
mi_int8	Signed eight-byte integer value
mi_unsigned_int8	Unsigned eight-byte integer value

The DataBlade API ensures that these integer data types are eight bytes on *all* computer architectures. Use these integer data types to hold values for the following SQL eight-byte integer data types:

- The SQL INT8 data type can hold integer values in the range from -9,223,372,036,854,775,807 to 9,223,372,036,854,775,807 [or $-(2^{63}-1)$ to $2^{63}-1$].
- The SQL SERIAL8 data type holds eight-byte integer values that the database server automatically assigns when a value is inserted in the column.

For a description of the SQL INT8 and SERIAL8 data types, see the *IBM Informix Guide to SQL: Reference*.

The **mi_int8** and **mi_unsigned_int8** data types hold the internal (binary) format of an INT8 or SERIAL8 value.

Server Only

Values of the **mi_int8** and **mi_unsigned_int8** data types *cannot* fit into an **MI_DATUM** structure. They must be passed by reference within C UDRs.

End of Server Only

Client Only

All data types, including **mi_int8** and **mi_unsigned_int8**, must be passed by reference within client LIBMI applications.

End of Client Only

The int8.h Header File: The **int8.h** header file contains the following declarations for use with the INT8 data type:

- The **ifx_int8_t** structure
- The INT8-type functions of the Informix ESQL/C library

The **mitypes.h** header file automatically includes **int8.h**. In turn, the **milib.h** header file automatically includes **mitypes.h**, and **mi.h** automatically includes **milib.h**. Therefore, you automatically have access to the **ifx_int8_t** structure, the **mi_int8** data type, or any of the Informix ESQL/C INT8-type functions when you include **mi.h** in your DataBlade API module.

Internal INT8 Format: The INT8 data type stores eight-byte integers in an Informix-proprietary internal format: the **ifx_int8_t** structure. This structure allows the database to store eight-byte integers in a computer-independent format.

Tip: The internal format of the INT8 data type is often referred to as its binary representation.

The **mi_int8** data type uses the **ifx_int8_t** structure to hold the binary representation of an INT8 value.

Important: The **ifx_int8_t** structure is an opaque C data structure to DataBlade API modules. Do not access its internal fields directly. The internal structure of **ifx_int8_t** may change in future releases.

ESQL/C INT8-Type Functions: Because the binary representation of an INT8 (and **mi_int8**) value is an Informix-proprietary format, you cannot use standard system functions to perform integer calculations on **mi_int8** values. Instead, the DataBlade API provides support for the following categories of Informix ESQL/C functions on the INT8 data type.

Type of INT8 Function	More Information
Conversion functions	"Converting INT8 Values" on page 3-7
Arithmetic-operation functions	"Performing Operations on Eight-Byte Values" on page 3-8

Any other operations, modifications, or analyses can produce unpredictable results.

Transferring Eight-Byte Integers (Server): To transfer eight-byte integers between different computer architectures, the DataBlade API provides the following functions that handle type alignment and byte order.

DataBlade API Function	Description
mi_get_int8( )	Copies an aligned eight-byte integer, converting any difference in alignment or byte order on the client computer to that of the server computer
mi_put_int8( )	Copies an aligned eight-byte integer, converting any difference in alignment or byte order on the server computer to that of the client computer

The **mi_get_int8( )** and **mi_put_int8( )** functions are useful in the send and receive support function of an opaque data type that contains **mi_int8** values. They ensure that eight-byte integer (INT8) values remain aligned when transferred to and from client applications. For more information, see "Conversion of Opaque-Type Data with Computer-Specific Data Types" on page 16-21.

Converting INT8 Values: The Informix ESQL/C library provides the following functions that facilitate conversion of the binary representation of INT8 (**mi_int8**) values to and from some C-language data types.

Function Name	Description
ifx_int8cvasc( )	Converts a C **char** type value to an **mi_int8** type value
ifx_int8cvdbl( )	Converts a C **double** (**mi_double_precision**) type value to an **mi_int8** type value
ifx_int8cvdec( )	Converts a **mi_decimal** type value to an **mi_int8** type value
ifx_int8cvflt( )	Converts a C **float** (**mi_real**) type value to an **mi_int8** type value
ifx_int8cvint( )	Converts a C two-byte integer value to an **mi_int8** type value
ifx_int8cvlong( )	Converts a C four-byte integer value to an **mi_int8** type value
ifx_int8toasc( )	Converts **an mi_int8** type value to a text string
ifx_int8todbl( )	Converts an **mi_int8** type value to a C **double** (**mi_double_precision**) type value

ifx_int8todec( )	Converts an **mi_int8** type value to a **mi_decimal** type value
ifx_int8toflt( )	Converts an **mi_int8** type value to a C **float** (**mi_real**) type value
ifx_int8toint( )	Converts an **mi_int8** type value to a C two-byte integer value
ifx_int8tolong( )	Converts an **mi_int8** type value to a C four-byte integer value

Performing Operations on Eight-Byte Values: Use the following Informix ESQL/C library functions to perform arithmetic operations on INT8 (**mi_int8**) type values.

Function Name	Description
ifx_int8add( )	Adds two **mi_int8** numbers
ifx_int8cmp( )	Compares two **mi_int8** numbers
ifx_int8copy( )	Copies an **mi_int8** number
ifx_int8div( )	Divides two **mi_int8** numbers
ifx_int8mul( )	Multiplies two **mi_int8** numbers
ifx_int8sub( )	Subtracts two **mi_int8** numbers

Any other operations, modifications, or analyses can produce unpredictable results.

Fixed-Point Data

Fixed-point data is a decimal value with a fixed number of digits to the right and left of the decimal point. The fixed number of digits to the right of the decimal point is called the *scale* of the value. The total number of digits in the fixed-point value is called the *precision* of the value.

The DataBlade API provides support for the following kinds of fixed-point data (which correspond to existing SQL data types).

Type of Fixed-Point Value	SQL Data Type
Decimal	DECIMAL(p,s)
Monetary	MONEY(p)

Each of these kinds of fixed-point values has a text and a binary representation.

Fixed-Point Text Representations

The text representation of a fixed-point value is a quoted string that contains a series of digits. The DataBlade API supports a text representation for both decimal and monetary values.

Decimal Text Representation

The DataBlade API supports a decimal value in text representation as a quoted string that contains the characters that the following table shows.

Contents of Fixed-Point String	Character
Digits	0–9

Contents of Fixed-Point String	Character
Thousands separator: symbol between every three digits	, (comma)
Decimal separator: symbol between the integer and fraction portions of the number	. (period)

A decimal value in its text representation is often called a *decimal string*. For example, the following decimal string contains the value for 1,345.77:

`"1,345.77"`

In a decimal string, the thousands separator is optional.

Global Language Support

A locale defines the end-user format for numeric values. The *end-user format* is the format in which data appears in a client application when the data is a literal string or character variable. The preceding decimal string is the end-user format for the default locale, U.S. English. A nondefault locale can define an end-user format that is particular to a country or culture outside the U.S. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Monetary Text Representation

The DataBlade API supports a monetary value in text representation as a quoted string that contains the characters that the following table shows.

Contents of Fixed-Point String	Character
Digits	0–9
Thousands separator: symbol between every three digits	, (comma)
Decimal separator: symbol between the integer and fraction portions of the number	. (period)
Currency symbol: symbol that identifies the units of currency (can appear in front of or at the end of the monetary value)	$ (dollar sign)

A monetary value in its text representation is often called a *monetary string*. For example, the following money string contains the value for $1,345.77:

`"$1,345.77"`

In a monetary string, the thousands separator and the currency symbol are optional. You can change the format of the monetary string with the **DBMONEY** environment variable.

Global Language Support

A locale defines the end-user format for monetary values. The *end-user format* is the format in which data appears in a client application when the data is a literal string or character variable. The preceding monetary string is the end-user format for the default locale, U.S. English. A nondefault locale can define monetary end-user formats that are particular to a country or culture outside the U.S. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Fixed-Point Binary Representations

The DataBlade API provides the following data types to support the binary representations of SQL fixed-point data types.

DataBlade API Data Type	SQL Fixed-Point Data Type
mi_decimal, **mi_numeric**	DECIMAL
mi_money	MONEY

Both the DECIMAL and MONEY data types use the same internal format to store a fixed-point value. For more information on this format, see "Internal Fixed-Point Decimal Format" on page 3-12.

DECIMAL Data Type: Fixed-Point Data

When you define a column with the DECIMAL(p,s) data type, the syntax of this definition specifies a fixed-point value for the column. This value has a total of p (<= 32) significant digits (the *precision*) and s (<= p) digits to the right of the decimal point (the *scale*).

Tip: The DECIMAL data type can also declare a floating-point value with the syntax DECIMAL(p). For more information, see "DECIMAL Data Type: Floating-Point Data" on page 3-17. For a complete description of the DECIMAL data type, see the *IBM Informix Guide to SQL: Reference*.

The SQL DECIMAL data type holds the internal (binary) format of a decimal value. This value is a computer-independent method that represents numbers of up to 32 significant digits, with valid values in the range 10^{-129} to 10^{+125}. For more information, see "Internal Fixed-Point Decimal Format" on page 3-12.

Tip: The internal format of the DECIMAL data type is often referred to as its binary representation.

The DataBlade API supports the SQL DECIMAL data type with the **mi_decimal** data type. Therefore, the **mi_decimal** data type also holds the binary representation of a decimal value. The **mi_numeric** data type is a synonym for **mi_decimal**.

──────────────────────────── **Server Only** ────────────────────────────

Values of the **mi_decimal** data type *cannot* fit into an **MI_DATUM** structure. They must be passed by reference within C UDRs.

──────────────────────────── **End of Server Only** ────────────────────────────

──────────────────────────── **Client Only** ────────────────────────────

All data types, including **mi_decimal**, must be passed by reference within client LIBMI applications.

──────────────────────────── **End of Client Only** ────────────────────────────

MONEY Data Type

When you define a column with the MONEY(p) data type, it has a total of p (<= 32) significant digits (the *precision*) and a *scale* of 2 digits.

You can also specify a scale with the MONEY(p,s) syntax, where s represents the scale. For a complete description of the MONEY data type, see the *IBM Informix Guide to SQL: Reference*.

Tip: The internal format of the MONEY data type is often referred to as its binary representation.

The DataBlade API supports the SQL MONEY data type with the **mi_money** data type. The **mi_money** data type holds the internal (binary) format of a MONEY value. This binary representation of the MONEY data type has the same structure as the fixed-point DECIMAL data type. For more information, see "Internal Fixed-Point Decimal Format" on page 3-12.

The decimal.h Header File

The **decimal.h** header file contains definitions for use with the DECIMAL and MONEY data types. This header file defines the following items:

- The **dec_t** typedef
- The decimal macros
- The DECIMAL-type functions of the Informix ESQL/C library

The **mitypes.h** header file automatically includes **decimal.h**. In turn, the **milib.h** header file automatically includes **mitypes.h**, and **mi.h** automatically includes **milib.h**. Therefore, you automatically have access to the **dec_t** structure, the **mi_decimal** and **mi_money** data types, any of the decimal macros, or any of the Informix ESQL/C DECIMAL-type functions when you include **mi.h** in your DataBlade API module.

Internal Fixed-Point Decimal Format: The DECIMAL and MONEY data types store fixed-point values in an Informix-proprietary internal format: the **dec_t** structure. This structure holds the internal (binary) format of a DECIMAL or MONEY value, as follows:

```
#define DECSIZE 16

struct decimal
    {
    short dec_exp;
    short dec_pos;
    short dec_ndgts;
    char  dec_dgts[DECSIZE];
    };

typedef struct decimal dec_t;
```

This **dec_t** structure stores the number in *pairs* of digits. Each pair is a number in the range 00 to 99. (Therefore, you can think of a pair as a base-100 digit.) Table 3-1 shows the four parts of the **dec_t** structure.

Table 3-1. Fields in the dec_t Structure

Field	Description
dec_exp	The *exponent* of the normalized **dec_t** type number
	The normalized form of this number has the decimal point at the left of the left-most digit. This exponent represents the number of digit pairs to count from the *left* to position the decimal point (or as a power of 100 for the number of base-100 numbers).
dec_pos	The *sign* of the **dec_t** type number
	The dec_pos can assume any one of the following three values:
	`1` when the number is zero or greater
	`0` when the number is less than zero
	`-1` when the value is null
dec_ndgts	The *number of digit pairs* (number of base-100 significant digits) in the **dec_t** type number
	This value is also the number of entries in the **dec_dgts** array.
dec_dgts[]	A character array that holds the significant digits of the normalized **dec_t** type number, assuming **dec_dgts[0] != 0**
	Each byte in the array contains the next significant base-100 digit in the **dec_t** type number, proceeding from **dec_dgts[0]** to **dec_dgts[dec_ndgts]**.

Table 3-2 shows some sample **dec_t** values.

Table 3-2. Sample Decimal Values

	dec_t Structure Field Values			
Value	**dec_exp**	**dec_pos**	**dec_ndgts**	**dec_dgts[]**
-12345.6789	3	0	5	dec_dgts[0] = 01
				dec_dgts[1] = 23
				dec_dgts[2] = 45
				dec_dgts[3] = 67
				dec_dgts[4] = 89

Table 3-2. Sample Decimal Values (continued)

	dec_t Structure Field Values			
Value	dec_exp	dec_pos	dec_ndgts	dec_dgts[]
1234.567	2	1	4	dec_dgts[0] = 12
				dec_dgts[1] = 34
				dec_dgts[2] = 56
				dec_dgts[3] = 70
-123.456	2	0	4	dec_dgts[0] = 01
				dec_dgts[1] = 23
				dec_dgts[2] = 45
				dec_dgts[3] = 60
480	2	1	2	dec_dgts[0] = 04
				dec_dgts[1] = 80
.152	0	1	2	dec_dgts[0] = 15
				dec_dgts[1] = 20
-6	1	0	1	dec_dgts[0] = 06

The **mi_decimal** and **mi_money** data types use the **dec_t** structure to hold the binary representation of a DECIMAL and MONEY value, respectively.

The Decimal Macros: The **decimal.h** header file also includes the following macros that might be useful in a DataBlade API module.

Decimal Macro	Description
DECLEN(*p*, *s*)	Calculates the minimum number of bytes required to hold the DECIMAL(*p,s*) value
DECPREC(*size*)	Calculates a default precision given the number of bytes (*size*) used to store the number
PRECTOT(*dec*)	Returns the total precision of the *dec* value
PRECDEC(*dec*)	Returns the scale of the *dec* value
PRECMAKE(*p*, *s*)	Creates a precision value from the specified total precision (*p*) and scale (*s*)
	Tip: For a complete list of decimal macros, consult the **decimal.h** header file that is installed with your database server. This header file resides in the **incl/public** subdirectory of the **INFORMIXDIR** directory.

ESQL/C DECIMAL-Type Functions: Because the binary representation of DECIMAL (**mi_decimal**) and MONEY (**mi_money**) values is an Informix-proprietary format, you cannot use standard system functions to perform decimal operations on **mi_decimal** and **mi_money** values. Instead, the DataBlade API provides support for the following Informix ESQL/C functions on the DECIMAL and MONEY data types.

Type of DECIMAL Function	More Information
Conversion functions	"ESQL/C Functions for Decimal Conversion" on page 3-15
Arithmetic-operation functions	"Performing Operations on Decimal Data" on page 3-16

Any other operations, modifications, or analyses can produce unpredictable results.

Transferring Fixed-Point Data (Server)

To transfer fixed-point data between different computer architectures, the DataBlade API provides the following functions that handle type alignment and byte order.

DataBlade API Function	Description
mi_get_decimal( )	Copies an aligned **mi_decimal** value, converting any difference in alignment or byte order on the client computer to that of the server computer
mi_get_money( )	Copies an aligned **mi_money** value, converting any difference in alignment or byte order on the client computer to that of the server computer
mi_put_decimal( )	Copies an aligned **mi_decimal** value, converting any difference in alignment or byte order on the server computer to that of the client computer
mi_put_money( )	Copies an aligned **mi_money** value, converting any difference in alignment or byte order on the server computer to that of the client computer

The **mi_get_decimal( )**, **mi_get_money( )**, **mi_put_decimal( )**, and **mi_put_money( )** functions are useful in the send and receive support function of an opaque data type that contains **mi_decimal** or **mi_money** values. They ensure that fixed-point (DECIMAL or MONEY) values remain aligned when transferred to and from client applications. For more information, see "Conversion of Opaque-Type Data with Computer-Specific Data Types" on page 16-21.

Converting Decimal Data

Both the DataBlade API library and the Informix ESQL/C library provide functions that convert the binary representation for DECIMAL (**mi_decimal**) or MONEY (**mi_money**) values.

DataBlade API Functions for Decimal Conversion

The DataBlade API library provides the following functions that convert between a text (string) representation of a decimal or monetary value and its binary (internal) equivalent.

DataBlade API Function	Converts from	Converts to
mi_decimal_to_string( )	DECIMAL (**mi_decimal**)	Decimal string
mi_money_to_string( )	MONEY (**mi_money**)	Interval string
mi_string_to_decimal( )	Decimal string	DECIMAL (**mi_decimal**)
mi_string_to_money( )	Monetary string	MONEY (**mi_money**)

The **mi_decimal_to_string( )**, **mi_money_to_string( )**, **mi_string_to_decimal( )**, and **mi_string_to_money( )** functions are useful in the input and output support function of an opaque data type that contains **mi_decimal** or **mi_money** values. They allow you to convert fixed-point (DECIMAL or MONEY) values between their external format (text) and their internal format (**dec_t**) when transferred to and from client applications. For more information, see "Conversion of Opaque-Type Data Between Text and Binary Representations" on page 16-16.

The **mi_decimal_to_string( )**, **mi_money_to_string( )**, **mi_string_to_decimal( )**, and **mi_string_to_money( )** functions use the current processing locale to handle locale-specific formats in the decimal or monetary string. For more information, see the *IBM Informix GLS User's Guide*.

ESQL/C Functions for Decimal Conversion

The Informix ESQL/C function library provides the following functions to convert a DECIMAL (or MONEY) value to and from some C-language data types.

Function Name	Description
deccvasc( )	Converts a C **char** type to an **mi_decimal** type value
deccvdbl( )	Converts a C **double** (**mi_double_precision**) type to an **mi_decimal** type value
deccvint( )	Converts a C two-byte integer value to an **mi_decimal** type value
deccvlong( )	Converts a C four-byte integer value to an **mi_decimal** type value
dececvt( ) and **decfcvt( )**	Converts an **mi_decimal** type value to text
dectoasc( )	Converts an **mi_decimal** type value to text
dectodbl( )	Converts an **mi_decimal** type value to a C **double** (**mi_double_precision**) type value
dectoint( )	Converts an **mi_decimal** type value to a C two-byte integer value
dectolong( )	Converts an **mi_decimal** type value to a C four-byte integer value
	Tip: The Informix ESQL/C library also provides functions to convert some numeric data types to formatted strings. For more information, see "Formatting Numeric Strings" on page 3-20.

Performing Operations on Decimal Data

The Informix ESQL/C function library provides the following functions to perform arithmetic operations on DECIMAL (**mi_decimal**) and MONEY (**mi_money**) values.

Function Name	Description
decadd( )	Adds two **mi_decimal** numbers
deccmp( )	Compares two **mi_decimal** numbers
deccopy( )	Copies a **mi_decimal** number
decdiv( )	Divides two **mi_decimal** numbers
decmul( )	Multiplies two **mi_decimal** numbers
decround( )	Rounds an **mi_decimal** number
decsub( )	Subtracts two **mi_decimal** numbers
dectrunc( )	Truncates an **mi_decimal** number

Any other operations, modifications, or analyses can produce unpredictable results.

Obtaining Fixed-Point Type Information

The DataBlade API provides the following functions to obtain the scale and precision of a fixed-point (DECIMAL and MONEY) data type.

Source	DataBlade API Functions
For a data type	**mi_type_precision( )**, **mi_type_scale( )**
For a UDR argument	**mi_fp_argprec( )**, **mi_fp_setargprec( )**, **mi_fp_argscale( )**, **mi_fp_setargscale( )**
For a UDR return value	**mi_fp_retprec( )**, **mi_fp_setretprec( )**, **mi_fp_retscale( )**, **mi_fp_setretscale( )**
For a column in a row (or field in a row type)	**mi_column_precision( )**, **mi_column_scale( )**
For an input parameter in a prepared statement	**mi_parameter_precision( )**, **mi_parameter_scale( )**

Floating-Point Data

A *floating-point value* is a large decimal value that is stored in a fixed field width. Because the field width is fixed, a floating-point number that is larger than the field width only retains its most significant digits. That is, digits that do not fit into the fixed width are dropped (rounded or truncated).

The DataBlade API provides support for the following kinds of floating-point data (which correspond to existing SQL data types).

Type of Floating-Point Value	SQL Data Type
Decimal	DECIMAL(p)
True floating-point	SMALLFLOAT, FLOAT

These floating-point values have both text and binary representations.

Floating-Point Text Representation

The DataBlade API supports a floating-point value in text representation as a quoted string that contains the following characters.

Contents of Integer String	Character
Digits	0–9
Thousands separator: symbol between every three digits	, (comma)
Decimal separator: symbol between the integer and fraction portions of the number	. (period)

For example, the following integer string contains the value for 1,345.77431:

"1**,**345.77431"

In a floating-point string, the thousands separator is optional.

Important: Because floating-point numbers retain only their most significant digits, the number that you enter in this type of column and the number the database server displays can differ slightly.

Global Language Support

A locale defines the end-user format for numeric values. The *end-user format* is the format in which data appears in a client application when the data is a literal string or character variable. The preceding floating-point string is the end-user format for the default locale, U.S. English. A nondefault locale can define an end-user format that is particular to a country or culture outside the U.S. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Floating-Point Binary Representations

The DataBlade API provides the following data types to support the binary representations of floating-point values.

SQL Floating-Point Data Type	DataBlade API Data Type
DECIMAL	**mi_decimal**
SMALLFLOAT	**mi_real**
FLOAT	**mi_double_precision**

DECIMAL Data Type: Floating-Point Data

When you define a column with the DECIMAL(p) data type, the syntax of this definition specifies a floating-point value for the column. This value has a total of p (<= 32) significant digits (its precision). DECIMAL(p) has an absolute value range between 10^{-130} and 10^{124}.

Tip: The DECIMAL data type can also declare a fixed-point value with the syntax DECIMAL(p,s). For more information, see "DECIMAL Data Type: Fixed-Point Data" on page 3-10. For a complete description of the DECIMAL data type, see the *IBM Informix Guide to SQL: Reference*.

The **mi_decimal** data type stores floating-point DECIMAL values as well as fixed-point values. Therefore, information about **mi_decimal** in "Fixed-Point Data"

on page 3-8 also applies to **mi_decimal** when it contains a floating-point value. In particular, the following statements are true.

Decimal Information	More Information
The **mi_decimal** data type stores values in an internal (binary) format.	"Internal Fixed-Point Decimal Format" on page 3-12
All the Informix ESQL/C library functions that handle fixed-point values in **mi_decimal** can also handle **mi_decimal** when it contains floating-point values.	"ESQL/C DECIMAL-Type Functions" on page 3-13
All DataBlade API functions that accept fixed-point values in **mi_decimal** also accept **mi_decimal** when it contains a floating-point value.	"Transferring Fixed-Point Data (Server)" on page 3-14 and "Converting Decimal Data" on page 3-14

───── Server Only ─────

Values of the **mi_decimal** data type *cannot* fit into an **MI_DATUM** structure. They must be passed by reference within C UDRs.

───── End of Server Only ─────

───── Client Only ─────

All data types, including **mi_decimal**, must be passed by reference within client LIBMI applications.

───── End of Client Only ─────

SMALLFLOAT Data Type

The SQL SMALLFLOAT data type can hold single-precision floating-point values. The DataBlade API supports the SMALLFLOAT data type with the **mi_real** data type. The **mi_real** data type stores internal SMALLFLOAT values, as 32-bit floating-point values.

───── Server Only ─────

Although an **mi_real** value can fit into an **MI_DATUM** structure, values of this data type are *always* passed by reference. Unlike other four-byte values, **mi_real** values cannot be passed by value. All values greater than four bytes are passed by reference.

Therefore, if a UDR is called from an SQL statement, the database server passed a pointer to any **mi_real** arguments; it does not pass the actual value. Similarly, if a user-defined function returns an **mi_real** value to an SQL statement, you must allocate space for the value, fill this space, and return a pointer to this space.

DataBlade API modules that are not invoked from SQL statements might pass **mi_real** values by value. However, for consistency, you might want to pass them by reference.

All data types, including **mi_real**, must be passed by reference within client LIBMI applications.

Important: To make your DataBlade API module portable across different architectures, it is recommended that you use the DataBlade API data type, **mi_real**, instead of the native C-language counterpart, **float**. The **mi_real** data type handles the different sizes of small floating-point values across computer architectures.

The FLOAT Data Type

The SQL FLOAT data type can hold double-precision floating-point values. The DataBlade API supports the FLOAT data type with the **mi_double_precision** data type. The **mi_double_precision** data type stores internal FLOAT values, as 64-bit floating-point values.

Values of the **mi_double_precision** data type *cannot* fit into an **MI_DATUM** structure. They must be passed by reference within C UDRs.

All data types, including **mi_double_precision**, must be passed by reference within client LIBMI applications.

Important: To make your DataBlade API module portable across different architectures, it is recommended that you use the DataBlade API data type, **mi_double_precision**, instead of the native C-language counterpart, **double**. The **mi_double_precision** data type handles the different sizes of large floating-point values across computer architectures.

Transferring Floating-Point Data (Server)

To transfer floating-point data between different computer architectures, the DataBlade API provides the following functions that handle type alignment and byte order.

DataBlade API Function	Description
mi_get_decimal( )	Copies an aligned **mi_decimal** value, converting any difference in alignment or byte order on the client computer to that of the server computer
mi_get_double_precision( )	Copies an aligned **mi_double_precision** value, converting any difference in alignment or byte order on the client computer to that of the server computer

DataBlade API Function	Description
mi_get_real( )	Copies an aligned **mi_real** value, converting any difference in alignment or byte order on the client computer to that of the server computer
mi_put_decimal( )	Copies an aligned **mi_decimal** value, converting any difference in alignment or byte order on the server computer to that of the client computer
mi_put_double_precision( )	Copies an aligned **mi_double_precision** value, converting any difference in alignment or byte order on the server computer to that of the client computer
mi_put_real( )	Copies an aligned **mi_real** value, converting any difference in alignment or byte order on the server computer to that of the client computer

The **mi_get_decimal( )**, **mi_get_double_precision( )**, **mi_get_real( )**, **mi_put_decimal( )**, **mi_put_double_precision( )**, and **mi_put_real( )** functions are useful in the send and receive support function of an opaque data type that contains **mi_decimal**, **mi_double_precision**, or **mi_real** values. They ensure that floating-point (DECIMAL, FLOAT, or SMALLFLOAT) values remain aligned when transferred to and from client applications. For more information, see "Conversion of Opaque-Type Data with Computer-Specific Data Types" on page 16-21.

Converting Floating-Point Decimal Data

Both the DataBlade API library and the Informix ESQL/C library provide functions that convert between floating-point decimal strings and internal DECIMAL formats. For more information, see "Converting Decimal Data" on page 3-14.

Obtaining Floating-Point Type Information

The DataBlade API provides the following functions to obtain the precision of a floating-point DECIMAL (DECIMAL(p)).

Source	DataBlade API Functions
For a data type	**mi_type_precision( )**
For a UDR argument	**mi_fp_argprec( )**, **mi_fp_setargprec( )**
For a UDR return value	**mi_fp_retprec( )**, **mi_fp_setretprec( )**
For a column	**mi_column_precision( )**
For an input parameter in a prepared statement	**mi_parameter_precision( )**

Tip: The FLOAT and SMALLFLOAT data types do not have precision and scale values.

Formatting Numeric Strings

The Informix ESQL/C library provides special functions that enable you to format numeric expressions as strings. These numeric-formatting functions apply a given formatting mask to a numeric value to allow you to line up decimal points, right- or left-justify the number, enclose a negative number in parentheses, and so on. The Informix ESQL/C library includes the following functions that support numeric-formatting masks for numeric values.

Function Name	Description

rfmtdec( )	Converts an **mi_decimal** value to a string
rfmtdouble( )	Converts a C-language **double** value to a string
rfmtlong( )	Converts a C-language **long integer** value to a string

> **Tip:** Both the Informix ESQL/C library and the DataBlade API library provide functions to convert between **mi_decimal** values and other C-language data types. For more information, see "Converting Decimal Data" on page 3-14.

This section describes the characters that you can use to create a numeric-formatting mask. It also provides extensive examples that show the results of applying these masks to numeric values. A *numeric-formatting mask* specifies a format to apply to some numeric value. This mask is a combination of the following formatting characters:

*	This character fills with asterisks any positions in the display field that would otherwise be blank.
&	This character fills with zeros any positions in the display field that would otherwise be blank.
#	This character changes leading zeros to blanks. Use this character to specify the maximum leftward extent of a field.
<	This character left-justifies the numbers in the display field. It changes leading zeros to a null string.
,	This character indicates the symbol that separates groups of three digits (counting leftward from the units position) in the whole-number part of the value. By default, this symbol is a comma. You can set the symbol with the **DBMONEY** environment variable. In a formatted number, this symbol appears only if the whole-number part of the value has four or more digits.
.	This character indicates the symbol that separates the whole-number part of a money value from the fractional part. By default, this symbol is a period. You can set the symbol with the **DBMONEY** environment variable. You can have only one period in a format string.
-	This character is a literal. It appears as a minus sign when *expr1* is less than zero. When you group several minus signs in a row, a single minus sign floats to the rightmost position that it can occupy; it does not interfere with the number and its currency symbol.
+	This character is a literal. It appears as a plus sign when *expr1* is greater than or equal to zero and as a minus sign when *expr1* is less than zero. When you group several plus signs in a row, a single plus or minus sign floats to the rightmost position that it can occupy; it does not interfere with the number and its currency symbol.
(	This character is a literal. It appears as a left parenthesis to the left of a negative number. It is one of the pair of accounting parentheses that replace a minus sign for a negative number. When

	you group several in a row, a single left parenthesis floats to the rightmost position that it can occupy; it does not interfere with the number and its currency symbol.
)	This is one of the pair of accounting parentheses that replace a minus sign for a negative value.
$	This character displays the currency symbol that appears at the front of the numeric value. By default, the currency symbol is the dollar ($) sign. You can set the currency symbol with the **DBMONEY** environment variable. When you group several dollar signs in a row, a single currency symbol floats to the rightmost position that it can occupy; it does not interfere with the number.

Any other characters in the formatting mask are reproduced literally in the result.

When you use the following characters within a formatting mask, the characters *float*; that is, multiple occurrences of the character at the left of the pattern in the mask appear as a single character as far to the right as possible in the formatted number (without removing significant digits):

-
+
(
)
$

For example, if you apply the mask $$$,$$$.## to the number 1234.56, the result is $1,234.56.

Global Language Support

When you use **rfmtdec( )**, **rfmtdouble( )**, or **rfmtlong( )** to format MONEY values, the function uses the currency symbols that the **DBMONEY** environment variable specifies. If you do not set this environment variable, the numeric-formatting functions use the currency symbols that the client locale defines. The default locale, U.S. English, defines currency symbols as if you set **DBMONEY** to "**$,.**". (For a discussion of **DBMONEY**, see the *IBM Informix Guide to SQL: Reference*). For more information on locales, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Table 3-3 shows sample format strings for numeric expressions. The character b represents a blank or space.

Table 3-3. Sample Format Patterns and Their Results

Formatting Mask	Numeric Value	Formatted Result
"#####"	0	bbbbb
"&&&&&"	0	00000
"$$$$$"	0	bbbb$
"*****"	0	*****
"<<<<<"	0	(null string)

Table 3-3. Sample Format Patterns and Their Results (continued)

Formatting Mask	Numeric Value	Formatted Result	
"##,###"	12345	12,345	
"##,###"	1234	b1,234	
"##,###"	123	bbb123	
"##,###"	12	bbbb12	
"##,###"	1	bbbbb1	
"##,###"	-1	bbbbb1	
"##,###"	0	bbbbbb	
"&&,&&&"	12345	12,345	
"&&,&&&"	1234	01,234	
"&&,&&&"	123	000123	
"&&,&&&"	12	000012	
"&&,&&&"	1	000001	
"&&,&&&"	-1	000001	
"&&,&&&"	0	000000	
"$$,$$$"	12345	*****	(overflow)
"$$,$$$"	1234	$1,234	
"$$,$$$"	123	bb$123	
"$$,$$$"	12	bbb$12	
"$$,$$$"	1	bbbb$1	
"$$,$$$"	-1	bbbb$1	
"$$,$$$"	0	bbbbb$	
"$$,$$$"	1234	DM1,234	
(**DBMONEY** set to DM)			
"**,***"	12345	12,345	
"**,***"	1234	*1,234	
"**,***"	123	***123	
"**,***"	12	****12	
"**,***"	1	*****1	
"**,***"	0	******	
"##,###.##"	12345.67	12,345.67	
"##,###.##"	1234.56	b1,234.56	
"##,###.##"	123.45	bbb123.45	
"##,###.##"	12.34	bbbb12.34	
"##,###.##"	1.23	bbbbb1.23	
"##,###.##"	0.12	bbbbbb.12	
"##,###.##"	0.01	bbbbbb.01	
"##,###.##"	-0.01	bbbbbb.01	
"##,###.##"	-1	bbbbb1.00	
"&&,&&&.&&"	.67	000000.67	
"&&,&&&.&&"	1234.56	01,234.56	
"&&,&&&.&&"	123.45	000123.45	
"&&,&&&.&&"	0.01	000000.01	
"$$,$$$.$$"	12345.67	*********	(overflow)
"$$,$$$.$$"	1234.56	$1,234.56	
"$$,$$$.##"	0.00	bbbbb$.00	
"$$,$$$.##"	1234.00	$1,234.00	
"$$,$$$.&&"	0.00	bbbbb$.00	
"$$,$$$.&&"	1234.00	$1,234.00	

Table 3-3. Sample Format Patterns and Their Results (continued)

Formatting Mask	Numeric Value	Formatted Result
"-##,###.##"	-12345.67	-12,345.67
"-##,###.##"	-123.45	-bbb123.45
"-##,###.##"	-12.34	-bbbb12.34
"--#,###.##"	-12.34	b-bbb12.34
"---,###.##"	-12.34	bb-bb12.34
"---,-##.##"	-12.34	bbbb-12.34
"---,--#.##"	-12.34	bbbb-12.34
"--#,###.##"	-1.00	b-bbbb1.00
"---,--#.##"	-1.00	bbbbb-1.00
"-##,###.##"	12345.67	b12,345.67
"-##,###.##"	1234.56	bb1,234.56
"-##,###.##"	123.45	bbbb123.45
"-##,###.##"	12.34	bbbbb12.34
"--#,###.##"	12.34	bbbbb12.34
"---,###.##"	12.34	bbbbb12.34
"---,-##.##"	12.34	bbbbb12.34
"---,---.##"	1.00	bbbbbb1.00
"---,---.--"	-.01	bbbbbb-.01
"---,---.&&"	-.01	bbbbbb-.01
"-$$$,$$$.&&"	-12345.67	-$12,345.67
"-$$$,$$$.&&"	-1234.56	-b$1,234.56
"-$$$,$$$.&&"	-123.45	-bbb$123.45
"--$,$$$.&&"	-12345.67	-$12,345.67
"--$,$$$.&&"	-1234.56	b-$1,234.56
"--$,$$$.&&"	-123.45	b-bb$123.45
"--$,$$$.&&"	-12.34	b-bbb$12.34
"--$,$$$.&&"	-1.23	b-bbbb$1.23
"----,--$.&&"	-12345.67	-$12,345.67
"----,--$.&&"	-1234.56	b-$1,234.56
"----,--$.&&"	-123.45	bbb-$123.45
"----,--$.&&"	-12.34	bbbb-$12.34
"----,--$.&&"	-1.23	bbbbb-$1.23
"----,--$.&&"	-.12	bbbbbb-$.12
"$***,***.&&"	12345.67	$*12,345.67
"$***,***.&&"	1234.56	$**1,234.56
"$***,***.&&"	123.45	$****123.45
"$***,***.&&"	12.34	$*****12.34
"$***,***.&&"	1.23	$******1.23
"$***,***.&&"	.12	$*******.12
"($$$,$$$.&&)"	-12345.67	($12,345.67)
"($$$,$$$.&&)"	-1234.56	(b$1,234.56)
"($$$,$$$.&&)"	-123.45	(bbb$123.45)
"(($$,$$$.&&)"	-12345.67	($12,345.67)
"(($$,$$$.&&)"	-1234.56	b($1,234.56)
"(($$,$$$.&&)"	-123.45	b(bb$123.45)
"(($$,$$$.&&)"	-12.34	b(bbb$12.34)
"(($$,$$$.&&)"	-1.23	b(bbbb$1.23)
"(((((,(($.&&)"	-12345.67	($12,345.67)
"(((((,(($.&&)"	-1234.56	b($1,234.56)
"(((((,(($.&&)"	-123.45	bbb($123.45)
"(((((,(($.&&)"	-12.34	bbbb($12.34)
"(((((,(($.&&)"	-1.23	bbbbb($1.23)
"(((((,(($.&&)"	-.12	bbbbbb($.12)

Table 3-3. Sample Format Patterns and Their Results (continued)

Formatting Mask	Numeric Value	Formatted Result
″($$$,$$$.&&)″	12345.67	b$12,345.67
″($$$,$$$.&&)″	1234.56	bb$1,234.56
″($$$,$$$.&&)″	123.45	bbbb$123.45
″(($$,$$$.&&)″	12345.67	b$12,345.67
″(($$,$$$.&&)″	1234.56	bb$1,234.56
″(($$,$$$.&&)″	123.45	bbbb$123.45
″(($$,$$$.&&)″	12.34	bbbbb$12.34
″(($$,$$$.&&)″	1.23	bbbbbb$1.23
″((((,(($.&&)″	12345.67	b$12,345.67
″((((,(($.&&)″	1234.56	bb$1,234.56
″((((,(($.&&)″	123.45	bbbb$123.45
″((((,(($.&&)″	12.34	bbbbb$12.34
″((((,(($.&&)″	1.23	bbbbbb$1.23
″((((,(($.&&)″	.12	bbbbbbb$.12
″<<<,<<<″	12345	12,345
″<<<,<<<″	1234	1,234
″<<<,<<<″	123	123
″<<<,<<<″	12	12

Chapter 4. Using Date and Time Data Types

In This Chapter

The DataBlade API provides support for the following date and time data types.

SQL Date and Time Data Type	Standard C or ESQL/C Date and Time Data Type	DataBlade API Date and Time Data Type
DATE	C: four-byte integerInformix ESQL/C: **date**	**mi_date**
DATETIME	Informix ESQL/C: **datetime, dtime_t**	**mi_datetime**
INTERVAL	Informix ESQL/C: **interval, intrvl_t**	**mi_interval**

This chapter describes these date and time data types as well as the functions that the DataBlade API supports to process date and time data.

Date Data

Date data is a calendar date. The DataBlade API provides support for date values in both their text and binary representations.

Date Text Representation

The DataBlade API supports a date value in text representation as a quoted string with the following format:

```
"mm/dd/yyyy"
```

mm	is the 2-digit month.
dd	is the 2-digit day of the month.
yyyy	is the 4-digit year.

A date value in its text representation is often called a *date string*. For example, the following date string contains the value for July 12, 1999 (for the default locale):

```
"7/12/1999"
```

You can change the format of the date string with the **DBDATE** environment variable.

Global Language Support

A locale defines the end-user format of a date. The *end-user format* is the format in which data appears in a client application when the data is a literal string or character variable. The preceding date string is the end-user format for the default locale, U.S. English. A nondefault locale can define an end-user format that is particular to a country or culture outside the U.S. You can also customize the end-user format of a date with the **GL_DATE** environment variable. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Date Binary Representation

The SQL DATE data type holds the internal (binary) format of a decimal value. This value is an integer value that represents the number of days since December 31, 1899. Dates before December 31, 1899, are negative numbers, while dates after December 31, 1899, are positive numbers. For a detailed description of the SQL DATE data type, see the *IBM Informix Guide to SQL: Reference*.

Tip: The internal format of the DATE data type is often referred to as its binary representation.

The DataBlade API supports the SQL DATE data type with the **mi_date** data type. Therefore, the **mi_date** data type also holds the binary representation of a date value.

Server Only

The **mi_date** data type is guaranteed to be four bytes on *all* computer architectures. All **mi_date** values can fit into an **MI_DATUM** structure and can be passed by value within C UDRs.

End of Server Only

Client Only

All data types, including **mi_date**, must be passed by reference within client LIBMI applications.

End of Client Only

Because the binary representation of a DATE (and **mi_date**) value is an
Informix-proprietary format, you cannot use standard system functions to obtain
date information from **mi_date** values. Instead, the DataBlade API provides the
following support for the DATE data type.

Category of DATE Function	More Information
Conversion functions	"Conversion of Date Representations" on page 4-3
Operation functions	"Operations on Date Data" on page 4-5

Transfers of Date Data (Server)

For date data to be portable when transferred across different computer
architectures, the DataBlade API provides the following functions to handle type
alignment and byte order.

DataBlade API Function	Description
mi_get_date( )	Copies an aligned **mi_date** value, converting any difference in alignment or byte order on the client computer to that of the server computer
mi_put_date( )	Copies an aligned **mi_date** value, converting any difference in alignment or byte order on the server computer to that of the client computer

The **mi_get_date( )** and **mi_put_date( )** functions are useful in the send and
receive support function of an opaque data type that contains **mi_date** values.
They enable you to ensure that DATE values remain aligned when transferred to
and from client applications, which possibly have unaligned data buffers. For more
information, see "Conversion of Opaque-Type Data with Computer-Specific Data
Types" on page 16-21.

Conversion of Date Representations

Both the DataBlade API library and the Informix ESQL/C library provide functions
that convert from the text (string) representation of a date value to the binary
(internal) representation for DATE.

DataBlade API Functions for Date Conversion

The DataBlade API provides the following functions for conversion between text
and binary representations of date data.

DataBlade API Function	Convert from	Convert to
mi_date_to_string( )	DATE (**mi_date**)	Date string
mi_string_to_date( )	Date string	DATE (**mi_date**)

Server Only

The **mi_date_to_string( )** and **mi_string_to_date( )** functions are useful in the
input and output support functions of an opaque data type that contains **mi_date**
values. They allow you to convert DATE values between their external format
(text) and their internal (binary) format when transferred to and from client
applications. For more information, see "Conversion of Opaque-Type Data Between
Text and Binary Representations" on page 16-16.

The **mi_date_to_string( )** and **mi_string_to_date( )** functions use the current processing locale to handle locale-specific formats in the date string. For more information, see the *IBM Informix GLS User's Guide*.

ESQL/C Functions for Date Conversion

The Informix ESQL/C function library provides the following functions to convert a DATE (**mi_date**) value to and from **char** strings.

Function Name	Description
rdatestr( )	Converts an internal format to string
rdefmtdate( )	Converts a string to an internal format using a formatting mask
rfmtdate( )	Converts an internal format to a string using a formatting mask
rstrdate( )	Converts a string to an internal format

The **rdatestr( )** and **rstrdate( )** functions convert **mi_date** values to and from a date string that is formatted with the **DBDATE** environment variable.

These functions also examine the **GL_DATE** environment variable for the format of the date string. When you use a nondefault locale and do not set the **DBDATE** or **GL_DATE** environment variable, **rdatestr( )** uses the date end-user format that the client locale defines. For more information, see the *IBM Informix GLS User's Guide*.

The **rdefmtdate( )** and **rfmtdate( )** functions convert **mi_datetime** values to and from a date-time string using a date-formatting mask. A *date-formatting mask* specifies a format to apply to some date value. This mask is a combination of the following formats.

Format	Meaning
dd	Day of the month as a two-digit number (01 through 31)
ddd	Day of the week as a three-letter abbreviation (Sun through Sat)
mm	Month as a two-digit number (01 through 12)
mmm	Month as a three-letter abbreviation (Jan through Dec)
yy	Year as a two-digit number in the 1900s (00 through 99)
yyyy	Year as a four-digit number (0001 through 9999)

Any other characters in the formatting mask are reproduced literally in the result.

Operations on Date Data

Use the following Informix ESQL/C library functions to perform operations on DATE (**mi_date**) values.

Function Name	Description
rdayofweek( )	Returns the day of the week
rjulmdy( )	Returns month, day, and year from an internal format
rleapyear( )	Determines whether a specified year is a leap year
rmdyjul( )	Returns an internal format from month, day, and year
rtoday( )	Returns a system date in internal format

Any other operations, modifications, or comparisons can produce unpredictable results.

Date-Time or Interval Data

The DataBlade API provides support for the following kinds of fixed-point data, which correspond to existing SQL data types.

Type of Fixed-Point Value	SQL Data Type
Date and time, date, or time	DATETIME
Year and month interval or day and time interval	INTERVAL

Date-time data is an instant in time that is expressed as a calendar date and time of day, just a calendar date, or just a time of day. A date-time value can also have a *precision* and a *scale.* The precision is the number of digits required to store the value. The scale is the end qualifier of the date-time value, such as YEAR TO HOUR.

Interval data is a span of time that is expressed as the number of units in either of the following interval classes:

• *Year-month intervals*

A year-month interval value specifies the number of years and months, years, or months that have passed.

- *Day-time intervals*

  A day-time interval value specifies the number of days and hours, days, or hours that have passed.

The DataBlade API provides support for date-time or interval data in both text and binary representations.

Date-Time or Interval Text Representation

The text representation of a date-time or interval value is a quoted string that contains a series of digits and symbols. The DataBlade API supports a text representation for date-time or interval values as quoted strings with the formats that the following table shows.

SQL Data Type	Text Representation
DATETIME	Date-time string:
	The date-time string must match the qualifier of the DATETIME column. The default format of the date-time string for the largest DATETIME column is:
	"yyyy-mm-dd HH:MM:SS.FFFF"
INTERVAL	Interval string:
	The interval string must match the qualifiers of the INTERVAL column. INTERVAL columns have two classes. The default format of an interval string for the largest year-month interval follows:
	"yyyy-mm"
	The default format of an interval string for the largest day-time interval follows:
	"dd HH:SS.FFFF"

The text representations in the preceding table use the following abbreviations:

yyyy	is the 4-digit year (for a DATETIME) or the number of years (for an INTERVAL).
mm	is the 2-digit month (for a DATETIME) or the number of months (for an INTERVAL).
dd	is the 2-digit day of the month (for a DATETIME) or the number of days (for an INTERVAL).
HH	is the 2-digit hour (for a DATETIME) or the number of hours (for an INTERVAL).
MM	is the 2-digit minute (for a DATETIME) or the number of minutes (for an INTERVAL).
SS	is the 2-digit second (for a DATETIME) or the number of seconds (for an INTERVAL).
FFFF	is a fraction of a second (for a DATETIME) or the number of years (for an INTERVAL). Fractions can be from 1 to 5 digits.

A date-time value in its text representation is often called a *date-time string*. For example, the following date-time string contains the value for 2 p.m. on July 12, 1999, with a qualifier of year to minute:

```
"1999-07-12 14:00:00"
```

Usually, a date-time string must match the qualifier of the date-time binary representation with which the string is associated.

The following interval string indicates a passage of three years and three months:

```
"03-06"
```

Date-Time or Interval Binary Representation

The DataBlade API supports the following SQL data types that can hold information about date-time or interval values.

DataBlade API Date and Time Data Type	SQL Date and Time Data Type
mi_datetime	DATETIME
mi_interval	INTERVAL

The DATETIME Data Type

The SQL DATETIME data type provides the internal (binary) format of a date-time value. This data type stores an instant in time expressed as a calendar date and time of day, just a calendar date, or just a time of day. You choose how precisely a DATETIME value is stored with a *qualifier*. The precision can range from a year to a fraction of a second. For a detailed description of the SQL DATETIME data type, see the *IBM Informix Guide to SQL: Reference*.

The DATETIME data type uses a computer-independent method to encode the date or time qualifiers. It stores the information in the **dtime_t** structure, as follows:

```
typedef struct dtime {
   short dt_qual;
   dec_t dt_dec;
} dtime_t;
```

The **dtime** structure and **dtime_t** typedef have two parts, which the following table shows.

Field	Description
dt_qual	The qualifier of the **datetime** value
dt_dec	The digits of the fields of the **datetime** value

This field is a **decimal** value.

Tip: The internal format of the DATETIME data type is often referred to as its binary representation.

The DataBlade API supports the SQL DATETIME data type with the **mi_datetime** data type. Therefore, the **mi_datetime** data type holds the binary representation of a date and/or time value.

──────────────────────── **Server Only** ────────────────────────

Values of the **mi_datetime** data type *cannot* fit into an **MI_DATUM** structure. They must be passed by reference within C UDRs.

──────────────────────── **End of Server Only** ────────────────────────

──────────────────────── **Client Only** ────────────────────────

All data types, including **mi_datetime**, must be passed by reference within client LIBMI applications.

──────────────────────── **End of Client Only** ────────────────────────

Because the binary representation of a DATETIME (**mi_datetime**) value is an Informix-proprietary format, you cannot use standard system functions to perform operations on **mi_datetime** values. Instead, the DataBlade API provides the following support for the DATETIME data type.

Category of DATETIME Function	More Information
Conversion functions	"Conversion of Date-Time or Interval Representations" on page 4-13
Arithmetic-operation functions	"Operations on Date and Time Data" on page 4-15

The INTERVAL Data Type

The SQL INTERVAL data type holds the internal (binary) format of an interval value. It encodes a value that represents a span of time. INTERVAL types are divided into two classes: *year-month intervals* and *day-time intervals*. A year-month interval can represent a span of years and months, and a day-time interval can represent a span of days, hours, minutes, seconds, and fractions of a second. For a detailed description of the SQL INTERVAL data type, see the *IBM Informix Guide to SQL: Reference*.

The INTERVAL data type uses a computer-independent method to encode the interval qualifiers. It stores the information in the **intrvl_t** structure, as follows:

```
typedef struct intrvl {
   short in_qual;
   dec_t in_dec;
} intrvl_t;
```

The **intrvl** structure and **intrvl_t** typedef have the two parts that Table 4-1 shows.

Table 4-1. Fields in the intrvl_t Structure

Field	Description
in_qual	The qualifier of the **interval** value

Table 4-1. Fields in the intrvl_t Structure (continued)

Field	Description
in_dec	The digits of the fields of the **interval** value
	This field is a **decimal** value.

Tip: The internal format of the INTERVAL data type is often referred to as its binary representation.

The DataBlade API supports the SQL INTERVAL data type with the **mi_interval** data type. Therefore, an **mi_interval** data type holds the binary representation of an interval value.

Server Only

Values of the **mi_interval** data type *cannot* fit into an **MI_DATUM** structure. They must be passed by reference within C UDRs.

End of Server Only

Client Only

All data types, including **mi_interval**, must be passed by reference within client LIBMI applications.

End of Client Only

Because the binary representation of an INTERVAL (**mi_interval**) value is an Informix-proprietary format, you cannot use standard system functions to perform operations on **mi_interval** values. Instead, the DataBlade API provides the following support for the INTERVAL data type.

Category of INTERVAL Function	More Information
Conversion functions	"Conversion of Date-Time or Interval Representations" on page 4-13
Arithmetic-operation functions	"Operations on Date and Time Data" on page 4-15

The datetime.h Header File

The **datetime.h** header file contains definitions for use with the DATETIME and INTERVAL data types. The header file **datetime.h** contains the declarations for the date, time, and interval data types, as follows:

- The internal format represents DATETIME and **mi_datetime** values with the **dtime_t** structure.
- The internal format represents INTERVAL and **mi_interval** values with the **intrvl_t** structure.

In addition to these data structures, the **datetime.h** file defines the constants and macros for date and time qualifiers that Table 4-2 shows.

Table 4-2. Qualifier Macros and Constants for mi_datetime and mi_interval Data Types

Name of Macro	Description
TU_YEAR	The time unit for the YEAR qualifier field

Table 4-2. Qualifier Macros and Constants for mi_datetime and mi_interval Data Types (continued)

Name of Macro	Description
TU_MONTH	The time unit for the MONTH qualifier field
TU_DAY	The time unit for the DAY qualifier field
TU_HOUR	The time unit for the HOUR qualifier field
TU_MINUTE	The time unit for the MINUTE qualifier field
TU_SECOND	The time unit for the SECOND qualifier field
TU_FRAC	The time unit for the leading qualifier field of FRACTION
TU_Fn	The names for **mi_datetime** ending fields of FRACTION(n), for n from 1 to 5
TU_START(q)	Returns the leading field number from qualifier q
TU_END(q)	Returns the trailing field number from qualifier q
TU_LEN(q)	Returns the length in digits of the qualifier q
TU_FLEN(f)	Returns the length in digits of the first field, f, of an **interval** qualifier
TU_ENCODE(p,f,t)	Creates a qualifier from the first field number f with precision p and trailing field number t
TU_DTENCODE(f,t)	Creates an **mi_datetime** qualifier from the first field number f and trailing field number t
TU_IENCODE(p,f,t)	Creates an **mi_interval** qualifier from the first field number f with precision p and trailing field number t

Tip: For a complete list of date and time macros, consult the **datetime.h** header file that is installed with your database server. This header file resides in the **incl/public** subdirectory of the **INFORMIXDIR** directory.

Table 4-2 on page 4-9 shows the macro definitions that you can use to compose qualifier values. You need these macros *only* when you work directly with qualifiers in binary form. For example, if your program does not provide an **mi_interval** qualifier in the variable declaration, you need to use the **mi_interval** qualifier macros to initialize and set the **mi_interval** variable, as the following example shows:

```
/* Use the variable that was declared intvl1. */
   mi_interval intvl1;
...
/* Set the interval qualifier for the variable */
intvl1.in_qual = TU_IENCODE(2, TU_DAY, TU_SECOND);
...
/* Assign a value to the variable */
incvasc ("5 2:10:02", &intvl1);
```

In the previous example, the **mi_interval** variable gets a **day to second** qualifier. The precision of the largest field in the qualifier, **day**, is set to 2.

In addition to the declaration of the **dtime_t typedef** and the preceding date and time macros, the **datetime.h** header file declares the DATETIME-type functions of the Informix ESQL/C library. The **mitypes.h** header file automatically includes **datetime.h**. In turn, the **milib.h** header file automatically includes **mitypes.h** and **mi.h** automatically includes **milib.h**. Therefore, you automatically have access to the **dtime_t** and **intrvl_t** structures, the **mi_datetime** and **mi_interval** data types,

any of the date or time macros, or any of the Informix ESQL/C DATETIME-type functions when you include **mi.h** in your DataBlade API module.

Retrieval and Insertion of DATETIME and INTERVAL Values

When an application retrieves or inserts a DATETIME or INTERVAL value, the DataBlade API module must ensure that the qualifier field of the variable is valid:

- When an application fetches a DATETIME value into an **mi_datetime** variable or inserts a DATETIME value from an **mi_datetime** variable, the application must ensure that the **dt_qual** field of the **dtime_t** structure is valid.
- When an application fetches an INTERVAL value into an **mi_interval** variable or inserts an INTERVAL value from an **mi_interval** variable, the application must ensure that the **in_qual** field of the **intrvl_t** structure is valid.

Fetch or Insert into an mi_datetime Variable

When a DataBlade API module uses an **mi_datetime** variable to fetch or insert a DATETIME value, the module must find a valid qualifier in the **mi_datetime** variable. The DataBlade API takes one the following actions, based on the value of the **dt_qual** field in the **dtime_t** structure that is associated with the variable:

- When the **dt_qual** field contains a valid qualifier, the DataBlade API extends the column value to match the **dt_qual** qualifier.

  Extending is the operation of adding or dropping fields of a DATETIME value to make it match a given qualifier. You can explicitly extend DATETIME values with the SQL EXTEND function and the **dtextend( )** function.

- When the **dt_qual** field does *not* contain a valid qualifier, the DataBlade API takes different actions for a fetch and an insert:

  - For a fetch, the DataBlade API uses the DATETIME column value and its qualifier to initialize the **mi_datetime** variable.

    Zero is an invalid qualifier. Therefore, if you set the **dt_qual** field to zero, you can ensure that the DataBlade API uses the qualifier of the DATETIME column.

  - For an insert, the DataBlade API cannot perform the insert or update operation.

    The DataBlade API sets the SQLSTATE status variable to an error-class code (and SQLCODE to a negative value) and the update or insert operation on the DATETIME column fails.

Fetch or Insert into an mi_interval Variable

When a DataBlade API module uses an **mi_interval** variable to fetch or insert an INTERVAL value, the DataBlade API must find a valid qualifier in the **mi_interval** variable. The DataBlade API takes one of the following actions, based on the value of the **in_qual** field the **intrvl_t** structure that is associated with the variable:

- When the **in_qual** field contains a valid qualifier, the DataBlade API checks it for compatibility with the qualifier from the INTERVAL column value.

  The two qualifiers are compatible if they belong to the *same* interval class: either **year to month** or **day to fraction**. If the qualifiers are incompatible, the DataBlade API sets the **SQLSTATE** status variable to an error-class code (and **SQLCODE** is set to a negative value) and the select, update, or insert operation fails.

  If the qualifiers are compatible but not the same, the DataBlade API extends the column value to match the **in_qual** qualifier. *Extending* is the operation of adding or dropping fields within one of the interval classes of an INTERVAL value to make it match a given qualifier. You can explicitly extend INTERVAL values with the **invextend( )** function.

- When the **in_qual** field does *not* contain a valid qualifier, the DataBlade API takes different actions for a fetch and an insert:
  - For a fetch, if the **in_qual** field contains zero or is not a valid qualifier, the DataBlade API uses the INTERVAL column value and its qualifier to initialize the **mi_interval** variable.
  - For an insert, if the **in_qual** field is not compatible with the INTERVAL column or if it does not contain a valid value, the DataBlade API cannot perform the insert or update operation.

    The DataBlade API sets the **SQLSTATE** status variable to an error-class code (and **SQLCODE** is set to a negative value) and the update or insert operation on the INTERVAL column fails.

Implicit Data Conversion

You can select a DATETIME or INTERVAL column value into a character variable. The DataBlade API converts the DATETIME or INTERVAL column value to a character string before it stores it in the character variable. This character string conforms to the ANSI SQL standards for DATETIME and INTERVAL values.

Important: IBM Informix products do not support automatic data conversion from DATETIME and INTERVAL column values to numeric (**mi_double_precision**, **mi_integer**, and so on) variables.

You can also insert a DATETIME or INTERVAL column value from a character variable. The DataBlade API uses the data type and qualifiers of the column value to convert the character value to a DATETIME or INTERVAL value. It expects the character string to contain a DATETIME or INTERVAL value that conforms to ANSI SQL standards.

If the conversion fails, the DataBlade API sets the **SQLSTATE** status variable to an error-class code (and **SQLCODE** status variable to a negative value) and the update or insert operation fails.

Important: IBM Informix products do not support automatic data conversion from numeric and **mi_date** variables to DATETIME and INTERVAL column values.

Transfers of Date-Time or Interval Data (Server)

For date-time or interval values to be portable when transferred across different computer architectures, the DataBlade API provides the following functions to handle type alignment and byte order.

DataBlade API Function	Description
mi_get_datetime( )	Copies an aligned **mi_datetime** value, converting any difference in alignment or byte order on the client computer to that of the server computer
mi_get_interval( )	Copies an aligned **mi_interval** value, converting any difference in alignment or byte order on the client computer to that of the server computer
mi_put_datetime( )	Copies an aligned **mi_datetime** value, converting any difference in alignment or byte order on the server computer to that of the client computer
mi_put_interval( )	Copies an aligned **mi_interval** value, converting

any difference in alignment or byte order on the
server computer to that of the client computer

The **mi_get_datetime( )**, **mi_get_interval( )**, **mi_put_datetime( )**, and
mi_put_interval( ) functions are useful in the send and receive support function of
an opaque data type that contains **mi_datetime** or **mi_interval** values. They allow
you to ensure that DATETIME or INTERVAL values remained aligned when
transferred to and from client applications. For more information, see "Conversion
of Opaque-Type Data with Computer-Specific Data Types" on page 16-21.

Conversion of Date-Time or Interval Representations

Both the DataBlade API library and the Informix ESQL/C library provide functions
that convert from the text (string) representation of a date, time, or interval value
to the binary (internal) representation for DATETIME or INTERVAL, respectively.

DataBlade API Functions for Date-Time or Interval Conversion

The DataBlade API provides the following functions for conversion between text
and binary representations of date-time or interval data.

DataBlade API Function	Convert from	Convert to
mi_datetime_to_string( )	DATETIME (**mi_datetime**)	Date-time string
mi_interval_to_string( )	INTERVAL (**mi_interval**)	Interval string
mi_string_to_datetime( )	Date-time string	DATETIME (**mi_datetime**)
mi_string_to_interval( )	Interval string	INTERVAL (**mi_interval**)

The **mi_datetime_to_string( )**, **mi_interval_to_string( )**, **mi_string_to_datetime( )**,
and **mi_string_to_interval( )** functions convert DATETIME and INTERVAL values
to and from the ANSI SQL standards formats for these data types.

──────────────── **Server Only** ────────────────

The **mi_datetime_to_string( )**, **mi_interval_to_string( )**, **mi_string_to_datetime( )**,
and **mi_string_to_interval( )** functions are useful in the input and output support
functions of an opaque data type that contains **mi_datetime** and **mi_interval**
values, as long as these values use the ANSI SQL formats. They enable you to
convert DATETIME and INTERVAL values between their external format (text) and
their internal (binary) format when transferred to and from client applications. For
more information, see "Conversion of Opaque-Type Data Between Text and Binary
Representations" on page 16-16.

──────────────── **End of Server Only** ────────────────

ESQL/C Functions for Date, Time, and Interval Conversion

The Informix ESQL/C function library provides functions for conversion between
text and binary representations of date, time, and interval data.

Data Conversion for DATETIME Values: The Informix ESQL/C library provides
the following functions that convert internal DATETIME (**mi_datetime**) values to
and from **char** strings.

Function Name	Description

dtcvasc( )	Converts an ANSI-compliant character string to an **mi_datetime** value
dtcvfmtasc( )	Converts a character string to an **mi_datetime** value
dtextend( )	Changes the qualifier of an **mi_datetime** value
dttoasc( )	Converts an **mi_datetime** value to an ANSI-compliant character string
dttofmtasc( )	Converts an **mi_datetime** value to a character string

The **dttoasc( )** and **dtcvasc( )** functions convert **mi_datetime** values to and from the ANSI SQL standard values for DATETIME strings. The ANSI SQL standards specify qualifiers and formats for character representations of DATETIME and INTERVAL values. The standard qualifier for a DATETIME value is YEAR TO SECOND, and the standard format is as follows:

```
YYYY-MM-DD HH:MM:SS
```

The **dttofmtasc( )** and **dtcvfmtasc( )** functions convert **mi_datetime** values to and from a date-time string using a time-formatting mask. This time-formatting mask contains the same formatting directives that the **DBTIME** environment variable supports. (For a list of these directives, see the description of **DBTIME** in the *IBM Informix Guide to SQL: Reference*.)

The **dtextend( )** function extends an **mi_datetime** value to a different qualifier. You can use it to convert between DATETIME and DATE values.

To convert a DATETIME value to a DATE value:

1. Use **dtextend( )** to adjust the DATETIME qualifier to *year to day*.
2. Apply **dttoasc( )** to create a character string in the form *yyyy-mm-dd*.
3. Use **rdefmtdate( )** with a pattern argument of *yyyy-mm-dd* to convert the string to a DATE value.

To convert a DATE value into a DATETIME value:

1. Declare a variable with a qualifier of *year to day* (or initialize the qualifier with the value that the TU_DTENCODE (TU_YEAR,TU_DAY) macro returns).
2. Use **rfmtdate( )** with a pattern of *yyyy-mm-dd* to convert the DATE value to a character string.
3. Use **dtcvasc( )** to convert the character string to a value in the prepared DATETIME variable.
4. If necessary, use **dtextend( )** to adjust the DATETIME qualifier.

Data Conversion for INTERVAL Values: The Informix ESQL/C library provides the following functions that convert internal INTERVAL (**mi_interval**) values to and from **char** text.

Function Name	Description
incvasc( )	Converts an ANSI-compliant character string to an **interval** value
incvfmtasc( )	Converts a character string to an **interval** value
intoasc( )	Converts an **interval** value to an ANSI-compliant character string

intofmtasc( )	Converts an **interval** value to a string
invextend( )	Copies an **interval** value under a different qualifier

The **intoasc( )** and **incvasc( )** functions convert **mi_interval** values to and from the ANSI SQL standards for INTERVAL strings. The ANSI SQL standards specify qualifiers and formats for character representations of DATETIME and INTERVAL values. The standards for an INTERVAL value specify the following two *classes* of intervals:

- The YEAR TO MONTH class has the following format:

  `YYYY-MM`

  A subset of this format is also valid: for example, just a month interval.

- The DAY TO FRACTION class has the following format:

  `DD HH:MM:SS.F`

  Any subset of contiguous fields is also valid: for example, MINUTE TO FRACTION.

The **intofmtasc( )** and **incvfmtasc( )** functions convert **mi_interval** values to and from an interval string using a time-formatting mask. This time-formatting mask contains the same formatting directives that the **DBTIME** environment variable supports. (For a list of these directives, see the description of **DBTIME** in the *IBM Informix Guide to SQL: Reference*.)

Operations on Date and Time Data

The Informix ESQL/C library provides the following functions to perform operations on DATETIME (**mi_datetime**) and INTERVAL (**mi_interval**) values.

Function Name	Description
dtaddinv( )	Adds an **mi_interval** value to a **mi_datetime** value
dtcurrent( )	Gets current date and time
dtsub( )	Subtracts one **mi_datetime** value from another
dtsubinv( )	Subtracts an **mi_interval** value from a **mi_datetime** value
invdivdbl( )	Divides an **mi_interval** value by a numeric value
invdivinv( )	Divides an **mi_interval** value by an **mi_interval** value
invmuldbl( )	Multiplies an **mi_interval** value by a numeric value

Any other operations, modifications, or analyses can produce unpredictable results.

Functions to Obtain Information on Date and Time Data

Table 4-3 shows the DataBlade API functions that obtain qualifier information for a DATETIME (**mi_datetime**) or INTERVAL (**mi_interval**) value.

Table 4-3. DataBlade API Functions That Obtain DATETIME or INTERVAL Information

Source	DataBlade API Functions
For a data type	**mi_type_qualifier( )**, **mi_type_precision( )**, **mi_type_scale( )**

Table 4-3. DataBlade API Functions That Obtain DATETIME or INTERVAL Information (continued)

Source	DataBlade API Functions
For a UDR argument	**mi_fp_argprec( )**, **mi_fp_setargprec( )** **mi_fp_argscale( )**, **mi_fp_setargscale( )**
For a UDR return value	**mi_fp_retprec( )**, **mi_fp_setretprec( )** **mi_fp_retscale( )**, **mi_fp_setretscale( )**
For a column in a row (or field in a row type)	**mi_column_precision( )**, **mi_column_scale( )**
For an input parameter in a prepared statement	**mi_parameter_precision( )**, **mi_parameter_scale( )**

Suppose you have a table with a single column, **dt_col**, of type DATETIME YEAR TO SECOND. If **row_desc** is a row descriptor for a row in this table, the code fragment in Figure 4-1 obtains the name, qualifier, precision, and scale for this column value.

```
MI_TYPE_DESC *col_type_desc;
MI_ROW_DESC *row_desc;
mi_string *type_name;
mi_integer type_qual;
...
col_type_desc = mi_column_typedesc(row_desc, 0);
type_name = mi_type_typename(col_type_desc);
type_qual = mi_type_qualifier(col_type_desc);
type_prec = mi_type_precision(col_type_desc);
type_scale = mi_type_scale(col_type_desc);
sprintf(type_buf,
   "column=%d: type name=%s, qualifier=%d precision=%d \
   scale=%d\n",
   i, type_name, type_qual, type_prec, type_scale);
```

Figure 4-1. Obtaining Type Information for a DATETIME Value

In Figure 4-1, the value in the **type_buf** buffer would be as follows:

```
column=0, type name=datetime year to second, qualifier=3594 precision=14 scale=10
```

Qualifier of a Date-Time or Interval Data Type

The **mi_type_qualifier( )** function returns the encoded qualifier of a DATETIME or INTERVAL data type from a type descriptor. This qualifier is the internal value that the database server uses to track the complete qualifier range, from the starting field to the end field. It is the value stored in the **collength** column of the **syscolumns** table for DATETIME and INTERVAL columns. You can use the qualifier macros and constants (see Table 4-2 on page 4-9) to interpret this encoded value.

In Figure 4-1, the value in **type_qual** contains the encoded integer qualifier (3594) for the **dt_col** column. You can obtain the starting qualifier for the DATETIME value from the encoded qualifier with the TU_START macro, as follows:

```
TU_START(type_qual)
```

This TU_START call yields 0, which is the value of the TU_YEAR constant in the **datetime.h** header file. You can obtain also the ending qualifier for the DATETIME value from the encoded qualifier with the TU_END macro, as follows:

```
TU_END(type_qual)
```

This TU_END call yields 10, which is the value of the TU_SECOND constant in the **datetime.h** header file. Therefore, the encoded qualifier 3594 represents the qualifier *year to second*.

Precision of a Date-Time or Interval Data Type

For the DATETIME and INTERVAL data types, the *precision* is the number of digits required to store a value with the specified qualifier. In Figure 4-1, the call to the **mi_type_precision( )** function saves in **type_prec** the precision for the **dt_col** column from its type descriptor. This precision has a value of 14 because a DATETIME YEAR TO SECOND value requires 14 digits:

```
YYYYMMDDHHMMSS
```

YYYY	is the 4-digit year.
MM	is the 2-digit month.
DD	is the 2-digit day of the month.
HH	is the 2-digit hour.
MM	is the 2-digit minute.
SS	is the 2-digit second.

The DataBlade API also provides functions that obtain DATETIME or INTERVAL precision of a column associated with an input parameter, a UDR argument, UDR return value, or a row column. For a list of these functions, see Table 4-3 on page 4-15.

Scale of a Date-Time or Interval Data Type

For the DATETIME and INTERVAL data types, the *scale* is the encoded integer value for the end qualifier. In Figure 4-1, the call to the **mi_type_scale( )** function stores in **type_scale** the scale for the **dt_col** column. This precision has a value of 10 because the end qualifier for the DATETIME YEAR TO SECOND data type is SECOND, whose encoded value (TU_SECOND) is 10.

The DataBlade API also provides functions that obtain DATETIME or INTERVAL scale of an input parameter, a UDR argument, UDR return value, or column. For a list of these functions, see Table 4-3 on page 4-15.

Chapter 5. Using Complex Data Types

In This Chapter

The DataBlade API provides support for the following complex data types.

Complex Data Type	DataBlade API Data Type
Collection data types: • LIST • MULTISET • SET	**MI_COLLECTION, MI_COLL_DESC**
Row types: • Named • Unnamed	**MI_ROW, MI_ROW_DESC**

This chapter describes these complex data types as well as the functions that the DataBlade API supports to process collection and row-type data.

Collections

A *collection* is a complex data type that is made up of *elements*, each of which has the same data type. A collection is similar to an array in the C language. The DataBlade API provides support for collections in both their text and binary representations.

Collection Text Representation

The DataBlade API supports a collection in text representation as a quoted string with the following format:

`"coll_type{elmnt_value, elmnt_value, ...}"`

coll_type   is the type of the collection: SET, MULTISET, or LIST.

elmnt_value   is the text representation of the element value.

A collection in its text representation is often called a *collection string*. For example, the following collection string provides the text representation for a SET of integer values:

`"SET{1, 6, 8, 3}"`

For a complete description of the text representation of a collection, see the description of the Literal Collection segment in the *IBM Informix Guide to SQL: Syntax*.

Collection Binary Representation

The database server supports the following kinds of collections.

Collection Data Type	Description
LIST	An ordered group of elements that can contain duplicate elements
MULTISET	An unordered group of elements that can contain duplicate elements
SET	An unordered group of elements that cannot contain duplicate elements

All collection data types use the same internal format to store their values. For more information on collection data types, see the *IBM Informix Guide to SQL: Reference*.

Tip: The internal format of a collection data type is often referred to as its binary representation.

The DataBlade API supports the following SQL collection data types and data type structures:

- A *collection structure* (**MI_COLLECTION**) holds the binary representation of the collection.
- A *collection descriptor* (**MI_COLL_DESC**) provides information about the collection.

Using a Collection Structure

A *collection structure*, **MI_COLLECTION**, is a DataBlade API structure that holds the collection (LIST, MULTISET, or SET) and its elements. The following table summarizes the memory operations for a collection structure.

Memory Duration	Memory Operation	Function Name
Current memory duration	Constructor	**mi_collection_copy( )**, **mi_collection_create( )**, **mi_streamread_collection( )**
	Destructor	**mi_collection_free( )**

The following DataBlade API functions return an existing collection structure.

DataBlade API Function	Description
mi_value( ), **mi_value_by_name( )**	Returns a collection structure as a column value when the function returns an MI_COLLECTION_VALUE value status

Using a Collection Descriptor

A *collection descriptor*, **MI_COLL_DESC**, is a DataBlade API structure that contains a collection cursor to access elements of a collection. The following table summarizes the memory operations for a collection descriptor.

Memory Duration	Memory Operation	Function Name
Current memory duration	Constructor	**mi_collection_open( )**, **mi_collection_open_with_options( )**
	Destructor	**mi_collection_close( )**

Important: To a DataBlade API module, the collection descriptor (**MI_COLL_DESC**) is an opaque C data structure. Do not access its internal fields directly. The internal structure of a collection descriptor may change in future releases. Therefore, to create portable code, always use the functions that access collection elements.

Creating a Collection

To create a collection, use the **mi_collection_create( )** function. The **mi_collection_create( )** function is the constructor function for the collection structure (**MI_COLLECTION**). The collection structure includes the type of collection (LIST, MULTISET, or SET) and the element data type.

The following code shows an example of how to use the **mi_collection_create( )** function to create a new list of integers:

```
/*
 * Create a LIST collection with INTEGER elements
 */
  MI_CONNECTION *conn;
  MI_TYPEID *typeid;
  MI_COLLECTION *coll;

  typeid = mi_typestring_to_id(conn, "list(integer not null)");

  if ( typeid != NULL )
     {
     coll = mi_collection_create(conn, typeid);
  ...
```

Opening a Collection

Once you have a collection structure for a collection, you can open the collection with one of the functions in Table 5-1.

Table 5-1. DataBlade API Functions To Open a Collection

DataBlade API Function	Use
mi_collection_open( )	Opens a collection in a read/write scroll cursor
mi_collection_open_with_options( )	Opens a collection in either of the following open modes: • Read only • Nonscrolling

Both of the functions in Table 5-1 are constructor functions for a collection descriptor. Use this collection descriptor in calls to DataBlade API functions that access the collection.

When one of the functions in Table 5-1 opens a collection, it creates a *collection cursor*, which is an area of memory that serves as a holding place for collection elements. This cursor has an associated *cursor position*, which points to one element of the collection cursor. When these functions complete, the cursor position points to the *first* element of the collection.

The difference between the **mi_collection_open( )** and **mi_collection_open_with_options( )** functions is the *open mode* that they create for the collection cursor.

Using mi_collection_open( )

When you open a collection with **mi_collection_open( )**, you obtain an update scroll cursor to hold the collection elements. Therefore, you can perform the following operations on a collection opened with **mi_collection_open( )**.

Cursor Attribute	Valid Operations
Read/write cursor	Insert, delete, update, fetch
Scroll cursor	Fetch forward and backward through the collection elements All Fetch operations are valid. (See Table 5-2 on page 5-6)

Figure 5-1 shows an example of using the **mi_collection_open( )** function to create and open a LIST collection with INTEGER elements.

```
/*
 * Create and open a collection
 */
MI_CONNECTION *conn;
MI_COLL_DESC *coll_desc;
MI_COLLECTION *coll_ptr;
MI_TYPEID *type_id;
...
type_id = mi_typestring_to_id(conn, "list(integer not null)");
coll_ptr = mi_collection_create(conn, type_id);
coll_desc = mi_collection_open(conn, coll_ptr);
```

Figure 5-1. Opening a LIST (INTEGER) Collection

Figure 5-2 shows the cursor position after the **mi_collection_open( )** call.
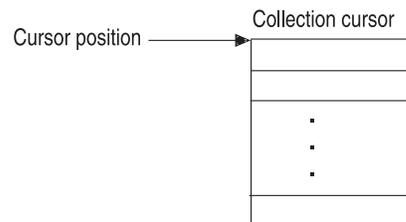
Figure 5-2. Collection Cursor After the Collection Is Opened

Using mi_collection_open_with_options( )

When you open a collection with **mi_collection_open_with_options( )**, you can override the cursor characteristics that **mi_collection_open( )** uses. The *control* argument of **mi_collection_open_with_options( )** can create a collection cursor with any of the cursor characteristics in the following table.

Cursor Attribute	Control Flag	Valid Operations
Read-only cursor	MI_COLL_READONLY	Fetch *only*
Sequential (nonscrolling) cursor	MI_COLL_NOSCROLL	Fetch forward *only* (MI_CURSOR_NEXT) through the collection elements Any fetch operation that moves the cursor position *backward* in the cursor is *not* valid.

Most collections need the capabilities of the read/write scroll cursor that **mi_collection_open( )** creates. However, the database server can perform a special optimization for a collection from a collection subquery if you use a read-only sequential cursor to hold the collection subquery. It can fetch each row of the subquery on demand. That is, you can fetch the elements one at a time with **mi_collection_fetch( )**. You can use **mi_collection_open( )** or **mi_collection_open_with_options( )** to create some other type of cursor for a collection subquery. However, if a collection subquery resides in some other type of cursor, the database server fetches *all* the rows of the subquery and puts them in the collection cursor.

To create a collection subquery, preface the query with the MULTISET keyword. For example, the following SQL statement creates a collection subquery of order numbers for customer 120 and then sends them to the **check_orders( )** user-defined function (which expects a MULTISET argument):

```
        SELECT check_orders(
           MULTISET(SELECT ITEM order_num FROM orders
              WHERE customer_num = 120))
        FROM customer
        WHERE customer_num = 120;
```

To have the database server perform the collection-subquery optimization, use the following call to **mi_collection_open_with_options( )** when you open a collection subquery:

```
mi_collection_open_with_options(conn, coll_ptr,
   (MI_COLL_READONLY | MI_COLL_NOSCROLL));
```

Accessing Elements of a Collection

The DataBlade API provides the following functions for accessing collection data types.

DataBlade API Collection Function	Description
mi_collection_copy( )	Creates a copy of an existing open collection
mi_collection_delete( )	Deletes an element from a collection
mi_collection_fetch( )	Fetches an element from a collection
mi_collection_insert( )	Inserts a new element into an open collection
mi_collection_update( )	Updates an element in an open collection

Positioning the Cursor

When you open a collection cursor with **mi_collection_open( )**, the cursor position points to the first element of the collection. The cursor position identifies the current element in the collection cursor. The DataBlade API functions that access a collection must specify where in the collection to perform the operation. To specify location, these functions all have an *action* argument of type **MI_CURSOR_ACTION**, which supports the cursor-action constants in Table 5-2.

Table 5-2. Valid Cursor-Action Constants

Cursor Movement	Cursor-Action Constant	Valid Cursor Types	
		Sequential	Scroll
Move the cursor position one element forward within the cursor	MI_CURSOR_NEXT	Yes	Yes
Move the cursor position one element backward within the cursor	MI_CURSOR_PRIOR	No	Yes
Move the cursor position to the beginning of the cursor, at the first element	MI_CURSOR_FIRST	*Only* if the cursor position does not move backward	Yes
Move the cursor position to the end of the cursor, at the last element	MI_CURSOR_LAST	Yes	Yes
Move the cursor to the absolute position within the cursor, where the first element in the cursor is at position 1.	MI_CURSOR_ABSOLUTE	Yes	Yes
		As long as collection is a LIST because only LISTs have ordered elements	

Table 5-2. Valid Cursor-Action Constants (continued)

Cursor Movement	Cursor-Action Constant	Valid Cursor Types	
		Sequential	Scroll
Move the cursor forward or back a specified number of elements from the current position.	MI_CURSOR_RELATIVE	*Only* if relative position is a positive value	Yes
			Relative position can be a negative or positive value
		As long as collection is a LIST because only LISTs have ordered elements	
Leave the cursor position at its current location.	MI_CURSOR_CURRENT	Yes	Yes

The following code fragment uses the **mi_collection_fetch( )** function to fetch a VARCHAR element from a collection:

```
/*
 * Fetch next VARCHAR( ) element from a collection.
 */

   MI_CONNECTION *conn;
   MI_COLL_DESC *colldesc;
   MI_ROW_DESC  *rowdesc;
   MI_COLLECTION *nest_collp;
   MI_DATUM value;
   mi_integer ret_code, ret_len;
   char *buf;

   /* Fetch a VARCHAR( ) type */
   ret_code = mi_collection_fetch(conn, colldesc,
      MI_CURSOR_NEXT, 0, &value, &ret_len);

   switch ( ret_code )
   {
   case MI_NORMAL_VALUE:
      buf = mi_get_vardata((mi_lvarchar *)value);
      DPRINTF("trace_class", 15, ("Value: %s", buf));
      break;

   case MI_NULL_VALUE:
      DPRINTF("trace_class", 15, ("NULL"));
      break;

   case MI_ROW_VALUE:
      rowdesc = (MI_ROW_DESC *)value;
      break;

   case MI_COLLECTION_VALUE:
      nested_collp = (MI_COLLECTION *)value;
      break;

   case MI_END_OF_DATA:
      DPRINTF("trace_class", 15,
         ("End of collection reached"));
      return (100);
   }
```

Inserting an Element

You insert an element into an open collection with the **mi_collection_insert( )** function. You can perform an insert operation *only* on a read/write cursor. An insert is *not* valid on a read-only cursor.

The **mi_collection_insert( )** function uses an **MI_DATUM** value to represent an element that it inserts into a collection. The contents of the **MI_DATUM** structure depend on the passing mechanism that the function used, as follows:

- In a C user-defined routine (UDR), when **mi_collection_insert( )** inserts an element value, it can pass the value by reference or by value, depending on the data type of the column value. If the function passes the element value by value, the **MI_DATUM** structure contains the value. If the function passes the element value by reference, the **MI_DATUM** structure contains a pointer to the value.

- In a client LIBMI application, when **mi_collection_insert( )** inserts an element value, it *always* passes the value in an **MI_DATUM** structure by reference. Even for values that you can pass by value in a C UDR (such as an INTEGER values), this function passes the element value by reference. The **MI_DATUM** structure contains a pointer to the value.

The **mi_collection_insert( )** function inserts the new element at the location in the collection cursor that its *action* argument specifies. For a list of valid cursor-action flags, see Table 5-2 on page 5-6.

The following call to **mi_collection_insert( )** can pass in an actual value because it inserts an INTEGER element into a LIST collection and integer values are passed by value in a C UDR:

```
MI_CONNECTION *conn;
MI_DATUM datum;
MI_COLL_DESC *colldesc;

datum=6;
mi_collection_insert(conn, colldesc, datum,
   MI_CURSOR_ABSOLUTE, 1);

datum=3;
mi_collection_insert(conn, colldesc, datum,
   MI_CURSOR_ABSOLUTE, 2);

datum=15;
mi_collection_insert(conn, colldesc, datum,
   MI_CURSOR_ABSOLUTE, 3);

datum=1;
mi_collection_insert(conn, colldesc, datum,
   MI_CURSOR_ABSOLUTE, 4);

datum=4;
mi_collection_insert(conn, colldesc, datum,
   MI_CURSOR_ABSOLUTE, 5);

datum=8;
mi_collection_insert(conn, colldesc, datum,
   MI_CURSOR_ABSOLUTE, 6);
```

Figure 5-3 shows the cursor position after the preceding calls to
mi_collection_insert( ) complete.

Collection cursor

6
3
15
1
4
8

Cursor position ⟶ 8
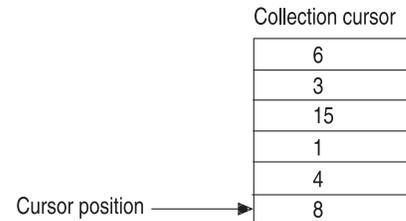
Figure 5-3. Collection Cursor After Inserts Complete

These **mi_collection_insert( )** calls specify absolute addressing
(MI_CURSOR_ABSOLUTE) for the collection because the collection is defined as a
LIST. Only LIST collections have ordered position assigned to their elements. SET
and MULTISET collections do not have ordered position of elements.

Fetching an Element

You fetch an element from an open collection with the **mi_collection_fetch( )**
function. You can perform a fetch operation on a read/write or a read-only cursor.
To fetch a collection element, you must specify:
- The connection with which the collection is associated
- The collection descriptor for the collection from which you want to fetch
  elements
- The location of the cursor position at which to begin the fetch
- A variable that holds a single fetched element and one that holds its length

Moving Through a Cursor: The **mi_collection_fetch( )** function obtains the
element specified by its *action* argument from the collection cursor. For a list of
valid cursor-action flags, see Table 5-2 on page 5-6. You can move the cursor
position back to the beginning of the cursor with the **mi_collection_fetch( )**
function, as the following example shows:

```
mi_collection_fetch(conn, coll_desc, MI_CURSOR_FIRST, 0,
   coll_element, element_len);

if ( ((mi_integer)coll_element != 1) ||
     (element_len != sizeof(mi_integer)) )
   /* raise an error */
```

This function moves the cursor position backward with respect to its position after
a call to **mi_collection_insert( )** (Figure 5-3 on page 5-9). The
mi_collection_fetch( ) function is valid only for the following kinds of cursors:
- Sequential collection cursors, if the cursor position does not move backward
- Scroll collection cursors

  Only scroll cursors provide the ability to move the cursor position forward and
  backward.

Figure 5-4 shows the cursor position and **coll_element** value after the preceding
call to **mi_collection_fetch( )**.
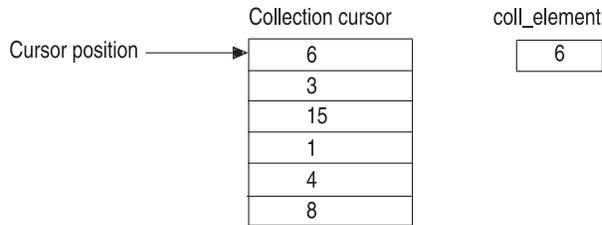
Figure 5-4. Collection Cursor After Fetch First

Figure 5-5 shows the cursor position and value of **coll_element** after the following **mi_collection_fetch( )** call:

```
mi_collection_fetch(conn, coll_desc, MI_CURSOR_NEXT, 0,
   coll_element, element_len);
```
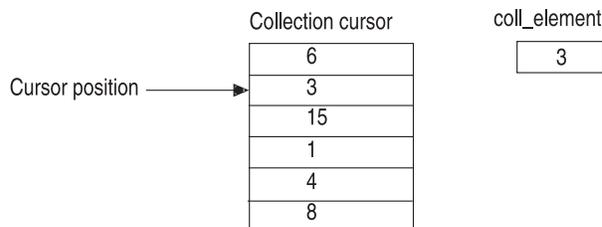

Figure 5-5. Collection Cursor After Fetch Next

Figure 5-6 shows the cursor position and value of **coll_element** after the following **mi_collection_fetch( )** call:

```
mi_collection_fetch(conn, coll_desc, MI_CURSOR_RELATIVE, 3,
   coll_element, element_len);
```
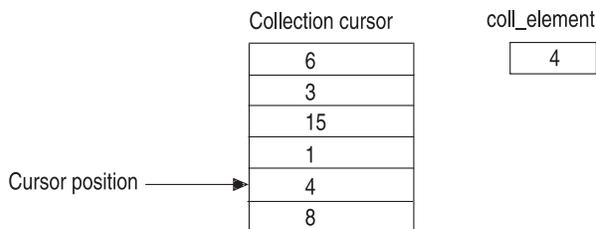

Figure 5-6. Collection Cursor After Fetch Relative 3

The preceding **mi_collection_fetch( )** call is valid *only* if the collection is a LIST. Only LIST collections are ordered. Therefore relative fetches, which specify the number of elements to move forward or backward, can only be used on LIST collections. If you try to perform a relative fetch on a SET or MULTISET, **mi_collection_fetch( )** generates an error.

Figure 5-7 shows the cursor position and value of **coll_element** after the following **mi_collection_fetch( )** call:

```
mi_collection_fetch(conn, coll_desc, MI_CURSOR_RELATIVE, -2,
   coll_element, element_len);
```
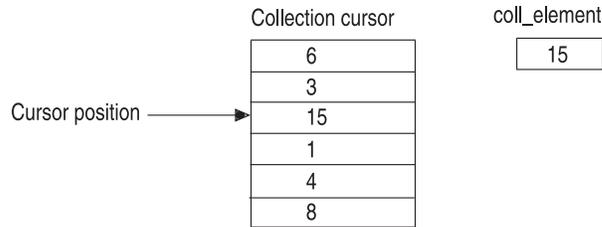
Figure 5-7. Collection Cursor After Fetch Relative -2

Because the preceding **mi_collection_fetch( )** call moves the cursor position backward, the call is valid *only* if the collection cursor is a scroll cursor. When you open a collection with **mi_collection_open( )**, you get a read/write scroll collection cursor. However, if you open the collection with **mi_collection_open_with_options( )** and the MI_COLL_NOSCROLL option, **mi_collection_fetch( )** generates an error.

Figure 5-8 shows the cursor position and value of **coll_element** after the following **mi_collection_fetch( )** call:

```
mi_collection_fetch(conn, coll_desc, MI_CURSOR_ABSOLUTE, 6,
   coll_element, element_len);
```
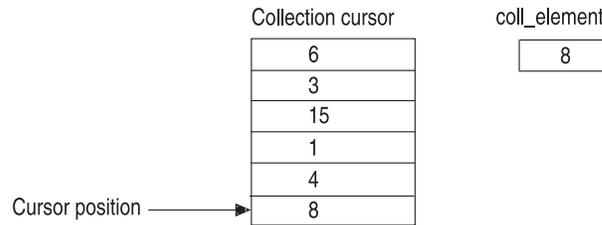

Figure 5-8. Collection Cursor After Fetch Absolute 6

The preceding **mi_collection_fetch( )** call is valid *only* if the collection is a LIST. Because absolute fetches specify a position within the collection by number, they can only be used on an ordered collection (a LIST). If you try to perform an absolute fetch on a SET or MULTISET, **mi_collection_fetch( )** generates an error.

Because only six elements are in this collection, the absolute fetch of 6 positions the cursor on the *last* element in the collection. This result is the same as if you had issued the following **mi_collection_fetch( )**:

```
mi_collection_fetch(conn, coll_desc, MI_CURSOR_LAST, 0,
   coll_element, element_len);
```

The fetch last is useful when you do not know the number of elements in a collection and want to obtain the last one.

Obtaining the Element Value: The **mi_collection_fetch( )** function uses an **MI_DATUM** value to represent an element that it fetches from a collection. You must pass in a pointer to the value buffer in which **mi_collection_fetch( )** puts the element value. However, you do not have to allocate memory for this buffer. The **mi_collection_fetch( )** function handles memory allocation for the **MI_DATUM** value that it passes back.

The contents of the **MI_DATUM** structure that holds the retrieved element depends on the passing mechanism that the function used, as follows:

```
───────────────────────── Server Only ─────────────────────────
```

- In a C UDR, when **mi_collection_fetch( )** passes back an element value, it passes back the value by reference or by value, depending on the data type of the column value. If the function passes back the element value by value, the **MI_DATUM** structure contains the value. If the function passes back the element value by reference, the **MI_DATUM** structure contains a pointer to the value.

```
──────────────────────── End of Server Only ───────────────────
```

```
───────────────────────── Client Only ─────────────────────────
```

- In a client LIBMI application, when **mi_collection_fetch( )** passes back an element value, it *always* passes back the value by reference. Even for values that you can pass by value in a C UDR (such as an INTEGER value), this function passes back the element value by reference. The **MI_DATUM** structure contains a pointer to the value.

```
──────────────────────── End of Client Only ───────────────────
```

Important: The difference in behavior of **mi_collection_fetch( )** between C UDRs and client LIBMI applications means that collection-retrieval code is not completely portable between these two types of DataBlade API modules. When you move your DataBlade API code from one of these uses to another, you must change the collection-retrieval code to use the appropriate passing mechanism for element values that **mi_collection_fetch( )** returns.

You declare a value buffer for the fetched element and pass in the address of this buffer to **mi_collection_fetch( )**. You can declare the buffer in either of the following ways:

- If you know the data type of the field value, declare the value buffer of this data type.

  Declare the value buffer as a pointer to the field data type, regardless of whether the data type is passed by reference or by value.

- If you do *not* know the data type of the field value, declare the value buffer to have the **MI_DATUM** data type.

  Your code can dynamically determine the field type with the **mi_column_type_id( )** or **mi_column_typedesc( )** function. You can then convert (or cast) the **MI_DATUM** value to a data type that you need.

Figures 5-4 through 5-8 fetch elements from a LIST collection of INTEGER values. To fetch elements from this LIST, you can declare the value buffer as follows:

```
mi_integer *coll_element;
```

```
───────────────────────── Server Only ─────────────────────────
```

Because you can pass INTEGER values by value in a C UDR, you access the **MI_DATUM** structure that these calls to **mi_collection_fetch( )** pass back as the actual value, as follows:

```
int_element = (mi_integer)coll_element;
```

If the element type is a data type that must be passed by reference, the contents of the **MI_DATUM** structure that **mi_collection_fetch( )** passes back is a pointer to the actual value. The following call to **mi_collection_fetch( )** also passes in the value buffer as a pointer. However, it passes back an **MI_DATUM** value that contains a pointer to a FLOAT (**mi_double_precision**) value:

```
mi_double_precision *coll_element, flt_element;
...
/* Fetch a FLOAT value in a C UDR */
mi_collection_fetch(conn, coll_desc, action, jump,
    &coll_element, &retlen);
flt_element = *coll_element;
```

─────────────── **End of Server Only** ───────────────

─────────────── **Client Only** ───────────────

For the fetches in Figures 5-4 through 5-8, a client LIBMI application declares the value buffer in the same way as a C UDR. However, because all data types are passed back by reference, the **MI_DATUM** structure that **mi_collection_fetch( )** passes back contains a pointer to the INTEGER value, not the actual value itself:

```
mi_integer *coll_element, int_element;
...
/* Fetch an INTEGER value in a client LIBMI application */
mi_collection_fetch(conn, coll_desc, action, jump,
    &coll_element, &retlen);
int_element = *coll_element;
```

─────────────── **End of Client Only** ───────────────

Updating a Collection

You update an element in an open collection with the **mi_collection_update( )** function. You can perform an update operation *only* on a read/write cursor. An update is *not* valid on a read-only cursor.

The **mi_collection_update( )** function uses an **MI_DATUM** value to represent the new value for the element it updates in a collection. The contents of this **MI_DATUM** structure depend on the passing mechanism that the function used, as follows:

─────────────── **Server Only** ───────────────

- In a C UDR, when **mi_collection_update( )** updates an element value, it can pass the value by reference or by value, depending on the data type of the column value. If the function passes back the element value by value, the **MI_DATUM** structure contains the value. If the function passes back the element value by reference, the **MI_DATUM** structure contains a pointer to the value.

─────────────── **End of Server Only** ───────────────

─────────────── **Client Only** ───────────────

- In a client LIBMI application, when **mi_collection_update( )** updates an element value, it *always* passes the value by reference. Even for values that you can pass by value in a C UDR (such as an INTEGER value), these functions return the

column value by reference. The **MI_DATUM** structure contains a pointer to the value.

──────────────────── **End of Client Only** ────────────────────

The **mi_collection_update( )** function updates the element at the location in the collection cursor that its *action* argument specifies. For a list of valid cursor-action flags, see Table 5-2 on page 5-6.

┌──────────────────── **Server Only** ────────────────────

The following code shows an example of using the **mi_collection_update( )** function to update the first element in a collection:

```
/*
 * Update position 1 in the collection to contain 3.0
 * Note that single-precision value is passed by REFERENCE.
 */
  MI_CONNECTION *conn;
  MI_COLL_DESC *colldesc;
  MI_DATUM val;
  mi_integer ret, jump;
  mi_real value;

  /* Update 1st element to 3.0 */
  value = 3.0;
  val = (MI_DATUM)&value;
  jump = 1;
  DPRINTF("trc_class", 11,
     ("Update set value %d @%d", value, jump));

  /* Pass single-precision values by reference */
  ret = mi_collection_update(conn, colldesc, val,
     MI_CURSOR_ABSOLUTE, jump);

  if ( ret != MI_OK )
     {
     DPRINTF("trc_class", 11,
        ("Update @%d value %d MI_CURSOR_ABSOLUTE\
        failed", jump, value));
     }
```

──────────────────── **End of Server Only** ────────────────────

Deleting an Element

You delete an element from an open collection with the **mi_collection_delete( )** function. You can perform a delete operation *only* on a read/write cursor. A delete is not valid on a read-only cursor.

The **mi_collection_delete( )** function deletes the element at the location in the collection cursor that its *action* argument specifies. For a list of valid cursor-action flags, see Table 5-2 on page 5-6.

The following code shows an example of using the **mi_collection_delete( )** function to delete the last element of a collection:

```
/*
 * Delete last element in the collection
 */
  MI_CONNECTION *conn;
  MI_COLL_DESC *coll_desc;
```

```
   mi_integer ret;

   ret = mi_collection_delete(conn, coll_desc,
      MI_CURSOR_LAST, 0);
```

Determining the Cardinality of a Collection

The DataBlade API provides the **mi_collection_card( )** function for obtaining the
number of elements in a collection (its cardinality). The following code fragment
uses the **mi_collection_card( )** function to perform separate actions based on
whether a collection is NULL or has elements (possibly 0 elements):

```
MI_COLLECTION *collp;
mi_integer cardinality;
mi_boolean isnull;


/* Attach collp to a collection */


cardinality = mi_collection_card(collp, &isnull);
if ( isnull == MI_TRUE )
   {
   mi_db_error_raise(conn, MI_MESSAGE, "Warning: Collection is NULL.");
   }
else
   {
   if ( cardinality > 0 )
      {
      /*  Open collection and perform action on individual elements */
      }
   }
```

Releasing Collection Resources

When your DataBlade API module no longer needs a collection, you can release
the resources that it uses with the following DataBlade API functions.

DataBlade API Function	Purpose
mi_collection_close( )	Closes the collection cursor and frees the collection descriptor.
mi_collection_free( )	Frees the collection structure

Closing a Collection

A collection descriptor contains a collection cursor. The scope of the collection
descriptor and its associated collection cursor is from the time they are created, by
mi_collection_open_with_options( ) or **mi_collection_open( )**, until one of the
following events occurs:

• The **mi_collection_close( )** function frees the collection descriptor, thereby
  closing and freeing the associated collection cursor.

───────────────────── **Server Only** ─────────────────────

• The current memory duration expires.

───────────────── **End of Server Only** ─────────────────

• The **mi_close( )** function closes the connection.

To conserve resources, use the **mi_collection_close( )** function to free the collection
descriptor as soon as your DataBlade API module no longer needs it. This function
also explicitly closes and frees the associated collection cursor. The

mi_collection_close( ) function is the destructor function for the collection descriptor as well as for its associated cursor.

Freeing the Collection Structure

The collection structure holds the collection elements. The scope of this structure is from the time it is created, by **mi_collection_create( )** or **mi_collection_copy( )**, until one of the following events occurs:

- The **mi_collection_free( )** function frees the collection structure.

```
─────────────────── Server Only ───────────────────
```

- The current memory duration expires.

```
─────────────── End of Server Only ───────────────
```

- The **mi_close( )** function closes the connection.

To conserve resources, use the **mi_collection_free( )** function to free the collection structure once your DataBlade API module no longer needs it. The **mi_collection_close( )** function is the destructor function for the collection structure.

The listpos( ) UDR

The sample **listpos( )** UDR consists of the following parts:

- The SQL statements that register the function, create a table, and run the **listpos( )** user-defined function
- The C code to implement the **listpos( )** UDR
- Sample output from the **listpos.trc** trace file that the **listpos( )** UDR generates

SQL Statements

The SQL statements for the following tasks handle the database objects that the **listpos( )** function requires:

1. Register the user-defined function named **listpos( )**:

```
CREATE FUNCTION listpos( )
RETURNS INTEGER
EXTERNAL NAME '$USERFUNCDIR/sql_listpos.udr'
LANGUAGE C;
```

2. Create a table named **tab2**:

```
CREATE TABLE tab2 (a INT);
INSERT INTO tab2 VALUES (1);
```

3. Add the trace class that the DPRINTF statements in **listpos( )** use:

```
INSERT INTO informix.systraceclasses(name)
    VALUES ('trace_class');
```

4. Run the **listpos( )** UDR:

```
SELECT listpos( ) FROM tab2;
```

5. Clean up the resources:

```
DROP FUNCTION listpos;
DROP TABLE tab2;
```

C-Language Implementation

The following C file contains the functions that implement the **listpos( )** user-defined function:

```
/* C file (listpos.c) contents:
 *   Examples of mi_collection_*( ) functions
 */
```

```
#include <stdio.h>
#include <mi.h>
#include <sqltypes.h>

void do_fetch(
      MI_CONNECTION *conn,
      MI_COLL_DESC *colldesc,
      MI_CURSOR_ACTION action,
      mi_integer type,
      mi_integer jump,
      MI_DATUM expected);

mi_integer create_collection(
      MI_CONNECTION *conn,
      char *typestring,
      MI_COLLECTION **ret_coll_struc,
      MI_COLL_DESC **ret_coll_desc);

mi_integer list_int_ins(MI_CONNECTION *conn);
mi_integer list_char_ins(MI_CONNECTION *conn);
mi_integer list_float_ins(MI_CONNECTION *conn);


/********************************************************
 * Function: The listpos( ) user-defined routine
 * Purpose: Run inserts on three types of LIST collections:
 *    LIST of INTEGER: list_int_ins( )
 *    LIST of CHAR: list_char_ins( )
 *    LIST of FLOAT: list_float_ins( )
 *    Results are printed to a trace file named 'listpos.trc',
 *    which is the file that the mi_tracefile_set( ) function
 *    specifies.
 * Return Values:
 *     0    Success
 *    -1    No valid connection descriptor
 *    -50   Unable to convert data type to type identifer
 *    -51   Unable to create specified collection
 *    -52   Unable to open new collection
 */
mi_integer listpos( )
{
    MI_CONNECTION *conn;
    mi_integer ret_code, error;

/* Obtain a UDR connection descriptor and verify that it
 * is valid
 */
   conn = mi_open(NULL, NULL, NULL);
   if ( conn == NULL )
      return (-1);

/* Turn on tracing of trace class "trace_class" and set the
 * trace file to listpos.trc.
 */
   mi_tracelevel_set("trace_class 20");
   mi_tracefile_set("/usr/local/udrs/colls/listpos.trc");

/* Run list_int_ins( ) to insert INTEGER values into the LIST */
   error = 0;
   ret_code = list_int_ins(conn);
   if ( ret_code )
      error = ret_code;

/* Run list_char_ins( ) to insert CHAR values into the LIST */
   list_char_ins(conn);
   if ( ret_code )
      error = ret_code;
```

```
                /* Run list_float_ins( ) to insert FLOAT values into the LIST  */
                   list_float_ins(conn);
                   if ( ret_code )
                      error = ret_code;

                   return (ret_code);
                } /* end listpos( ) */


                /*********************************************************
                 * Function: list_int_ins( )
                 * Purpose:
                 *   1. insert 3 INTEGER values into a LIST
                 *   2. verify each inserted value
                 *   3. update first element
                 * Return Values:
                 *     0    Success
                 *   -50    Unable to convert data type to type identifer
                 *   -51    Unable to create specified collection
                 *   -52    Unable to open new collection
                 *    (status of steps in trace file)
                 */
                mi_integer list_int_ins(MI_CONNECTION *conn)
                {
                   MI_COLLECTION *list;
                   MI_COLL_DESC *colldesc;

                   MI_CURSOR_ACTION action;
                   mi_integer jump, value, ret_code;

                /* Create the LIST of INTEGERs */
                   ret_code = create_collection(conn, "list(int not null)",
                      &list, &colldesc);
                   if ( ret_code != 0 )
                      return (ret_code);

                   action = MI_CURSOR_ABSOLUTE;

                /* Insert three INTEGER values
                 *   position 1: 1
                 *   position 2: 2
                 *   position 3: 3
                 * INTEGER datums are passed by value. Normally one would use
                 * an action of MI_CURSOR_NEXT (jump is ignored), but this
                 * function inserts at positions.
                 */
                   value = jump = 1;
                   DPRINTF("trace_class", 15,
                      ("Insert %d into LIST of INTEGER @%d", value,
                         jump));
                   ret_code = mi_collection_insert(conn, colldesc,
                      (MI_DATUM) value, action, jump);
                   if ( ret_code != MI_OK )
                      {
                      DPRINTF("trace_class", 15,
                   ("list_int_ins: insert MI_CURSOR_ABSOLUTE %d @%d failed",
                      value, jump));
                      }

                   value = jump = 2;
                   DPRINTF("trace_class", 15,
                      ("Insert %d into LIST of INTEGER @%d", value,
                         jump));
                   ret_code = mi_collection_insert(conn, colldesc,
                      (MI_DATUM) value, action, jump);
                   if ( ret_code != MI_OK )
```

```
            {
            DPRINTF("trace_class", 15,
        ("list_int_ins: insert MI_CURSOR_ABSOLUTE %d @%d failed",
          value, jump));
            }

      value = jump = 3;
      DPRINTF("trace_class", 15,
          ("Insert %d into LIST of INTEGER @%d", value,
              jump));
      ret_code = mi_collection_insert(conn, colldesc,
          (MI_DATUM) value, action, jump);
      if ( ret_code != MI_OK )
            {
            DPRINTF("trace_class", 15,
        ("list_int_ins: insert MI_CURSOR_ABSOLUTE %d @%d failed",
          value, jump));
            }

/* Fetch each inserted INTEGER value from the collection,
 * comparing it against the value actually inserted.
 * Use a jump equal to the data value to simplify the
 * validation.
 */
    dofetch(conn, colldesc, MI_CURSOR_ABSOLUTE, SQLINT, 1,
        (MI_DATUM) 1);
    dofetch(conn, colldesc, MI_CURSOR_ABSOLUTE, SQLINT, 3,
        (MI_DATUM) 3);
    dofetch(conn, colldesc, MI_CURSOR_ABSOLUTE, SQLINT, 2,
        (MI_DATUM) 2);
    dofetch(conn, colldesc, MI_CURSOR_PRIOR, SQLINT, 1,
        (MI_DATUM) 1);
    dofetch(conn, colldesc, MI_CURSOR_LAST, SQLINT, 3,
        (MI_DATUM) 3);
    dofetch(conn, colldesc, MI_CURSOR_FIRST, SQLINT, 1,
        (MI_DATUM) 1);
    dofetch(conn, colldesc, MI_CURSOR_RELATIVE, SQLINT, 2,
        (MI_DATUM) 3);
    dofetch(conn, colldesc, MI_CURSOR_RELATIVE, SQLINT, -2,
        (MI_DATUM) 1);

/* Update 1st element to 3. */
    jump=1;
    value=3;
    DPRINTF("trace_class", 15,
        ("Update %d into LIST of INTEGER @%d", value,
          jump));
    ret_code = mi_collection_update(conn, colldesc,
        (MI_DATUM) value, action, jump);
    if ( ret_code != MI_OK )
            {
            DPRINTF("trace_class", 15,
        ("list_int_ins: update MI_CURSOR_ABSOLUTE @%d failed",
          jump));
            }

/* Fetch the updated element back and validate it */
    dofetch(conn, colldesc, MI_CURSOR_ABSOLUTE, SQLINT, 1,
        (MI_DATUM) 3);

/* Free collection resources */
    mi_collection_close(conn, colldesc);
    mi_collection_free(conn, list);

    return 0;
} /* end list_int_ins( ) */
```

```
/********************************************************
 * Function: list_float_ins( )
 * Purpose:
 *   1. insert 3 FLOAT values into a LIST
 *   2. verify each inserted value
 *   3. update first element
 * Return Values:
 *     0    Success
 *   -50    Unable to convert data type to type identifer
 *   -51    Unable to create specified collection
 *   -52    Unable to open new collection
 *   (status of steps in trace file)
 */
mi_integer list_float_ins(MI_CONNECTION *conn)
{
   MI_COLLECTION *list;
   MI_COLL_DESC *colldesc;

   MI_CURSOR_ACTION action;
   mi_integer jump, value, ret_code;
   mi_double_precision val1, val2, val3, val4;

/* Create the LIST of FLOATs */
   ret_code = create_collection(conn,
      "list(float not null)", &list, &colldesc);
   if ( ret_code != 0 )
      return (ret_code);

   action = MI_CURSOR_ABSOLUTE;

/* Insert three FLOAT values
 *   position 1: 1.1
 *   position 2: -2.2
 *   position 3: 3.3
 * FLOAT datums are passed by reference.
 */
   val1 = 1.1;
   val2 = -2.2;
   val3 = 3.3;

   jump = 1;
   DPRINTF("trace_class", 15,
      ("Insert %f into LIST of FLOAT @%d", val1, jump));
   ret_code = mi_collection_insert(conn, colldesc,
      (MI_DATUM) &val1, action, jump);
   if ( ret_code != MI_OK )
      {
      DPRINTF("trace_class", 15,
("list_float_ins: insert MI_CURSOR_ABSOLUTE %f @%d failed",
      val1, jump));
      }

   jump = 2;
   DPRINTF("trace_class", 15,
      ("Insert %f into LIST of FLOAT @%d", val2, jump));
   ret_code = mi_collection_insert(conn, colldesc,
      (MI_DATUM) &val2, action, jump);
   if ( ret_code != MI_OK )
      {
      DPRINTF("trace_class", 15,
("list_float_ins: insert MI_CURSOR_ABSOLUTE %f @%d failed",
      val2, jump));
      }

   jump = 3;
   DPRINTF("trace_class", 15,
```

```
            ("Insert %f into LIST of FLOAT @%d", val3, jump));
       ret_code = mi_collection_insert(conn, colldesc,
           (MI_DATUM) &val3, action, jump);
       if ( ret_code != MI_OK )
           {
           DPRINTF("trace_class", 15,
   ("list_float_ins: insert MI_CURSOR_ABSOLUTE %f @%d failed",
           val3, jump));
           }

   /* Fetch each inserted FLOAT value from the collection,
    * comparing it against the value actually inserted.
    */
      dofetch(conn, colldesc, MI_CURSOR_ABSOLUTE, SQLFLOAT, 1,
          (MI_DATUM) &val1);
      dofetch(conn, colldesc, MI_CURSOR_ABSOLUTE, SQLFLOAT, 3,
          (MI_DATUM) &val3);
      dofetch(conn, colldesc, MI_CURSOR_ABSOLUTE, SQLFLOAT, 2,
          (MI_DATUM) &val2);
      dofetch(conn, colldesc, MI_CURSOR_PRIOR, SQLFLOAT, 1,
          (MI_DATUM) &val1);
      dofetch(conn, colldesc, MI_CURSOR_LAST, SQLFLOAT, 3,
          (MI_DATUM) &val3);
      dofetch(conn, colldesc, MI_CURSOR_FIRST, SQLFLOAT, 1,
          (MI_DATUM) &val1);
      dofetch(conn, colldesc, MI_CURSOR_RELATIVE, SQLFLOAT, 2,
          (MI_DATUM) &val3);
      dofetch(conn, colldesc, MI_CURSOR_RELATIVE, SQLFLOAT, -2,
          (MI_DATUM) &val1);

   /* Update 1st element to 44E-4. */
      jump=1;
      val4=44e-4;
      DPRINTF("trace_class", 15,
          ("Update %f into LIST of FLOAT @%d", val4, jump));
      ret_code = mi_collection_update(conn, colldesc,
          (MI_DATUM) &val4, action, jump);
      if ( ret_code != MI_OK )
          {
          DPRINTF("trace_class", 15,
   ("list_float_ins: update MI_CURSOR_ABSOLUTE @%d failed",
          jump));
          }

   /* Fetch the updated element back and validate it */
      dofetch(conn, colldesc, MI_CURSOR_ABSOLUTE, SQLFLOAT, 1,
          (MI_DATUM) &val4);

   /* Free collection resources */
      mi_collection_close(conn, colldesc);
      mi_collection_free(conn, list);

      return 0;
   } /* end list_float_ins( ) */


   /*****************************************************
    * Function: list_char_ins( )
    * Purpose:
    *   1. insert 3 CHAR values into a LIST
    *   2. verify each inserted value
    *   3. update first element
    * Return Values:
    *    0    Success
    *   -50   Unable to convert data type to type identifer
    *   -51   Unable to create specified collection
    *   -52   Unable to open new collection
```

```
 *     (status of steps in trace file)
 */
mi_integer list_char_ins(MI_CONNECTION *conn)
{
   MI_COLLECTION *list;
   MI_COLL_DESC *colldesc;

   MI_CURSOR_ACTION action;
   MI_DATUM val;
   mi_integer retlen, jump, ret_code;
   mi_lvarchar  *lvc;
   char *buf;
   char *val1, *val2, *val3;

/* Create the LIST of CHAR(10)s */
   ret_code = create_collection(conn,
      "list(char(10) not null)", &list, &colldesc);
   if ( ret_code != 0 )
      return (ret_code);

   action = MI_CURSOR_ABSOLUTE;

/* Insert three CHAR(10) values:
 *    position 1: "1234567689"
 *    position 2: "abcdefghij"
 *    position 3: "three"
 * CHAR datums are passed by reference in an mi_lvarchar
 * structure.
 */
   val1 = "1234567689";
   val2 = "abcdefghij";
   val3 = "three";

   lvc = mi_new_var(10);
   buf = mi_get_vardata(lvc);

   jump = 1;
   strcpy(buf, val1);
   DPRINTF("trace_class", 15,
      ("Insert '%s' into LIST of CHAR @%d",
       buf, jump));
   ret_code = mi_collection_insert(conn, colldesc,
      (MI_DATUM)lvc, action, jump);
   if ( ret_code != MI_OK )
      {
      DPRINTF("trace_class", 15,
   ("list_char_ins: insert MI_CURSOR_ABSOLUTE @%d failed",
      jump));
      }

   jump = 2;
   strcpy(buf, val2);
   DPRINTF("trace_class", 15,
      ("Insert '%s' into LIST of CHAR @%d",
       buf, jump));
   ret_code = mi_collection_insert(conn, colldesc,
      (MI_DATUM)lvc, action, jump);
   if ( ret_code != MI_OK )
      {
      DPRINTF("trace_class", 15,
   ("list_char_ins: insert MI_CURSOR_ABSOLUTE @%d failed",
      jump));
      }

   jump = 3;
   strcpy(buf, val3);
   DPRINTF("trace_class", 15,
```

```
                        ("Insert '%s' into LIST of CHAR @%d",
                         buf, jump));
          ret_code = mi_collection_insert(conn, colldesc,
                (MI_DATUM)lvc, action, jump);
          if ( ret_code != MI_OK )
                {
                DPRINTF("trace_class", 15,
          ("list_char_ins: insert MI_CURSOR_ABSOLUTE @%d failed",
                jump));
                }

     /* Fetch each inserted CHAR value from the collection,
      * comparing it against the value actually inserted.
      */
        dofetch(conn, colldesc, MI_CURSOR_ABSOLUTE, SQLCHAR, 1,
                val1);
        dofetch(conn, colldesc, MI_CURSOR_ABSOLUTE, SQLCHAR, 3,
                val3);
        dofetch(conn, colldesc, MI_CURSOR_ABSOLUTE, SQLCHAR, 2,
                val2);
        dofetch(conn, colldesc, MI_CURSOR_PRIOR, SQLCHAR, 1,
                val1);
        dofetch(conn, colldesc, MI_CURSOR_LAST, SQLCHAR, 3,
                val3);
        dofetch(conn, colldesc, MI_CURSOR_FIRST, SQLCHAR, 1,
                val1);
        dofetch(conn, colldesc, MI_CURSOR_RELATIVE, SQLCHAR, 2,
                val3);
        dofetch(conn, colldesc, MI_CURSOR_RELATIVE, SQLCHAR, -2,
                val1);

     /* Update 1st element to "mnopqrstuv". */
        jump=1;
        strcpy(buf, "mnopqrstuv");
        DPRINTF("trace_class", 15,
                ("Update '%s' into LIST of CHAR @ %d", buf, jump));
        ret_code = mi_collection_update(conn, colldesc,
                (MI_DATUM)lvc, action, jump);
        if ( ret_code != MI_OK )
                {
                DPRINTF("trace_class", 15,
          ("list_char_ins: update MI_CURSOR_ABSOLUTE @%d failed",
                jump));
                }

     /* Fetch the updated element back and validate it */
        dofetch(conn, colldesc, MI_CURSOR_FIRST, SQLCHAR, 1,
                buf);

     /* Free collection resources */
        mi_collection_close(conn, colldesc);
        mi_collection_free(conn, list);

        return 0;
} /* end list_char_ins( ) */


/*****************************************************
 * Function: do_fetch( )
 * Purpose: Fetch specified element from a collection and
 *              compare it with the specified expected value
 * Return Values: NONE
 */
void do_fetch(
        MI_CONNECTION *conn,
        MI_COLL_DESC *colldesc,
        MI_CURSOR_ACTION action,
```

```
                     mi_integer type,
                     mi_integer jump,
                     MI_DATUM expected)
{
   MI_DATUM val;
   mi_integer retlen, ret_code;
   char *actionstr, *buf;

   switch ( action )
   {
      case MI_CURSOR_NEXT:
         actionstr="MI_CURSOR_NEXT";
         break;

      case MI_CURSOR_PRIOR:
         actionstr="MI_CURSOR_PRIOR";
         break;

      case MI_CURSOR_FIRST:
          actionstr="MI_CURSOR_FIRST";
          break;

      case MI_CURSOR_LAST:
         actionstr="MI_CURSOR_LAST";
          break;

      case MI_CURSOR_ABSOLUTE:
           actionstr="MI_CURSOR_ABSOLUTE";
         break;

      case MI_CURSOR_RELATIVE:
           actionstr="MI_CURSOR_RELATIVE";
         break;

      default:
         actionstr="UNKNOWN";
   }

   DPRINTF("trace_class", 15,
      ("Fetch %s @ jump=%d:", actionstr, jump));

/* Print what is the expected value */
   switch ( type )
   {
      case SQLINT:
         DPRINTF("trace_class", 15,
            (" should get %d: ", expected));
         break;

      case SQLCHAR:
         DPRINTF("trace_class", 15,
            (" should get '%s': ", expected));
         break;

      case SQLFLOAT:
         DPRINTF("trace_class", 15,
            (" should get %f: ", *(double *)expected));
         break;

      default:
         DPRINTF("trace_class", 15,
            (" type not handled: %d", type));
   }

/* Fetch collection element at position 'jump' into 'val' */
   ret_code = mi_collection_fetch(conn, colldesc, action,
      jump, &val, &retlen);
```

```
      if ( ret_code != MI_NORMAL_VALUE )
         {
         DPRINTF("trace_class", 15,
            ("do_fetch: %s @%d failed", actionstr, jump));
         return;
         }

/* Compare fetched value with expected value */
   switch ( type )
   {
      case SQLINT:
         if ( expected != val )
            {
            DPRINTF("trace_class", 15,
         ("do_fetch: fetch value not expected; got %d",
            val));
            }
         else
            {
            DPRINTF("trace_class", 15,
               ("  got %d, fetch succeeded", val));
            }
         break;

      case SQLCHAR:
         buf = mi_get_vardata((mi_lvarchar *)val);
         if ( strcmp(buf, (char *)expected) != 0 )
            {
            DPRINTF("trace_class", 15,
         ("do_fetch: fetch value not expected; got %s",
            buf));
            }
         else
            {
            DPRINTF("trace_class", 15,
               ("  got '%s', fetch succeeded", buf));
            }
         break;

      case SQLFLOAT:
         if ( *(double *)expected != *(double *)val )
            {
            DPRINTF("trace_class", 15,
         ("do_fetch: fetch value not expected; got %f",
            *(double *)val));
            }
         else
            {
            DPRINTF("trace_class", 15,
               ("  got %f, fetch succeeded",
                *(double *)val));
            }
         break;

      default:
         DPRINTF("trace_class", 15,
            ("do_fetch: %d type not handled", type));
   }
} /* end do_fetch( ) */


/********************************************************
 * Function: create_collection( )
 * Purpose: create a collection of the specified type
 * Return Values:
 *   thru parameters:
 *      ret_coll_desc: address of collection descriptor
```

```
 *      ret_coll_struc: address of collection structure
 *   thru return value:
 *       0    Success
 *     -50    Unable to convert data type to type identifer
 *     -51    Unable to create specified collection
 *     -52    Unable to open new collection
 */
mi_integer create_collection(
      MI_CONNECTION *conn,
      char *typestring,
      MI_COLLECTION **ret_coll_struc,
      MI_COLL_DESC **ret_coll_desc)
{
   MI_TYPEID *typeid;
   MI_COLLECTION *collstruc;
   MI_COLL_DESC *colldesc;

/* Convert data type string to type identifier */
   typeid = mi_typestring_to_id(conn, typestring);
   if ( typeid == NULL )
      {
      DPRINTF("trace_class", 15,
   ("create_collection: mi_typestring_to_id( ) failed"));
      return (-50);
      }

/* Create collection whose elements have the data type
 * indicated by the specified type identifer
 */
   if ( (collstruc =
        mi_collection_create(conn, typeid)) == NULL )
      {
      DPRINTF("trace_class", 15,
   ("create_collection: mi_collection_create( ) failed"));
      return (-51);
      }

/* Open the collection */
   if ( (colldesc =
        mi_collection_open(conn, collstruc)) == NULL )
      {
      DPRINTF("trace_class", 15,
        ("mi_collection_open( ) failed"));
      return -52;
      }

/* Return through the parameters the addresses of:
 *    the collection descriptor: ret_coll_desc
 *    the collection structure: ret_coll_struc
 */
   *ret_coll_desc = colldesc;
   *ret_coll_struc = collstruc;

/* Return a status of zero to indicate success */
   return 0;
/* end create_collection( ) */
```

Sample listpos( ) Trace Output

When the **listpos( )** user-defined function executes successfully, it produces the
following output in the **listpos.trc** file:

```
===============================================================

Tracing session: 18 on 03/16/2000

13:12:24  Insert 1 into LIST of INTEGER @1
13:12:24  Insert 2 into LIST of INTEGER @2
```

```
13:12:24  Insert 3 into LIST of INTEGER @3
13:12:24  Fetch MI_CURSOR_ABSOLUTE @ jump=1:
13:12:24    should get 1
13:12:24    got 1, fetch succeeded
13:12:24  Fetch MI_CURSOR_ABSOLUTE @ jump=3:
13:12:24    should get 3
13:12:24    got 3, fetch succeeded
13:12:24  Fetch MI_CURSOR_ABSOLUTE @ jump=2:
13:12:24    should get 2
13:12:24    got 2, fetch succeeded
13:12:24  Fetch MI_CURSOR_PRIOR @ jump=1:
13:12:24    should get 1
13:12:24    got 1, fetch succeeded
13:12:24  Fetch MI_CURSOR_LAST @ jump=3:
13:12:24    should get 3
13:12:24    got 3, fetch succeeded
13:12:24  Fetch MI_CURSOR_FIRST @ jump=1:
13:12:24    should get 1
13:12:24    got 1, fetch succeeded
13:12:24  Fetch MI_CURSOR_RELATIVE @ jump=2:
13:12:24    should get 3
13:12:24    got 3, fetch succeeded
13:12:24  Fetch MI_CURSOR_RELATIVE @ jump=-2:
13:12:24    should get 1
13:12:24    got 1, fetch succeeded
13:12:24  Update 3 into LIST of INTEGER @1
13:12:24  Fetch MI_CURSOR_ABSOLUTE @ jump=1:
13:12:24    should get 3
13:12:24    got 3, fetch succeeded
13:12:24  Insert '1234567689' into LIST of CHAR @1
13:12:24  Insert 'abcdefghij' into LIST of CHAR @2
13:12:24  Insert 'three' into LIST of CHAR @3
13:12:24  Fetch MI_CURSOR_ABSOLUTE @ jump=1:
13:12:24    should get '1234567689'
13:12:24    got '1234567689', fetch succeeded
13:12:24  Fetch MI_CURSOR_ABSOLUTE @ jump=3:
13:12:24    should get 'three'
13:12:24    got 'three', fetch succeeded
13:12:24  Fetch MI_CURSOR_ABSOLUTE @ jump=2:
13:12:24    should get 'abcdefghij'
13:12:24    got 'abcdefghij', fetch succeeded
13:12:24  Fetch MI_CURSOR_PRIOR @ jump=1:
13:12:24    should get '1234567689'
13:12:24    got '1234567689', fetch succeeded
13:12:24  Fetch MI_CURSOR_LAST @ jump=3:
13:12:24    should get 'three'
13:12:24    got 'three', fetch succeeded
13:12:24  Fetch MI_CURSOR_FIRST @ jump=1:
13:12:24    should get '1234567689'
13:12:24    got '1234567689', fetch succeeded
13:12:24  Fetch MI_CURSOR_RELATIVE @ jump=2:
13:12:24    should get 'three'
13:12:24    got 'three', fetch succeeded
13:12:24  Fetch MI_CURSOR_RELATIVE @ jump=-2:
13:12:24    should get '1234567689'
13:12:24    got '1234567689', fetch succeeded
13:12:24  Update 'mnopqrstuv' into LIST of CHAR @1
13:12:24  Fetch MI_CURSOR_FIRST @ jump=1:
13:12:24    should get 'mnopqrstuv'
13:12:24    got 'mnopqrstuv', fetch succeeded
13:12:24  Insert 1.100000 into LIST of FLOAT @1
13:12:24  Insert -2.200000 into LIST of FLOAT @2
13:12:24  Insert 3.300000 into LIST of FLOAT @3
13:12:24  Fetch MI_CURSOR_ABSOLUTE @ jump=1:
13:12:24    should get 1.100000
13:12:24    got 1.100000, fetch succeeded
13:12:24  Fetch MI_CURSOR_ABSOLUTE @ jump=3:
```

```
13:12:24    should get 3.300000
13:12:24    got 3.300000, fetch succeeded
13:12:24  Fetch MI_CURSOR_ABSOLUTE @ jump=2:
13:12:24    should get -2.200000
13:12:24    got -2.200000, fetch succeeded
13:12:24  Fetch MI_CURSOR_PRIOR @ jump=1:
13:12:24    should get 1.100000
13:12:24    got 1.100000, fetch succeeded
13:12:24  Fetch MI_CURSOR_LAST @ jump=3:
13:12:24    should get 3.300000
13:12:24    got 3.300000, fetch succeeded
13:12:24  Fetch MI_CURSOR_FIRST @ jump=1:
13:12:24    should get 1.100000
13:12:24    got 1.100000, fetch succeeded
13:12:24  Fetch MI_CURSOR_RELATIVE @ jump=2:
13:12:24    should get 3.300000
13:12:24    got 3.300000, fetch succeeded
13:12:24  Fetch MI_CURSOR_RELATIVE @ jump=-2:
13:12:24    should get 1.100000
13:12:24    got 1.100000, fetch succeeded
13:12:24  Update 0.004400 into LIST of FLOAT @1
13:12:24  Fetch MI_CURSOR_ABSOLUTE @ jump=1:
13:12:24    should get 0.004400
13:12:24    got 0.004400, fetch succeeded
```

Row Types

A *row type* is a complex data type that is made up of a sequence of one or more elements called *fields*. Each field has a name and a data type. A row type is similar to a C **struct** data type. The DataBlade API provides support for row types in both their text and binary representations.

Row-Type Text Representation

The DataBlade API supports a text representation for row types as a quoted string with the formats that the following table shows.

Row Type	Text Representation
Unnamed	"ROW(*fld_value1*, *fld_value2*, ...)"
Named	"*row_type*(*fld_value1*, *fld_value2*, ...)"

The text representations in the preceding table use the following abbreviations:

fld_value1, *fld_value2*
         are the text representations of the field values.

row_type   is the name of the named row type.

A row type in its text representation is often called a *row-type string*. For example, suppose you have the following unnamed row type defined:

```
ROW(fld1 INTEGER, fld2 CHAR(20))
```

The following row-type string provides the text representation for this unnamed row type:

```
"ROW(7, 'Dexter')"
```

For a detailed description of the text representation of a row type, see the description of the Literal Row segment in the *IBM Informix Guide to SQL: Syntax*.

Row-Type Binary Representation

The database server supports the following kinds of row types.

Row Type	Description

Named row type

> A named row type is identified by its name. With the CREATE ROW TYPE statement, you create a template of a row type. You can then use this template to take the following actions:
>
> - Use type inheritance
> - Define columns that all have the same row type
> - Assign a named row type to a table with the OF TYPE clause of the CREATE TABLE statement

Unnamed row type

> An unnamed row type is identified by its structure. With the ROW keyword, you create a row type. This row type contains fields but has no user-defined name. Therefore, if you want a second column to have the same row type, you must specify all fields.

All row types use the same internal format to store their values. For more information, see the *IBM Informix Guide to SQL: Reference*.

Tip: The internal format of a row type is often referred to as its binary representation.

The DataBlade API supports the SQL row types with the following data type structures:

- A *row descriptor* (MI_ROW_DESC) provides information about the row type.
- A *row structure* (MI_ROW) holds the binary representation of the field values in the row type.

Important: The fields of a row type are comparable to the columns in the row of a table. This similarity means that you use the same DataBlade API data type structures to access row types that you do to access columns in a row.

Using a Row Descriptor

A *row descriptor*, **MI_ROW_DESC**, is a DataBlade API structure that describes the type of data in each field of a row type. The following table summarizes the memory operations for a row descriptor.

Memory Duration	Memory Operation	Function Name
Current memory duration	Constructor	**mi_row_desc_create( )**
	Destructor	**mi_row_desc_free( )**

Tip: A row descriptor can describe a row type or a row in a table. Therefore, you use the same DataBlade API functions to handle memory operations for a row descriptor when it describes a row type or a table row.

─────────────────── **Server Only** ───────────────────

In a C UDR, the row structure and row descriptor are part of the same data type structure. The row structure is just a data buffer in the row descriptor that holds

the column values of a row. A one-to-one correspondence exists between the row descriptor (which **mi_row_desc_create( )** allocates) and its row structure (which **mi_row_create( )** allocates). Therefore:

- When the **mi_row_desc_create( )** function creates a new row descriptor, it assigns a NULL-valued pointer to the data buffer.
- The **mi_row_desc_free( )** function frees both the row descriptor *and* its associated row structure.

───────────────── **End of Server Only** ─────────────────

───────────────── **Client Only** ─────────────────

In a client LIBMI application, a row structure and a row descriptor are separate data type structures. A one-to-many correspondence can exist between a row descriptor and its associated row structures. When you call **mi_row_desc_free( )**, you free only the specified row descriptor.

───────────────── **End of Client Only** ─────────────────

Table 5-3 lists the DataBlade API accessor functions that obtain information about fields of a row type (or columns of a row) from the row descriptor.

Table 5-3. Field and Column Information in the Row Descriptor

Column Information	DataBlade API Accessor Functions
The *number* of columns and/or fields in the row descriptor	**mi_column_count( )**
The *name* of the column or field, given its position in the row	**mi_column_name( )**
The *column identifier*, which is the position of the column or field within the row, given its name	**mi_column_id( )**
The *precision* (total number of digits) of a column or field data type	**mi_column_precision( )**
The *scale* of a column or field data type	**mi_column_scale( )**
Whether a column or field in the row descriptor has the *NOT NULL* constraint	**mi_column_nullable( )**
The *type identifier* of the column or field data type	**mi_column_type_id( )**
The *type descriptor* of the column or field data type	**mi_column_typedesc( )**

Important: To DataBlade API modules, the row descriptor (MI_row_DESC) is an opaque C data structure. Do not access its internal fields directly. The internal structure of MI_ROW_DESC may change in future releases. Therefore, to create portable code, always use the accessor functions for this structure to obtain column information.

The row descriptor stores column information in several parallel arrays.

Column Array	Contents
Column-type ID array	Each element is a pointer to a type identifier (MI_TYPEID) that indicates the data type of the column.
Column-type-descriptor array	Each element is a pointer to a type descriptor (MI_TYPE_DESC) that describes the data type of the column.

Column Array	Contents
Column-scale array	Each element is the scale of the column data type.
Column-precision array	Each element is the precision of the column data type.
Column-nullable array	Each element has either of the following values: • MI_TRUE: The column *can* contain SQL NULL values. • MI_FALSE: The column *cannot* contain SQL NULL values.

All of the column arrays in the row descriptor have zero-based indexes. Within the row descriptor, each column has a *column identifier*, which is a zero-based position of the column (or field) in the column arrays. When you need information about a column (or field), specify its column identifier to one of the row-descriptor accessor functions in Table 5-3 on page 5-30.

Tip: The system catalog tables refer to the unique number that identifies a column definition as its "column identifier." However, the DataBlade API refers to this number as a "column number" and the position of a column within the row structure as a "column identifier." These two terms do not refer to the same value.

Figure 5-9 shows how the information at index position 1 of these arrays holds the column information for the second column in a row descriptor.
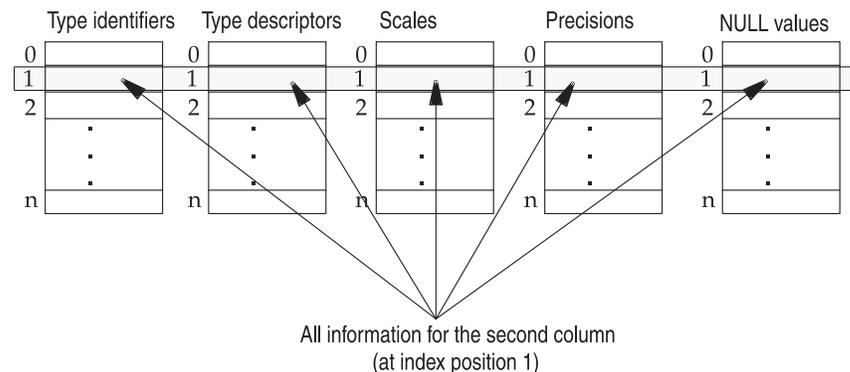
Figure 5-9. Column Arrays in the Row Descriptor

To access information for the *n*th column, provide an index value of *n*-1 to the appropriate accessor function in Table 5-3 on page 5-30. The following calls to the **mi_column_type_id( )** and **mi_column_nullable( )** functions obtain from a row descriptor that **row_desc** identifies the type identifier (**col_type**) and whether the column is nullable (**col_nullable**) for the *second* column:

```
MI_ROW_DESC *row_desc;
MI_TYPEID *col_type;
mi_integer col_nullable;
...
col_type = mi_column_type_id(row_desc, 1);
col_nullable = mi_column_nullable(row_desc, 1);
```

To obtain the number of columns in the row descriptor (which is also the number of elements in the column arrays), use the **mi_column_count( )** function.

Using a Row Structure

The DataBlade API always holds fields of a row type in a *row structure* (**MI_ROW** structure). Each row structure stores the data from a single row-type column in a table. The following table summarizes the memory operations for a row structure.

Memory Duration	Memory Operation	Function Name
Current memory duration	Constructor	**mi_row_create( )**, **mi_streamread_row( )**
	Destructor	**mi_row_free( )**

Tip: A row structure can hold values for the fields of a row type or the columns of a row in a table. Use the same DataBlade API functions to handle memory operations for a row structure when it holds values for a row type or a table row.

─────────────────────── **Server Only** ───────────────────────

In a C UDR, the row structure and row descriptor are part of the same data type structure. The **mi_row_create( )** function just adds a data buffer, which holds the column values of a row, to the row descriptor. A one-to-one correspondence exists between the row descriptor (which **mi_row_desc_create( )** allocates) and its row structure (which **mi_row_create( )** allocates).

If you call **mi_row_create( )** twice with the same row descriptor, the second call overwrites the row values of the first call.

The **mi_row_free( )** function frees the memory associated with the data buffer and assigns a NULL-valued pointer to this buffer in the row descriptor.

─────────────────────── **End of Server Only** ───────────────────────

─────────────────────── **Client Only** ───────────────────────

In a client LIBMI application, a row structure and a row descriptor are separate data type structures. A one-to-many correspondence exists between a row descriptor and its associated row structures. When you call **mi_row_create( )** a second time with the same row descriptor, you obtain a second row structure. The **mi_row_free( )** function frees a row structure.

─────────────────────── **End of Client Only** ───────────────────────

The following DataBlade API functions obtain field values from an existing row structure.

DataBlade API Function	Description
mi_value( ), **mi_value_by_name( )**	Returns a row structure as a column value when the function returns an MI_ROW_VALUE value status
	The row structure holds the fields of the row type.

Tip: A row structure can hold the fields of a row type or the columns of a database row. You use the same DataBlade API functions to handle memory operations for a row structure when it holds row-type fields as when it

describes columns of a row. For more information on how to obtain column values from a row, see "Obtaining Column Values" on page 8-42.

Creating a Row Type

To create a row type, you create a row structure (MI_ROW) that holds the row type. The **mi_row_create( )** function is the constructor function for the row structure (MI_ROW). To create a row type with **mi_row_create( )**, you must provide the following information to the function:

- A row descriptor that describes the fields of the row type (or columns of a row)
- The values of the row-type fields (or row columns)

Creating the Row Descriptor

You create a new row descriptor for a row type with the **mi_row_desc_create( )** function. The **mi_row_desc_create( )** is the constructor function for a row descriptor. You provide this function with the type identifier of the row type for which you want the row descriptor. If you do not know the type identifier for your row type, use the **mi_type_typename( )** function or **mi_typestring_to_id( )** to create a type identifier based on the type name. The type name for a row type is its text representation. For more information, see "Row-Type Text Representation" on page 5-28.

Assigning the Field Values

To provide values for the columns (or fields) of a row structure, you pass information for the columns in several parallel arrays:

- Column-value array
- Column-value null array

These column-value arrays are similar to the column arrays in the row descriptor (see Figure 5-9 on page 5-31). They have an element for each column in the row descriptor. The column-value arrays are different from the column arrays in the row descriptor, in the following ways:

- The column-value arrays describe the actual value for a column.

  Column arrays describe the column data type.

- You must allocate and manage the column-value arrays.

  The DataBlade API does *not* provide accessor functions for these column-value arrays. For each column, your DataBlade API module must declare, allocate, and assign values to these arrays.

All of the column-value arrays have zero-based indexes. Figure 5-10 shows how the information at index position 1 of these arrays holds the column-value information for the second column of a row.
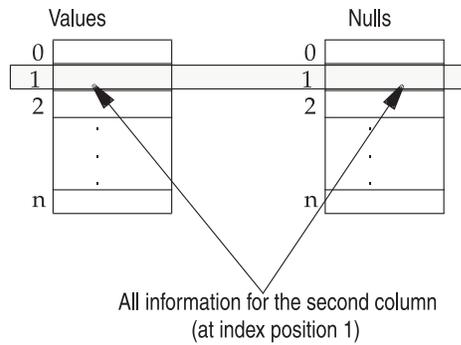
Figure 5-10. Arrays for Initialization of Column

The following sections provide additional information about each of the column-value arrays.

Column-Value Array: The column-value array, *col_values*, is the third argument of the **mi_row_create( )** function. Each element of the column-value array is a pointer to an **MI_DATUM** structure that holds the value for each column. The format of this value depends on whether the **MI_DATUM** value is passed by reference or by value:

Server Only

- For C UDRs, the data type of the value determines the passing mechanism. If the function passes the value by value, the **MI_DATUM** structure contains the value. If the function passes value by reference, the **MI_DATUM** structure contains a pointer to the value.

End of Server Only

Client Only

- For client LIBMI applications, pass *all* values (regardless of data type) by reference. The **MI_DATUM** structure contains a pointer to the value.

End of Client Only

Important: The difference in behavior of **mi_row_create( )** between C UDRs and client LIBMI applications means that row-creation code is not completely portable between these two types of DataBlade API module. When you move your DataBlade API code from one of these uses to another, you must change the row-creation code to use the appropriate passing mechanism for column values that **mi_row_create( )** accepts.

For more information on the passing mechanism for an **MI_DATUM** value, see "Contents of an MI_DATUM Structure" on page 2-33.

Column-Value Null Array: The column-value null array, *col_nulls*, is the fourth argument of the **mi_row_create( )** function. Each element of the column-value null array is either:

- MI_FALSE

  The column value is *not* an SQL NULL value.

- MI_TRUE

The column value *is* an SQL NULL value.

Example: Creating a Row Type

Suppose you have the row type that the following SQL statement creates:

```
CREATE ROW TYPE rowtype_t
(
 id INTEGER,
 name CHAR(20)
);
```

Server Only

The following code shows how to use the **mi_row_create( )** function to create a
new row type of type **rowtype_t**:

```
/*
 * Create a row structure for the 'rowtype_t' row type
 */

   MI_CONNECTION *conn;
   MI_ROW_DESC *rowdesc;
   MI_ROW *row;
   MI_DATUM *values;
   mi_boolean *nulls;
   mi_integer num_cols;

/* Allocate a row descriptor for the 'rowtype_t' row type */
rowdesc = mi_row_desc_create(
        mi_typestring_to_id(conn, "rowtype_t"));

/* Assume number of columns is known */
num_cols = 2;

/* Allocate the 'col_values' and 'col_nulls' arrays */
values = mi_alloc(num_cols *sizeof(MI_DATUM));
nulls = mi_alloc(num_cols *sizeof(mi_boolean));

/* Populate the 'col_values' and 'col_nulls' arrays */

   /* Initialize value for field 1: 'id' */
   values[0] = 1;
   nulls[0] = MI_FALSE;

   /* Initialize value for field 2: 'name' */
   values[1] = mi_string_to_lvarchar("Dexter");
   nulls[1] = MI_FALSE;

/* Create row structure for 'name_t' */
row = mi_row_create(conn, rowdesc, values, nulls);
```

When this code completes, the **row** variable points to a row structure that contains
the following field values.

Field Name	Field Value
fname	″Dexter″
middle	″M″
lname	″Haven″

If the preceding code fragment were part of a client LIBMI application, it would require changes to the way the values are addressed in the **values** array. For example, the INTEGER value would require the following cast to create a copy of the column value:

```
mi_integer col_val;
...
/* Initialize value for field 1: 'id' */
col_val = 1;
values[0] = &col_val;
nulls[0] = MI_FALSE;
```

This different kind of addressing is required because in client LIBMI applications, **mi_row_create( )** passes values for *all* data types by reference. Therefore, the contents of the **MI_DATUM** structure is *always* a pointer to the actual value, *never* the value itself.

Accessing a Row Type

When a row type (named or unnamed) is used as a column of a table, its fields can be accessed in exactly the same ways that the columns of a row are accessed. That is, you create a series of nested loops that use the following functions:

- The **mi_next_row( )** function controls a loop that iterates through each retrieved row type.
- The **mi_value( )** or **mi_value_by_name( )** function controls a loop that iterates through each field value.

For more information on how to use these functions, see "Obtaining Row Values" on page 8-50.

Copying a Row Structure

To create a copy of a row structure, you must:

- Create a new row descriptor that describes the row type.

  For more information, see "Creating the Row Descriptor" on page 5-33.
- Copy the row values from the old row structure into the *col_values* and *col_nulls* arrays to be used for the new row structure.
- Create the new row structure with the values in the *col_values* and *col_nulls* arrays.

The following code fragment copies a row structure:

```
MI_CONNECTION *conn;
MI_ROW_DESC *rowdesc, *new_rowdesc;
mi_integer num_cols, i, len;
MI_DATUM *values;
mi_boolean *nulls;
MI_ROW *new_row;
...

/* Allocate a new row descriptor for the 'name_t' row type */
new_rowdesc = mi_row_desc_create(
        mi_typestring_to_id(conn, "name_t"));
```

```
/* Determine number of columns needed */
num_cols = mi_column_count(new_rowdesc);

/* Allocate the 'col_values' and 'col_nulls' arrays */
values = mi_alloc(num_cols * sizeof(MI_DATUM));
nulls = mi_alloc(num_cols * sizeof(mi_boolean));

/* Populate the 'col_values' and 'col_nulls' arrays */
for ( i=0; i < num_cols; i++ )
   {
   nulls[i] = MI_FALSE; /* assume non-NULL value */

   /* Put field value from original row type ('rowdesc')
    * into 'values' array for new row type ('new_rowdesc'
    */
   switch ( mi_value(rowdesc, i, &values[i], &len) )
      {
      case MI_ERROR:
         /* Unable to get field value. Raise an error */
         break;

      case MI_NULL_VALUE:
         /* Field value is an SQL NULL value. Set 'nulls'
          * array for new row type ('new_rowdesc')
          */
         nulls[i] = MI_TRUE;
         break;

      case MI_NORMAL_VALUE:
         /* No action needed: mi_value( ) call has already
          * copied field value into 'values' array
          */
         break;

      case MI_COLLECTION_VALUE:
         /* Need to add code to handle collection */
         break;

      case MI_ROW_VALUE:
         /* Need to add code to handle nested rows */
         break;

      default:
         /* Handle error */
         break;
      } /* end switch */
   } /* end for */

/* Create new row type with values copied from old row type */
new_row = mi_row_create(conn, new_rowdesc, values, nulls);

/* Deallocate memory for 'values' and 'nulls' arrays */
mi_free(values);
mi_free(nulls);
```

After this code fragment executes, the **new_row** row structure contains a copy of the values in the **row** row structure.

Releasing Row Resources

After your DataBlade API module no longer needs the row type (or row) that you allocated, you need to assess whether you can release resources that the row is using, specifically the row descriptor and the row structure.

Freeing a Row Structure

A row structure has the current memory duration. A row remains valid until one of the following events occurs:

- The **mi_row_free( )** function frees the row.

- The current memory duration expires.

- The **mi_close( )** function closes the current connection.

To conserve resources, use the **mi_row_free( )** function to explicitly deallocate the row once your DataBlade API module no longer needs it. The **mi_row_free( )** function is the destructor function for a row structure. It frees the row and any resources that are associated with it.

In a C UDR, the row structure and row descriptor are part of the same data type structure. The **mi_row_create( )** function just adds a data buffer, which holds the column values of a row, to the row descriptor. The **mi_row_free( )** function drops the row structure from the row descriptor. It is useful for big rows where the data you want has already been examined.

However, the **mi_row_desc_free( )** function frees a row descriptor *and* the associated row structure. Once **mi_row_desc_free( )** frees the row descriptor, you no longer have access to the row structure. Examine the contents of a row structure *before* you deallocate the row descriptor with **mi_row_desc_free( )**.

In a client LIBMI application, a row structure and a row descriptor are separate data type structures. When you free a row descriptor with **mi_row_desc_free( )**, the associated row structure is *not* freed. You must explicitly free the row structure with **mi_row_free( )**.

Important: Use **mi_row_free( )** only for row structures that you have explicitly allocated with **mi_row_create( )**. Do not use this function to free row structures that other DataBlade API functions (such as **mi_next_row( )**) allocate.

Freeing a Row Descriptor

A row descriptor has the current memory duration. A row descriptor remains valid until one of the following events occurs:

- The **mi_row_desc_free( )** function frees the row.

• The current memory duration expires.

• The **mi_close( )** function closes the current connection.

To conserve resources, use the **mi_row_desc_free( )** function to explicitly deallocate the row descriptor once your DataBlade API module no longer needs it. The **mi_row_desc_free( )** function is the destructor function for a row descriptor. It frees the row descriptor and any resources that are associated with it.

In a C UDR, the row structure and row descriptor are part of the same data type structure. The **mi_row_create( )** function just adds a data buffer, which holds the column values of a row, to the row descriptor. The **mi_row_desc_free( )** function frees a row descriptor *and* the associated row structure. Once **mi_row_desc_free( )** frees the row descriptor, you no longer have access to the row structure.

In a client LIBMI application, a row structure and a row descriptor are separate data type structures. When you free a row descriptor with **mi_row_desc_free( )**, the associated row structure is *not* freed. You must explicitly free the row structure with **mi_row_free( )**.

Important: Use **mi_row_desc_free( )** only for row descriptors that you have explicitly allocated with **mi_row_desc_create( )**. Do not use this function to free row structures that other DataBlade API functions (such as **mi_get_row_desc_without_row( )**) allocate.

Chapter 6. Using Smart Large Objects

In This Chapter

This chapter describes smart large objects and provides information about performing the following tasks:

- Storing a smart large object in a database
- Using the smart-large-object interface
- Creating a smart large object
- Accessing a smart large object
- Modifying a smart large object
- Obtaining status information for a smart large object
- Deleting a smart large object
- Converting a smart large object to a file or buffer
- Converting an LO handle to text or binary representation
- Using byte-range locking
- Passing a NULL connection

Tip: For information on the DataBlade API support for simple large objects, see "Simple Large Objects" on page 2-32.

Understanding Smart Large Objects

A *smart large object* is a large object with the following features:

- A smart large object can hold a very large amount of data.

  Currently, a single smart large object can hold up to four terabytes of data. This data is stored in a separate disk space called an *sbspace*.
- A smart large object is recoverable.

  The database server can log changes to smart large objects and therefore can recover smart-large-object data in the event of a system or hardware failure. Logging of smart large objects is not the default behavior.
- A smart large object supports random access to its data.

Access to a simple large object (BYTE or TEXT) is on an "all or nothing" basis; that is, the database server returns all of the simple-large-object data that you request at one time. With smart large objects, you can seek to a desired location and read or write the desired number of bytes.

- You can customize storage characteristics of a smart large object.

  When you create a smart large object, you can specify storage characteristics for the smart large object, such as the following characteristics:

  – Whether the database server logs the smart large object in accordance with the current database logging mode

  – Whether the database server keeps track of the last time the smart large object was accessed

  – Whether the database server uses page headers to detect data corruption

The rest of this section describes the parts of a smart large object and the information that the database server keeps about a smart large object.

Parts of a Smart Large Object

Each smart large object has two parts:

- The *sbspace*, which stores the data of the smart large object
- An *LO handle*, which identifies the location of the smart-large-object data in its sbspace

Suppose you store the picture of an employee as a smart large object. Figure 6-1 shows how the LO handle contains information about the location of the actual employee picture in the **sbspace1_100** sbspace.
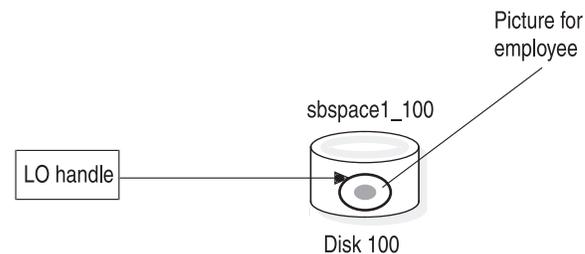
Figure 6-1. Parts of a Smart Large Object

The Sbspace

An *sbspace* is a logical storage area that contains one or more chunks and stores only smart large objects. The sbspace can contain the following parts:

- A *metadata area*

  The database server writes the following information to the metadata area of an sbspace:

  – Internal information that helps the smart-large-object optimizer manage the data efficiently

  – Storage characteristics for the smart large object

  – Status information for the smart large object

- A *user-data area*

  User applications write smart-large-object data to the user-data area of an sbspace.

In Figure 6-1 on page 6-3, the **sbspace1_100** sbspace holds the actual employee image that the LO handle identifies. For more information about the structure of an sbspace, see the chapter on disk structures and storage in your *IBM Informix Administrator's Guide*.

The **onspaces** database utility creates and drops sbspaces for the database server. For more information about the **onspaces** utility, see the chapter on utilities in your *IBM Informix Administrator's Guide*.

Important: Smart large objects can only be stored in sbspaces. They cannot be stored in dbspaces. You must create an sbspace before you attempt to insert smart large objects into the database.

The LO Handle

An *LO handle* is an opaque C data structure that identifies the location of the smart-large-object data in its sbspace. Because a smart large object is potentially very large, the database server stores only its LO handle in a database table; it can then use this LO handle to locate the actual data of the smart large object in the sbspace. This arrangement minimizes the table size.

Applications obtain the LO handle from the database and use it to locate the smart-large-object data and to open the smart large object for read and write operations. In Figure 6-1 on page 6-3, the LO handle identifies the location of the actual employee image in the **sbspace1_100** sbspace. You can store this LO handle in a database column to save this reference for future use. For more information, see "Access to a Smart Large Object" on page 6-14.

Information About a Smart Large Object

The database server keeps the following information about a smart large object:

- Storage characteristics
- Status information

The database server stores this information in a metadata area of the sbspace for the smart large object.

Storage Characteristics

The *storage characteristics* tell the database server how to manage a smart large object in an sbspace. Three groups of information make up the storage characteristics for a smart large object:

- Disk-storage information
- Attribute information
- Open-mode information

You can specify storage characteristics at three points.

When Specified	Method of Specification
When an sbspace is created	Options of **onspaces** utility
When a database table is created	Keywords in PUT clause of CREATE TABLE statement
When a smart large object is created	DataBlade API functions

The following sections describe the three groups of storage characteristics. For additional information, see "Choosing Storage Characteristics" on page 6-28.

Disk-Storage Information: Disk-storage information helps the smart-large-object optimizer of the database server determine how to manage the smart large object most efficiently on disk. The smart-large-object optimizer manages the allocation of and access to smart large objects in an sbspace.

Each smart-large-object has the following disk-storage information:
- Allocation-extent information

  An *allocation extent* is a collection of contiguous bytes within an sbspace that the smart-large-object optimizer allocates to the smart large object at one time. Information about allocation extents is as follows:
  - Extent size

    The smart-large-object optimizer allocates storage for the smart large object in the amount of the extent size.
  - Next-extent size

    The smart-large-object optimizer attempts to allocate an extent as a single, contiguous region in a chunk. If no single extent is large enough for the smart large object, the optimizer uses multiple extents as necessary to satisfy the current write request. After the initial extent fills, the smart-large-object optimizer attempts to allocate another extent of contiguous disk space. This process is called next-extent allocation.

  For more information on extents, see the chapter on disk structure and storage in the *IBM Informix Administrator's Reference*.
- Sizing information
  - Average size of smart large objects in the sbspace
  - Estimated number of bytes in the new smart large object
  - Maximum number of bytes to which the smart large object can grow
- Location

  The name of the sbspace identifies the location at which to store the smart large object.

The smart-large-object optimizer uses the disk-storage information to determine how best to size, allocate, and manage the extents of the sbspace. It can calculate all disk-storage information for a smart large object *except* the sbspace name.

Important: For most applications, use the values that the smart-large-object optimizer calculates for the disk-storage information.

For special situations, you can set disk-storage information for a smart large object as part of its storage characteristics. For more information, see "Choosing Storage Characteristics" on page 6-28.

Attribute Information: Attribute information tells the database server what options, or attributes, to assign to the smart large object:
- *Logging indicators*, which tell the database server whether to log changes to the smart large object in the system log file
- *Last-access-time indicators*, which tell the database server whether to save the last-access time for a smart large object
- *Data-integrity indicators*, which tell the database server how to format the pages in the sbspace of the smart large object

Logging: When a database performs logging, smart large objects might result in long transactions for the following reasons:

- Smart large objects can be very large, even several gigabytes in size.

  The amount of log storage needed to log user data can easily overflow the log.
- Smart large objects might be used in situations where the data collection process can be quite long.

  For example, if a smart large object holds low-quality audio recording, the amount of data collection might be modest but the recording session might be quite long.

A simple workaround is to divide a long transaction into multiple smaller transactions. If this solution is not acceptable, you can control when the database server performs logging of smart large objects. Table 6-10 on page 6-31 shows how you can control the logging behavior for a smart large object.

When logging is enabled, the database server logs changes to the user data of a smart large object. It performs this logging in accordance with the current database log mode. For a database that is not ANSI compliant, the database server does not guarantee that log records that pertain to smart large objects are flushed at transaction commit. However, the metadata is always restorable to an action-consistent state; that is, to a state that ensures no structural inconsistencies exist in the metadata (control information of the smart large object, such as reference counts).

American National Standards Institute

An ANSI-compliant database uses unbuffered logging. When smart-large-object logging is enabled, all log records (metadata and user-data) that pertain to smart large objects are flushed to the log at transaction commit. However, user data is not guaranteed to be flushed to its stable storage location at commit time.

End of American National Standards Institute

When logging is disabled, the database server does *not* log changes to user data even if the database server logs other database changes. However, the database server always logs changes to the metadata. Therefore, the database server can still restore the metadata to an action-consistent state.

Important: Consider carefully whether to enable logging for a smart large object. The database server incurs considerable overhead to log smart large objects. You must also ensure that the system log file is large enough to hold the value of the smart large object. The logical-log size must exceed the total amount of data that the database server logs while the update transaction is active.

Write your DataBlade API modules so that any transactions with smart large objects that have potentially long updates do not cause other transactions to wait. Multiple transactions can access the same smart-large-object instance if the following conditions are satisfied:

- The transaction can access the database row that contains an LO handle for the smart large object.

  Multiple references can exist on the same smart large object if more than one column holds an LO handle for the same smart large object.
- Another transaction does not hold a conflicting lock on the smart large object.

  For more information on smart-large-object locks, see "Locking Modes" on page 6-11.

The best update performance and fewest logical-log problems result when you disable the logging feature when you load a smart large object and re-enable it after the load operation completes. If logging is turned on, you might want to turn logging off before a bulk load and then perform a level-0 backup.

By default, the database server does *not* log the user data of a smart large object. You can control the logging behavior for a smart large object as part of its storage characteristics. For more information, see "Choosing Storage Characteristics" on page 6-28.

Last-Access Time: The last-access time of a smart large object is the system time at which the database server last read or wrote to the smart large object. The last-access time records access to the user data *and* metadata of a smart large object. The database server stores this system time as number of seconds since January 1, 1970, in the metadata area of the sbspace.

Tip: The database server automatically tracks the last-change and last-modification time for a smart large object in the status information. For more information, see "Status Information" on page 6-12.

By default, the database server does *not* save the last access time. You can choose to track the last-access time for a smart large object as part of its storage characteristics. For more information, see "Choosing Storage Characteristics" on page 6-28.

Important: Consider carefully whether to track last-access time for a smart large object. To maintain last-access times for smart large objects, the database server incurs considerable overhead in logging and concurrency.

Data Integrity: The structure of an sbpage in the sbspace determines how much data integrity the database server can provide. An sbpage is the unit of allocation for smart-large-object data, which is stored in the user-data area of an sbspace. The database server supports the following levels of data integrity:

- High integrity tells the database server to use both a page header and a page trailer in each sbpage.
- Moderate integrity tells the database server to use only a page header in each sbpage.

The database server uses the page header and trailer to detect incomplete writes and data corruption. Moderate integrity provides the following benefits:

- It eliminates an additional data copy operation that is necessary when an sbpage has page headers and page trailers.
- It preserves the user data alignments on pages because no page header and page trailer are present.

Moderate integrity might be useful for smart large objects that contain large amounts of audio or video data that is moved through the database server and that does not require a high data integrity. By default, the database server uses high integrity (page headers and page trailers) for sbspace pages. You can control the data integrity for a smart large object as part of its storage characteristics. For more information, see "Choosing Storage Characteristics" on page 6-28.

Important: Consider carefully whether to use moderate integrity for sbpages of a smart large object. Although moderate integrity takes less disk space per page, it also reduces the ability of the database server to recover information if disk errors occur.

For information on the structure of sbspace pages, see your *IBM Informix Administrator's Guide*.

Open-Mode Information: When you open a smart large object, you can specify the *open mode* for the data. The open mode describes the context in which the I/O operations on the smart large object are performed. It includes the following information:

- The *access mode* for the smart large object: read-only, dirty-read, read/write, write-only, or write-append
- The *access method* for the smart large object: random or sequential
- The *buffering mode* for the data to and from the smart large object: buffered or unbuffered
- The *locking mode* for the smart large object: lock-all or byte-range mode

The database server uses the following system default open mode when it opens a smart large object.

Open-Mode Information	Default Open Mode
Access mode	Read-only
Access method	Random
Buffering	Buffered access
Locking	Whole-object locks

If your smart large object usually requires certain access capabilities when it is opened, you can associate a *default open mode* with the smart large object. The database server stores this default open mode with other storage characteristics of the smart large object. For more information, see "Choosing Storage Characteristics" on page 6-28. To override the default open mode, you can specify an open mode for a particular smart large object when you open it. For more information, see "Opening a Smart Large Object" on page 6-48.

Access Modes: The smart-large-object open mode includes an *access mode*, which determines which read and write operations are valid on the open smart large object. Table 6-1 shows the access modes for a smart large object.

Table 6-1. Access Modes for Smart Large Objects

Access Mode	Purpose
Read-only mode	Only read operations are valid on the data.
Dirty-read mode	You can read uncommitted data pages for the smart large object. No locks are requested on the data. You cannot write to a smart large object after you set the mode to MI_LO_DIRTY_READ. When you set this flag, you reset the current transaction isolation mode to dirty read for this smart large object.
Write-only mode	Only write operations are valid on the data.

Table 6-1. Access Modes for Smart Large Objects (continued)

Access Mode	Purpose
Write/append mode	Any data you write is appended to the end of the smart large object. By itself, it is equivalent to write-only mode followed by a seek to the end of the smart large object. Read operations fail. When you open a smart large object in write/append mode only, the smart large object is opened in write-only mode. Seek operations move the seek position, but read operations to the smart large object fail, and the LO seek position remains unchanged from its position just before the write. Write operations occur at the LO seek position, and then the seek position is moved.
Read/write mode	Both read and write operations are valid on the data.
Truncate	Delete any existing data in the smart large object and move the LO seek position to the start of the smart large object (byte 0). If the smart large object does not contain data, this access mode has no effect.

Access Methods: The smart-large-object open mode includes the *access method*, which determines whether to access the smart-large-object data sequentially or with random access. Table 6-2 shows the access methods for a smart large object.

Table 6-2. Access Methods for a Smart Large Object

Method of Access	Purpose		
Random access	Indicates that I/O is random When you plan to read in nonsequential locations in the smart large object, the smart-large-object optimizer should not read ahead a few pages.		
Sequential access	Indicates that reads are sequential in either forward or reverse direction When you read a smart large object sequentially, the smart-large-object optimizer can read ahead a few pages.		
		Forward	Indicates that the direction of sequential access is forward If you do not specify a direction, the default is forward.
		Reverse	Indicates that the direction of sequential access is reverse

The default access method is random, although the smart-large-object optimizer might change this default based on a particular read pattern.

Buffering Modes: The smart-large-object open mode includes a *buffering mode*, which determines how read and write operations on the open smart large object are buffered. Table 6-3 shows the buffering modes for a smart large object.

Table 6-3. Buffering Modes for a Smart Large Object

Buffering Mode	Purpose
Buffered access	Indicates that I/O of the smart-large-object data goes through the buffer pool of the database server
	This method of access is called *buffered I/O*. Buffered I/O tells the optimizer that someone might be planning to reread the same LO page.
Unbuffered access	Indicates that I/O of the smart-large-object data does *not* use the buffer pool
	This method of access is called *lightweight I/O*. lightweight I/O tells the smart-large-object optimizer to use private buffers instead of the buffer pool for these I/O operations. These private buffers are allocated out of the session pool of the database server. With lightweight I/O, you bypass the overhead of the buffer pool management when the database server performs a sequential scan.

Keep the following issues in mind when you use lightweight I/O:

- Be sure that you close smart large objects that use lightweight I/O.

    Otherwise, the memory that has been allocated to the private buffers remains allocated. This private-buffer memory is only deallocated when you close the smart large object.

- Be careful about using lightweight I/O when you open the same smart large object many times and concurrently access this object in the same transaction.

    All opens of the same smart large object share the same lightweight I/O buffers. Potentially, an operation can cause the pages in the buffer to be flushed while other operations might still expect these pages to exist.

Important: In general, if read and write operations to the smart large objects are less than 8080 bytes, do not use lightweight I/O. In other words, if you are reading or writing short blocks of data, such as two kilobytes or four kilobytes, the default buffered I/O operations provide better performance.

The smart-large-object optimizer imposes the following restrictions when you switch from lightweight I/O to buffered I/O for a given smart large object:

- You can alter the buffering mode of a smart large object that was created with lightweight I/O to buffered I/O as long as no open instances exist for that smart large object.

    However, you cannot alter the buffering mode from buffered I/O to one with lightweight I/O.

- You must specify lightweight I/O when you open a smart large object that was created with lightweight I/O.

    If an open smart large object specifies buffered I/O, the smart-large-object optimizer ignores any attempt to open it with lightweight I/O. However, if you first change the buffering mode from lightweight I/O to buffered I/O, you can then specify buffered I/O when you open the smart large object.

- You can specify lightweight I/O when you open a smart large object that was created with buffered I/O *only* if you open the smart large object in read-only mode.

In this case, the smart-large-object optimizer does not allow write operations on the smart large object. Attempts to do so generate an error. To write to the smart large object, you must close it then reopen it with buffered I/O and an access mode that enables write operations.

These limitations ensure consistency of the smart-large-object buffers without imposing processing overhead for I/O operations.

If you do not specify a buffering mode, the default is buffered I/O. The smart-large-object optimizer determines the default buffering mode for a smart large object.

Locking Modes: To prevent simultaneous access to smart-large-object data, the smart-large-object optimizer obtains a lock on this data when you open the smart large object. This smart-large-object lock is distinct from the following kinds of locks:

- Row locks

  A lock on a smart large object does *not* lock the row in which the smart large object resides. However, if you retrieve a smart large object from a row and the row is still current, the database server might hold a row lock as well as a smart-large-object lock. Locks are held on the smart large object instead of on the row because many columns could be accessing the same smart-large-object data.

- Locks of different smart large objects in the same row of a table

  A lock on one smart large object does not affect other smart large objects in the row.

The smart-large-object open mode includes a *lock mode*, which determines the kind of the lock requests made on a smart large object. Table 6-4 shows the lock modes that a smart large object can support.

Table 6-4. Lock Modes for a Smart Large Object

Lock Mode	Purpose	Description
Lock-all	Lock the entire smart large object	Indicates that lock requests apply to *all* data for the smart large object
Byte-range	Lock only specified portions of the smart large object	Indicates that lock requests apply *only* to the specified number of bytes of smart-large-object data

When the smart-large-object optimizer opens a smart large object, it uses the following information to determine the lock mode of the smart large object:

- The access mode of the smart large object

  The database server obtains a lock as follows:

  – In *share mode,* when you open a smart large object for reading (read-only or dirty read)

  – In *update mode*, when you open a smart large object for writing (write-only, read-write, write/append, truncate)

    When a write operation (or some other update) is actually performed on the smart large object, the database server upgrades this lock to an *exclusive lock*.

- The isolation level of the current transaction

If you have selected an isolation mode of repeatable read, the smart-large-object optimizer does not release any locks that it obtains on a smart large object until the end of the transaction.

By default, the smart-large-object optimizer chooses the lock-all lock mode. You can request locks on the data of a smart large object at the byte level with a *byte-range lock*. For more information, see "Accessing the Default Open Flag" on page 6-38.

The smart-large-object optimizer retains the lock as follows:

- It holds share-mode locks and update locks (which have not yet been upgraded to exclusive locks) until one of the following events occurs:
  - The closing of the smart large object
  - The end of the transaction
  - An explicit request to release the lock (for a byte-range lock *only*)
- It holds exclusive locks until the end of the transaction even if you close the smart large object.

When one of the preceding conditions occurs, the smart-large-object optimizer releases the lock on the smart large object.

Important: You lose the lock at the end of a transaction even if the smart large object remains open. When the smart-large-object optimizer detects that a smart large object has no active lock, it automatically obtains a new lock when the first access occurs to the smart large object. The lock that it obtains is based on the original access mode of the smart large object.

The smart-large-object optimizer releases the lock when the current transaction terminates. However, the optimizer obtains the lock again when the next function that needs a lock executes. If this behavior is undesirable, use BEGIN WORK transaction blocks and place a COMMIT WORK or ROLLBACK WORK statement after the last statement that needs to use the lock.

Status Information

Table 6-5 shows the status information that the database server maintains for a smart large object.

Table 6-5. Status Information for a Smart Large Object

Status Information	Description
Last-access time	The time, in seconds, that the smart large object was last accessed
	This value is available *only* if the last-access time attribute is enabled for the smart large object.
Storage characteristics	The storage characteristics for the smart large object
Last-change time	The time, in seconds, of the last change in status for the smart large object
	A change in status includes changes to metadata and user data (data updates and changes to the number of references). This system time is stored as number of seconds since January 1, 1970.

Table 6-5. Status Information for a Smart Large Object (continued)

Status Information	Description
Last-modification time	The time, in seconds, that the smart large object was last modified
	A modification includes only changes to user data (data updates). This system time is stored as number of seconds since January 1, 1970.
	On some platforms, the last-modification time might also have a microseconds component, which can be obtained separately from the seconds component.
Reference count	The number of references (LO handles) to the smart large object
Size	The size, in bytes, of the smart large object

The database server stores the status information in the metadata area of the sbspace.

Tip: The time values (such as last-access time and last-change time) might differ slightly from the system time. This difference is due to the algorithm that the database server uses to obtain the time from the operating system.

For more information on how to obtain status information in a DataBlade API module, see "Obtaining Status Information for a Smart Large Object" on page 6-52.

Storing a Smart Large Object in a Database

To store a smart large object in a database, you must save its LO handle in a column. This section describes the valid data types to hold an LO handle and how to access a smart large object.

Valid Data Types

In the database, you can use either of the following ways to store a smart large object in a column:

- For direct access to the smart large object, create a column of the CLOB or BLOB data type.
- To hide the smart large object within an atomic data type, create an opaque type that holds a smart large object.

CLOB and BLOB Data Types

You can store a smart large object directly in a column that has one of the following data types:

- The CLOB data type holds text data.
- The BLOB data type can store any kind of binary data in an undifferentiated byte stream.

The CLOB or BLOB column holds an LO handle for the smart large object. Therefore, when you select a CLOB or BLOB column, you do not obtain the actual data of the smart large object, but the LO handle that identifies this data. The BLOB and CLOB data types have identical internal representation. Externally, an LO handle is represented as a flat array of bytes with a length of MI_LO_SIZE.

Suppose an **employee** table has a BLOB column named **emp_picture** to hold the picture of an employee. Figure 6-2 shows that in a row of the **employee** table, the **emp_picture** column contains an LO handle. This LO handle contains information about the location of the actual employee picture in the **sbspace1_100** sbspace.
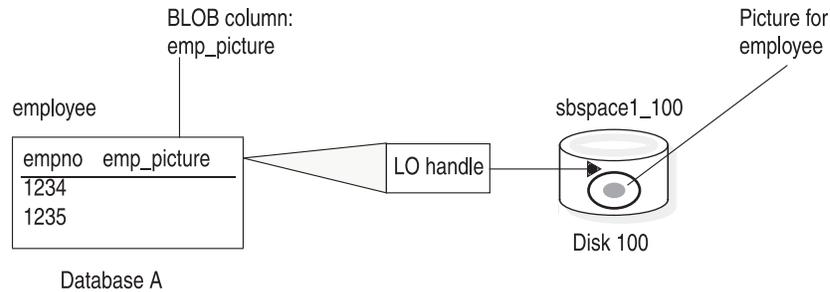
Figure 6-2. A Smart Large Object in a Database Column

The CLOB and BLOB data types are often referred to collectively as *smart-large-object data types*. For more information on these data types, see the *IBM Informix Guide to SQL: Reference*.

Opaque Data Type

An opaque data type is a user-defined atomic data type. You can define a field of an opaque data type to be a smart large object. The support functions of the opaque type must perform the conversion between the LO handle in the opaque type and the smart-large-object data in the sbspace. For more information, see "Managing the Reference Count" on page 6-56.

In Figure 6-2, the **emp_picture** column could be an opaque data type named **picture** instead of a BLOB data type. The **picture** data type could hold the image in a smart large object in one field of its internal structure and other information about the picture in other fields.

For more information on opaque data types, see the *IBM Informix Guide to SQL: Reference* and the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Access to a Smart Large Object

The DataBlade API provides the smart-large-object interface for access to smart large objects. This interface contains a set of functions and data types to provide access to smart large objects. (For more information, see "Using the Smart-Large-Object Interface" on page 6-15.) The smart-large-object interface provides access to the smart large object through its LO handle, as follows:

- Once you *select* a column that contains an LO handle, you can use this handle to access the smart-large-object data in an sbspace.
- To *store* a new smart large object, you create a new LO handle, write the data to the sbspace, and store the LO handle in the column.

Selecting a Smart Large Object

A SELECT statement on a CLOB, BLOB, or opaque-type column retrieves an LO handle for a smart large object. It does *not* retrieve the actual data for the smart large object because this data resides in an sbspace.

To select a smart large object:

1. Use a SELECT statement to retrieve the LO handle from the CLOB, BLOB, or opaque-type column.

   The LO handle identifies the location of the smart large object on disk.

2. Read the smart-large-object data from the sbspace of the smart large object.

   The LO handle identifies the smart large object to open. Once you open the smart large object, you obtain an LO file descriptor, which you can use to read data from the sbspace of the smart large object.

Storing a Smart Large Object

Because a smart large object can be quite large, it is not practical to store it directly in the database table. Instead, the INSERT and UPDATE statements store the LO handle of the smart large object in the CLOB, BLOB, or opaque-type column. The data of the smart large object resides in an sbspace.

To save a smart large object in a CLOB, BLOB, or opaque-type column:

1. For a new smart large object, ensure that the smart large object has an sbspace specified for its data.

   For most smart large objects, the sbspace name is the *only* storage characteristic that you need to specify. The smart-large-object optimizer can calculate values for all other storage characteristics. You can set particular storage characteristics to override these calculated values. However, most applications do not need to set storage characteristics at this level of detail. For more information, see "Obtaining Storage Characteristics" on page 6-28.

2. Create a new LO handle for the smart large object and open the smart large object.

   When you create a smart large object, you obtain an LO handle and an LO file descriptor for the new smart large object.

3. Write the smart-large-object data to the sbspace of the smart large object.

   Use the LO file descriptor to identify the smart large object whose data you want to write to the sbspace.

4. Use the INSERT or UPDATE statement to store the LO handle into the CLOB, BLOB, or opaque-type column.

   The LO handle for the smart large object identifies the location of the smart large object on disk. Once you have written the data to the smart large object, provide its LO handle to the INSERT or UPDATE statement to save it in the database. The smart-large-object data remains in the sbspace.

Important: The sbspace for the smart large object must exist before the INSERT statement executes.

When you store an LO handle in the database, the database server can ensure that the smart large objects are only freed when no more database columns reference them. For more information, see "Deleting a Smart Large Object" on page 6-56. For information on how to insert a smart large object from within a DataBlade API module, see "Creating a Smart Large Object" on page 6-24.

Using the Smart-Large-Object Interface

The smart-large-object interface contains a set of functions and data types to provide access to smart large objects. It enables you to access the data of a smart large object in much the same way as you would access an operating-system file on UNIX, Linux, or Windows. The interface provides the following:

- Smart-large-object functions

- Smart-large-object data type structures

The **milo.h** header file defines the functions and data type structures of the smart-large-object interface. The **mi.h** header file automatically includes the **milo.h** header file. You must include either **mi.h** or **milo.h** in any DataBlade API routine that calls a smart-large-object function or declares one of the smart-large-object data type structures.

Sections of this chapter describe how to use the smart-large-object interface to perform the following operations on a smart large object.

Smart-Large-Object Operation	More Information
Create a new smart large object	page 6-24
Access data in an existing smart large object	page 6-46
Modify an existing smart large object	page 6-50
Obtain status information about an existing smart large object	page 6-52
Delete a smart large object	page 6-56

Smart-Large-Object Data Type Structures

The smart-large-object interface provides data type structures that store information about a smart large object. Table 6-6 summarizes the data type structures of the smart-large-object interface.

Table 6-6. Data Types of the Smart-Large-Object Interface

Smart-Large-Object Data Type Structure	Data Type	Description
The LO-specification structure	**MI_LO_SPEC**	Holds storage characteristics for a smart large object
The LO handle	**MI_LO_HANDLE**	Identifies the location of the smart large object; analogous to the filename of an operating-system file
The LO file descriptor	**MI_LO_FD**	Identifies an open smart large object; analogous to the file descriptor of an operating-system file
The LO-status structure	**MI_LO_STAT**	Holds status information about a smart large object

These structures are all opaque to a DataBlade API module; that is, you do not access their fields directly but instead use accessor functions that the smart-large-object interface provides.

LO-Specification Structure

The *LO-specification structure*, **MI_LO_SPEC**, defines the storage characteristics for an existing or a new smart large object. The storage characteristics provide information about features of the smart large object and how to store it on disk. For a description of the storage characteristics available, see "Storage Characteristics" on page 6-4.

The following table summarizes the memory operations for an LO-specification structure.

Memory Duration	Memory Operation	Function Name
Current memory duration	Constructor	**mi_lo_spec_init( )**
	Destructor	**mi_lo_spec_free( )**

To access an LO-specification structure in a DataBlade API module, declare a pointer to an **MI_LO_SPEC** structure. For example, the following line shows the valid syntax of a variable that accesses an LO-specification structure:

```
MI_LO_SPEC *myspec; /* valid syntax */
```

Declaration of a flat LO-specification structure generates a compile error. The following line shows *invalid syntax* for an LO-specification structure:

```
MI_LO_SPEC myspec; /* INVALID syntax */
```

The **milo.h** header file defines the **MI_LO_SPEC** data type. Therefore, you must include the **milo.h** (or **mi.h**) file in DataBlade API modules that access this structure. For information on how to use an LO-specification structure, see "Obtaining the LO-Specification Structure" on page 6-25.

LO Handle

An *LO handle*, **MI_LO_HANDLE**, serves as a reference to a smart large object. It is analogous to the filename of an operating-system file in that it is a unique identifier of a smart large object. The LO handle contains encoded information about the smart large object, such as its physical disk location and other security-related information. After a smart large object is created, an associated LO handle is a valid reference for the life of that smart large object.

The following table summarizes the memory operations for an LO handle.

Memory Duration	Memory Operation	Function Name
Current memory duration	Constructor	**mi_get_lo_handle( )**, **mi_lo_copy( )**, **mi_lo_create( )**, **mi_lo_expand( )**, **mi_lo_from_buffer( )**, **mi_lo_from_string( )**, **mi_streamread_lo( )**
	Destructor	**mi_lo_delete_immediate( )**, **mi_lo_release( )**

To access an LO handle in a user-defined routine (UDR), declare it in one of the following ways:

- As a pointer to the **MI_LO_HANDLE** data type:

  ```
  MI_LO_HANDLE *my_LOhndl; /* an LO-handle pointer */
  ```

  When you declare an LO handle in this way, you must allocate memory for it before you use it. For more information, see "Obtaining an LO Handle" on page 6-40.

- As a flat **MI_LO_HANDLE** structure:

  ```
  MI_LO_HANDLE my_flat_LOhnld; /* a flat LO handle */
  ```

  When you declare a flat **MI_LO_HANDLE** structure, you do *not* need to allocate memory for it. This flat structure is useful when you need to embed an LO handle within an opaque data type.

The **milo.h** header file defines the **MI_LO_HANDLE** data type. Therefore, you must include the **milo.h** (or **mi.h**) file in DataBlade API modules that access this handle. For information on how to use an LO handle, see "Initializing an LO-Specification Structure" on page 6-27 and "Selecting the LO Handle" on page 6-47.

LO File Descriptor

The *LO file descriptor*, **MI_LO_FD**, is a reference to an open smart large object. An LO file descriptor is similar to a file descriptor for an operating-system file. It is an integer number that serves as a transient descriptor for performing I/O on the data of the smart large object. It provides the following information about an open smart large object:

- The LO *seek position*, the current position at which read and write operations occur.

  When you first open a smart large object, the seek position is at byte zero (0).

- The *open mode* of the smart large object, which determines which operations can be performed on the data and how to buffer the data for I/O operations.

  You specify the open mode when you open a smart large object. For more information, see "Open-Mode Information" on page 6-8.

The following table summarizes the memory operations for an LO file descriptor.

Memory Duration	Memory Operation	Function Name
Not allocated from memory-duration pools	Constructor	**mi_lo_copy( )**, **mi_lo_create( )**, **mi_lo_expand( )**, **mi_lo_from_file( )**, **mi_lo_open( )**
	Destructor	**mi_lo_close( )**

To access an LO file descriptor in a DataBlade API module, declare a variable with the **MI_LO_FD** data type. For example, the following line declares the variable **my_lofd** that is an LO file descriptor:

```
MI_LO_FD my_lofd;
```

The **milo.h** header file defines the **MI_LO_FD** data type. Therefore, you must include the **milo.h** (or **mi.h**) file in DataBlade API modules that access this handle.

Tip: Other smart-large-object data type structures require that you declare a pointer to them because the DataBlade API handles memory allocation for these structures. However, you can declare an LO file descriptor directly.

─────────────── Server Only ───────────────

Because you declare an LO file descriptor directly, its scope is that of the variable you declare to hold it. When you assign an LO file descriptor to a local variable, the LO file descriptor is deallocated when the function that declares it ends. If you want to keep the LO file descriptor longer, you can allocate user memory with the memory duration you want (up to the advanced duration of PER_SESSION) and copy the LO file descriptor into this memory. For example, you could assign the LO file descriptor to PER_COMMAND memory and copy it into the user state of the **MI_FPARAM** structure. For more information, see "Managing the Memory Duration" on page 14-21 and "Saving a User State" on page 9-8.

Important: Although the scope of an LO file descriptor is determined by its declaration, the scope of the open smart large object (which the LO file descriptor identifies) is the entire session. Make sure you explicitly close a smart large object before the scope of its LO file descriptor expires. For more information, see "Freeing a Smart Large Object" on page 6-49.

──────────── End of Server Only ────────────

For information on how to use an LO file descriptor, see "Initializing an LO-Specification Structure" on page 6-27.

LO-Status Structure

The *LO-status structure*, **MI_LO_STAT**, contains the status information for an existing smart large object. The following table summarizes the memory operations for an LO-status structure.

Memory Duration	Memory Operation	Function Name
Current memory duration	Constructor	**mi_lo_stat( )**
	Destructor	**mi_lo_stat_free( )**

To access an LO-status structure in a DataBlade API module, declare a pointer to an **MI_LO_STAT** structure. For example, the following line declares the variable **mystat** that points to an LO-specification structure:

```
MI_LO_STAT *mystat; /* valid syntax */
```

Declaration of a flat LO-status structure generates a compile error. The following line shows *invalid syntax* for an LO-status structure:

```
MI_LO_STAT mystat; /* INVALID syntax */
```

The **milo.h** header file defines the **MI_LO_STAT** data type. Therefore, you must include the **milo.h** (or **mi.h**) file in DataBlade API modules that access this structure. For information on how to allocate and use an LO-status structure, see "Obtaining Status Information" on page 6-54.

Smart-Large-Object Functions

The smart-large-object interface includes functions that provide the following operations on a smart large object:
- Creating a smart large object
- Performing input and output (I/O) on smart-large-object data
- Manipulating LO handles
- Accessing storage characteristics
- Obtaining status information
- Moving smart large objects to and from operating-system files

Most of the smart-large-object function names begin with the string '**mi_lo_**'. The *IBM Informix DataBlade API Function Reference* contains an alphabetical list of all DataBlade API functions, including the smart-large-object functions.

Functions That Create a Smart Large Object

The smart-large-object creation functions create a new smart large object, open it, and return a new LO handle and LO file descriptor for it. Table 6-7 lists the

smart-large-object creation functions.

Table 6-7. Smart-Large-Object Creation Functions

Smart-Large-Object Creation Function	Description
mi_lo_create( )	Creates a new, empty smart large object
mi_lo_copy( )	Creates a new smart large object that is a copy of an existing smart large object
mi_lo_expand( ) *(deprecated)*	Creates a new smart large object from existing multirepresentational data
mi_lo_from_file( )	Creates a new smart large object from data in an operating-system file

For more information on how to use the smart-large-object creation functions, see "Creating a Smart Large Object" on page 6-24.

Functions That Perform Input and Output on a Smart Large Object

The smart-large-object interface for Dynamic Server includes functions that provide basic file operations such as create, open, seek, read, write, alter, and truncate. These routines bypass the query processor, executor, and optimizer, and give the application direct access to a smart large object. These functions use an LO file descriptor to identify the open smart large object.

Table 6-8 shows the basic file-like operations on a smart large object with the smart-large-object function that performs them and the analogous operating-system calls for file operations.

Table 6-8. Main DataBlade API Functions of the Smart-Large-Object Interface

Smart-Large-Object Operation	Smart-Large-Object Function	Operating-System Call
Open the smart large object that the LO handle identifies: the open operation generates an LO file descriptor for the smart large object.	mi_lo_open( )	open( )
Seek to the desired LO seek position to begin a read or write operation.	mi_lo_seek( )	seek( )
Obtain the current LO seek position.	mi_lo_tell( )	tell( )
Lock the specified number of bytes of data.	mi_lo_lock( )	lock( )
Perform the read or write operation for the specified number of bytes.	mi_lo_read( ), mi_lo_readwithseek( ), mi_lo_write( ), mi_lo_writewithseek( )	read( ), write( )
Unlock the specified number of bytes of data.	mi_lo_unlock( )	unlock( )
Obtain status information about a particular smart large object.	mi_lo_stat( )	stat( )
Truncate smart-large-object data at a specified location.	mi_lo_truncate( )	truncate( )
Close the smart large object and free the LO file descriptor.	mi_lo_close( )	close( )

For more information, see "Opening a Smart Large Object" on page 6-48.

Functions That Manipulate an LO Handle

The following table shows the smart-large-object functions that act on an LO handle, not on the smart large object that it identifies.

DataBlade API Function	Purpose
mi_get_lo_handle( )	Obtains an LO handle from a user-defined buffer
mi_lo_alter( )	Alters the storage characteristics of the smart large object that the LO handle identifies
mi_lo_copy( )	Copies the contents of a smart large object (that an LO handle identifies) into a new smart large object and initializes the LO handle of the new smart large object
mi_lo_create( )	Creates a new smart large object and initializes its LO handle
mi_lo_decrefcount( )	Decrements the reference count of the smart large object that the LO handle identifies
mi_lo_expand( ) *(deprecated)*	Copies multirepresentational data into a new smart large object and initializes the LO handle
mi_lo_filename( )	Returns the name of the file where the **mi_lo_to_file( )** function would store the smart large object that the LO handle identifies
mi_lo_from_buffer( )	Copies a specified number of bytes from a user-defined buffer into a smart large object that the LO handle identifies
mi_lo_from_file( )	Copies the contents of an operating-system file to a smart large object that the LO handle identifies
mi_lo_from_string( )	Converts an LO handle from its text representation to its binary representation
mi_lo_increfcount( )	Increments the reference count of the smart large object that the LO handle identifies
mi_lo_invalidate( )	Marks an LO handle as invalid
mi_lo_lolist_create( )	Converts an array of LO handles into an **MI_LO_LIST** structure
mi_lo_open( )	Opens the smart large object that the LO handle identifies
mi_lo_ptr_cmp( )	Compares two LO handles to see if they identify the same smart large object
mi_lo_release( )	Releases resources held by a transient smart large object, including its LO handle
mi_lo_to_buffer( )	Copies a specified number of bytes from a smart large object that the LO handle identifies into a user-defined buffer
mi_lo_to_file( )	Copies the smart large object that the LO handle identifies to an operating-system file

mi_lo_to_string( )	Converts an LO handle from its binary representation to its text representation
mi_lo_validate( )	Checks whether an LO handle is valid
mi_put_lo_handle( )	Puts an LO handle into a user-defined buffer

| | **Important:** The LO handle, **MI_LO_HANDLE**, is an opaque structure to DataBlade API modules. Do not access its internal structure directly. There is no guarantee that the internal structure of **MI_LO_HANDLE** will not change. To create portable code, use the appropriate DataBlade API function to access this structure. |

For more information on how to use these functions, see "Obtaining an LO Handle" on page 6-40.

Functions That Access an LO-Specification Structure

The following table shows the smart-large-object functions that access the LO-specification structure.

DataBlade API Function	Purpose
mi_lo_alter( )	Alters the storage characteristics of an existing smart large object
mi_lo_colinfo_by_ids( )	Updates the LO-specification structure with the column-level storage characteristics for a column identified by a row descriptor
mi_lo_colinfo_by_name( )	Updates the LO-specification structure with the column-level storage characteristics for a column identified by name
mi_lo_copy( )	Copies the contents of the smart large object into a new smart large object, whose storage characteristics the LO-specification structure contains
mi_lo_create( )	Creates a new smart large object that has the storage characteristics in the LO-specification structure
mi_lo_expand( ) *(deprecated)*	Copies multirepresentational data into a new smart large object, whose storage characteristics the LO-specification structure contains
mi_lo_from_file( )	Copies the contents of an operating-system file to a smart large object, whose storage characteristics the LO-specification structure contains
mi_lo_spec_free( )	Frees the resources of the LO-specification structure
mi_lo_spec_init( )	Allocates and initializes an LO-specification structure
mi_lo_specget_def_open_flags( )	Retrieves the default open mode from the LO-specification structure
mi_lo_specget_estbytes( )	Retrieves the estimated number of bytes from the LO-specification structure
mi_lo_specget_extsz( )	Accessor function to get the allocation extent size from the LO-specification structure

DataBlade API Function	Purpose
mi_lo_specget_flags( )	Accessor function to get the attributes flag from the LO-specification structure
mi_lo_specget_maxbytes( )	Accessor function to get the maximum number of bytes from the LO-specification structure
mi_lo_specget_sbspace( )	Accessor function to get the name of the sbspace from the LO-specification structure
mi_lo_specset_def_open_flags( )	Accessor function to set the default open mode in the LO-specification structure
mi_lo_specset_estbytes( )	Accessor function to set the estimated number of bytes in the LO-specification structure
mi_lo_specset_extsz( )	Accessor function to set the allocation extent size in the LO-specification structure
mi_lo_specset_flags( )	Accessor function to set the attribute flags in the LO-specification structure
mi_lo_specset_maxbytes( )	Accessor function to set the maximum number of bytes in the LO-specification structure
mi_lo_specset_sbspace( )	Accessor function to set the name of the sbspace in the LO-specification structure
mi_lo_stat_cspec( )	Returns a pointer to the LO-specification structure that contains the storage characteristics obtained from the LO-status structure of an existing smart large object

Important: The *LO*-specification structure, **MI_LO_SPEC**, is an opaque structure to DataBlade API modules. Do not access its internal structure directly. The internal structure of **MI_LO_SPEC** may change in future releases. Therefore, to create portable code, always use the LO-specification accessor functions to obtain and store values in this structure.

For more information on how to use these functions, see "Obtaining the LO-Specification Structure" on page 6-25 and "Choosing Storage Characteristics" on page 6-28.

Functions That Access an LO-Status Structure

The following table shows the smart-large-object functions that access the LO-status structure.

DataBlade API Function	Purpose
mi_lo_stat( )	Allocates and initializes an LO-status structure with status information of an open smart large object
mi_lo_stat_atime( )	Accessor function to get the last-access time
mi_lo_stat_cspec( )	Accessor function to get the storage characteristics
mi_lo_stat_ctime( )	Accessor function to get the last-change time
mi_lo_stat_free( )	Frees the resources of the LO-status structure
mi_lo_stat_mtime_sec( )	Accessor function to get the seconds component of the last-modification time
mi_lo_stat_mtime_usec( )	Accessor function to get the microseconds component of the last-modification time
mi_lo_stat_refcnt( )	Accessor function to get the reference count

DataBlade API Function	Purpose
mi_lo_stat_size( )	Accessor function to get the size of smart large object

Important: The *LO*-status structure, **MI_LO_STAT**, is an opaque structure to
DataBlade API modules. Do not access its internal structure directly.
The internal structure of **MI_LO_STAT** may change in future releases.
Therefore, to create portable code, always use the LO-status accessor
functions to obtain and store values from this structure.

For more information on how to use these functions, see "Obtaining Status
Information" on page 6-54.

Functions That Move Smart Large Objects to and from Operating-System Files

The following table shows the smart-large-object functions that move smart large
objects to and from operating-system files.

DataBlade API Function	Purpose
mi_file_to_file( )	Copies the contents of one operating-system file to another
mi_lo_from_file( )	Copies the contents of an operating-system file to a new smart large object
mi_lo_from_file_by_lofd( )	Copies the contents of an operating-system file to an existing smart large object
mi_lo_to_file( )	Copies the contents of a smart large object to a new operating-system file

For more information on how to use these functions, see "Using Operating-System
Files" on page 6-59.

Creating a Smart Large Object

To create a smart large object and save its LO handle in the database, you need to
take the following steps. For details on a step, see the page listed under "More
Information."

Step	Task	Smart-Large-Object Function	More Information
1.	Obtain an LO-specification structure to hold the storage characteristics for the new smart large object.	mi_lo_spec_init( ), mi_lo_stat_cspec( )	page 6-25
2.	Ensure that the LO-specification structure contains the desired storage characteristics for the new smart large object.	System-specified storage characteristics: mi_lo_spec_init( ) Column-level storage characteristics: mi_lo_colinfo_by_name( ), mi_lo_colinfo_by_ids( ) User-specified storage characteristics: Table 6-14 on page 6-35, Table 6-15 on page 6-36	page 6-28
3.	Create an LO handle for the new smart large object and open the smart large object.	mi_lo_create( ), mi_lo_expand( ), mi_lo_copy( ), mi_lo_from_file( )	page 6-40

Step	Task	Smart-Large-Object Function	More Information
4.	Write a specified number of bytes from a user-defined buffer to the open smart large object.	**mi_lo_write( )**, **mi_lo_writewithseek( )**	page 6-42
5.	Pass the LO handle as the column value for an INSERT or UPDATE statement.	C Casting	page 6-42
6.	Execute an INSERT or UPDATE statement to save the LO handle of the smart large object in a database column.	**mi_exec( )**, **mi_exec_prepared_statement( )**, **mi_value( )**	page 6-42
7.	Close the smart large object.	**mi_lo_close( )**	page 6-49
8.	Free resources.	**mi_lo_spec_free( )**, **mi_lo_release( )**	page 6-43

Figure 6-3 shows the first six of these steps that a DataBlade API module uses to insert the smart-large-object data into the **emp_picture** column of the **employee** table (Figure 6-2 on page 6-14).
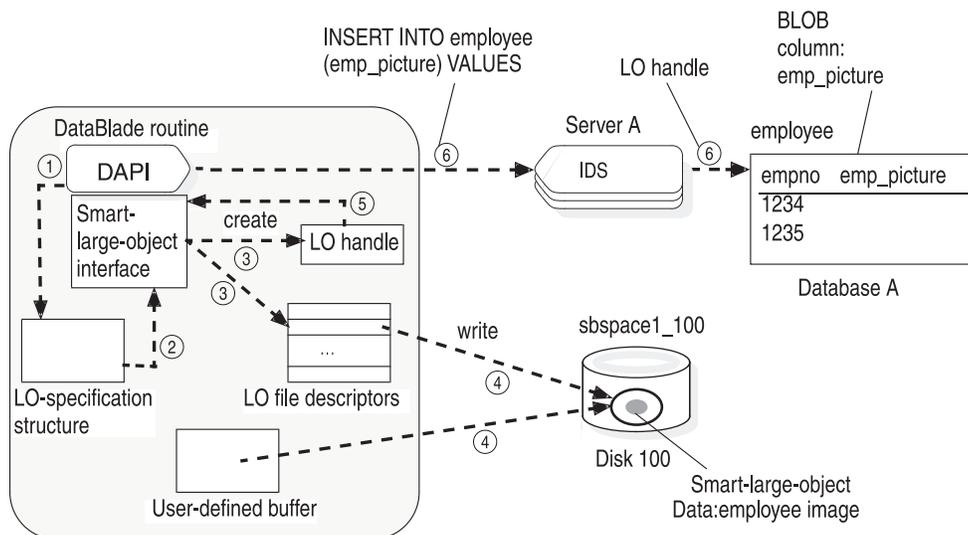
Figure 6-3. Inserting Into a BLOB Column

Obtaining the LO-Specification Structure

Before you create a new smart large object, obtain a valid LO-specification structure to hold its storage characteristics. You can obtain an LO-specification structure in either of the following ways:

- Create a new LO-specification structure to hold the storage characteristics of a *new* smart large object with the **mi_lo_spec_init( )** function.
- Obtain an LO-specification structure that holds the storage characteristics of an *existing* smart large object with the **mi_lo_stat_cspec( )** function.

Specifying New Storage Characteristics

The **mi_lo_spec_init( )** function is the constructor for the LO-specification structure. This function performs the following tasks to create a new LO-specification structure:

1. It allocates a new LO-specification structure when you provide a NULL-valued pointer as an argument.

2. It initializes all fields of the LO-specification structure (disk-storage information and attributes flag) to the appropriate null values.

Important: Do *not* handle memory allocation for an LO-specification structure with system memory-allocation routines (such as **malloc( )** or **mi_alloc( )**) or by direct declaration. You must use the LO-specification constructor, **mi_lo_spec_init( )**, to allocate a new LO-specification structure.

Allocating Memory for an LO-Specification Structure: When you pass a NULL-valued pointer as the second argument of the **mi_lo_spec_init( )** function, this function allocates an LO-specification structure.

$\qquad$ **Server Only** $\qquad$

This new LO-specification structure has the current memory duration.

$\qquad$ **End of Server Only** $\qquad$

The following code fragment declares a pointer named **myspec** and initializes this pointer to NULL:

```
MI_LO_SPEC *myspec;
MI_CONNECTION *conn;
...
/* Allocate a new LO-specification structure */
myspec = NULL;
if ( mi_lo_spec_init(conn, &myspec) != MI_OK )
   handle_error( );
/* Perform tasks with LO-specification structure */
...

/* Once finished with LO-specification structure, free it */
if ( mi_lo_spec_free(conn, myspec)!= MI_OK )
   handle_error( );
```

After the execution of **mi_lo_spec_init( )**, the **myspec** variable points to the newly allocated LO-specification structure. For more information on how to use an LO-specification structure to create a new smart large object, see "Choosing Storage Characteristics" on page 6-28.

If you provide a second argument that does not point to NULL, the **mi_lo_spec_init( )** function assumes that this pointer references an existing LO-specification structure that a previous call to **mi_lo_spec_init( )** has allocated. An LO-specification pointer that is not NULL allows a DataBlade API module to reuse an LO-specification structure. The following code fragment reuses the LO-specification structure that the **LO_spec** pointer references when the **first_time** flag is false:

```
MI_CONNECTION *conn;
MI_LO_SPEC *LO_spec = NULL;
mi_integer first_time = 1;
...
if ( first_time )
   {
   ...
   LO_spec = NULL; /* tell interface to allocate memory */
   first_time = 0; /* set "first_time" flag to false */
   ...
   }
```

```
if ( mi_lo_spec_init(conn, &LO_spec) != MI_OK )
   {
   /* error */
   }
```

Important: Before you use an LO-specification structure, make sure that you either
call **mi_lo_spec_init( )** with the LO-specification pointer set to NULL,
or that you have initialized this pointer with a previous call to
mi_lo_spec_init( ).

Once you have a valid LO-specification structure, you can use the accessor
functions to obtain the storage characteristics from this LO-specification structure.
For more information, see "Defining User-Specified Storage Characteristics" on
page 6-35. For the syntax of **mi_lo_spec_init( )**, see the *IBM Informix DataBlade API
Function Reference*.

Initializing an LO-Specification Structure: The **mi_lo_spec_init( )** function
initializes the LO-specification structure with values that obtain the
system-specified storage characteristics. The system-specified storage characteristics
are the defaults that the database server uses. They are the storage characteristics
at the bottom of the storage-characteristics hierarchy.

After this initialization, you can change the values in the LO-specification
structure:
* The new smart large object inherits column-level storage characteristics of a
  CLOB or BLOB column.
* You provide user-specified storage characteristics for the new smart large object.

For more information on storage characteristics and the storage-characteristics
hierarchy, see "Choosing Storage Characteristics" on page 6-28.

Copying Storage Characteristics from an Existing Smart Large Object

The **mi_lo_stat_cspec( )** function copies the create specification storage
characteristics from an existing smart large object to a flags field that can then be
passed to **mi_lo_create** to create a new smart large object. This function is used
when you want a new smart large object to have the same characteristics as an
existing smart large object.

The LO_stat structure in the following example holds status information for an
existing smart large object. You initialize an LO-status structure with the
mi_lo_stat( ) function. For more information on an LO-status structure, see
"Obtaining Status Information" on page 6-54.

The following code fragment assumes that the **old_LOfd** variable has already been
initialized as the LO file descriptor of an existing smart large object. This code
fragment uses the storage characteristics of the existing smart large object (which
the **mi_lo_stat( )** function puts into the MI_LO_STAT structure that **LO_stat**
specifies) as the create time storage characteristics for the new smart large object
that the **mi_lo_create( )** function creates.

```
MI_LO_HANDLE *LO_hdl = NULL;
MI_LO_STAT *LO_stat = NULL;
MI_LO_SPEC *LO_spec;
MI_LO_FD new_LOfd, old_LOfd;
mi_integer  flags;
...
if ( mi_lo_stat(conn, old_LOfd, &LO_stat) != MI_OK )
```

```
      {
      /* handle error and exit */
      }
LO_spec = mi_lo_stat_cspec(LO_stat);
new_LOfd = mi_lo_create(conn, LO_spec, flags, &LO_hdl);
```

Choosing Storage Characteristics

After initializing an LO-specification structure, you need to ensure that this structure contains the appropriate values for the storage characteristics you want the smart large object to have. Then you pass this LO-specification structure to one of the smart-large-object creation functions (Table 6-7 on page 6-20) so that the smart-large-object optimizer can obtain the storage characteristics to use for the new smart large object.

To choose storage characteristics for a new smart large object:

1. Use the system-specified storage characteristics as a basis for obtaining the storage characteristics of a smart large object.

   The *system-specified storage characteristics* are the default storage characteristics for a smart large object.

2. Customize the storage characteristics.

   You can override the system-specified storage characteristics with one of the following levels of the storage-characteristics hierarchy:

   • Storage characteristics defined for a particular CLOB or BLOB column in which you want to store the smart large object

     Storage characteristics that are unique to a particular CLOB or BLOB column are called *column-level storage characteristics*.

   • User-specified storage characteristics

     Special storage characteristics that you define for this smart large object only are called *user-specified storage characteristics*.

Important: For most applications, use the system-specified values for the disk-storage information. Most DataBlade API modules need to ensure correct storage characteristics only for an sbspace name (the location of the smart large object) and for the smart-large-object attributes.

Obtaining Storage Characteristics

For most smart large objects, all you need to do is obtain the system-specified storage characteristics. When you obtain these storage characteristics for a smart large object, you can specify a location for it and override system-specified attributes.

To obtain system-specified storage characteristics:

1. Use the **mi_lo_spec_init( )** function to allocate an LO-specification structure and to initialize this structure to the appropriate null values.

   When a storage characteristic in the LO-specification structure has the appropriate null value (zero or a NULL-valued pointer), the smart-large-object optimizer obtains the system-specified value for the storage characteristic. The smart-large-object optimizer calculates the system-specified values for disk-storage storage characteristics. Most applications can use these system-specified values. For more information, see "Using System-Specified Storage Characteristics" on page 6-32.

2. Specify the location of the smart large object to override the default location. You can specify the location as one of the following:

- The name of the sbspace associated with the CLOB or BLOB column in which you want to store the smart large object

  To store a new smart large object in a CLOB or BLOB column, use the **mi_lo_colinfo_by_name( )** or **mi_lo_colinfo_by_ids( )** function. These functions obtain the column-level storage characteristics for this column. One of the storage characteristics they obtain is the sbspace name for the column. For more information, see "Obtaining Column-Level Storage Characteristics" on page 6-33.

- The name of some other sbspace

  You might want to specify an sbspace name for a new smart large object that is embedded in an opaque data type. The **mi_lo_specset_sbspace( )** accessor function sets the name of the sbspace in the LO-specification structure. For more information, see "Defining User-Specified Storage Characteristics" on page 6-35.

3. Optionally, override any attributes for the smart large object with the **mi_lo_specset_flags( )** accessor function.

   The system-specified attributes have both logging and last-access time disabled. You might want to enable one or more attributes for the new smart large object. The **mi_lo_specset_flags( )** function sets the attributes flag in the LO-specification structure. For more information, see "Defining User-Specified Storage Characteristics" on page 6-35.

4. Pass this LO-specification structure to one of the smart-large-object creation functions (**mi_lo_create( )**, **mi_lo_copy( )**, **mi_lo_expand( )**, or **mi_lo_from_file( )**) to create the new smart large object.

   The smart-large-object creation function creates a new smart large object that has storage characteristics that the LO-specification structure indicates. For more information, see "Initializing an LO-Specification Structure" on page 6-27.

You would probably want to modify the storage characteristics of the new smart large object in the following cases:

- Your application needs to obtain extra performance.

  You can use other LO-specification accessor functions to change the disk-storage information of a new smart large object. For more information, see "Defining User-Specified Storage Characteristics" on page 6-35.

- You want to use the storage characteristics of an existing smart large object.

  The **mi_lo_stat_cspec( )** function can obtain the storage characteristics of an open smart large object through its LO-status structure. For more information, see "Copying Storage Characteristics from an Existing Smart Large Object" on page 6-27.

Using the Storage-Characteristics Hierarchy

Dynamic Server uses the *storage-characteristics hierarchy*, which Figure 6-4 shows, to obtain the storage characteristics for a new smart large object.

Database server storage characteristics
(system defaults)

System-specified
storage characteristics

Sbspace storage characteristics
(assigned when the sbspace is created with the **onspaces** utility
or when you change the sbspace with **onspaces -ch**)

Column-level storage characteristics
(assigned when the table is created with the CREATE TABLE statement
or when you change the table with the ALTER TABLE statement)

User-specified storage characteristics
(assigned when the smart large object is created with a DataBlade API
mi_lo_create( ) function or ESQL/C **ifx_lo_create( )** function)

Figure 6-4. Storage-Characteristics Hierarchy

For a given storage characteristic, any value defined at the column level overrides the system-specified value, and any user-level value overrides the column-level value. Table 6-9 summarizes the ways to specify disk-storage information for a smart large object.

Table 6-9. Specifying Disk-Storage Information

Disk-Storage Information	System-Specified Storage Characteristics		Column-Level Storage Characteristics	User-Specified Storage Characteristics
	System Default Value	Specified by onspaces Utility	Specified by the PUT clause of CREATE TABLE	Specified by a DataBlade API Function
Size of extent	Calculated by smart-large-object optimizer	EXTENT_SIZE	EXTENT SIZE	Yes
Size of next extent	Calculated by smart-large-object optimizer	NEXT_SIZE	No	No
Minimum extent size	Four kilobytes	MIN_EXT_SIZE	No	No
Size of smart large object	Calculated by smart-large-object optimizer	Average size of all smart large objects in sbspace: AVG_LO_SIZE	No	Estimated size of a particular smart large object Maximum size of a particular smart large object
Maximum size of I/O block	Calculated by smart-large-object optimizer	MAX_IO_SIZE	No	No

Table 6-9. Specifying Disk-Storage Information (continued)

Disk-Storage Information	System-Specified Storage Characteristics		Column-Level Storage Characteristics	User-Specified Storage Characteristics
	System Default Value	Specified by onspaces Utility	Specified by the PUT clause of CREATE TABLE	Specified by a DataBlade API Function
Name of sbspace	SBSPACENAME	-S *option*	Name of an existing sbspace that stores a smart large object: IN clause	Yes

For most applications, use the values for the disk-storage information that the smart-large-object optimizer determines. If you know the size of the smart large object, it is recommended that you specify this size as a user-specified storage characteristic, instead of as a system-specified or column-level storage characteristic.

For more information on any of the disk-storage information in Table 6-9, see "Disk-Storage Information" on page 6-5.

Table 6-10 summarizes the ways to specify attribute information for a smart large object.

Table 6-10. Specifying Attribute Information

Attribute Information	System-Specified Storage Characteristics		Column-Level Storage Characteristics	User-Specified Storage Characteristics
	System Default Value	Specified by the onspaces Utility	Specified by the PUT clause of CREATE TABLE	Specified by a DataBlade API Function
Logging	OFF	LOGGING	LOG, NO LOG	Yes
Last-access time	OFF	ACCESSTIME	KEEP ACCESS TIME, NO KEEP ACCESS TIME	Yes
Buffering mode	OFF	BUFFERING	No	Yes
Lock mode	Lock entire smart large object	LOCK_MODE	No	Yes
Data integrity	High integrity	No	HIGH INTEG, MODERATE INTEG	Yes

For more information on any of the attributes in Table 6-10, see "Attribute Information" on page 6-5.

Table 6-11 summarizes the ways to specify open-mode information for a smart large object.

Table 6-11. Specifying Open-Mode Information

| Storage Characteristic | System-Specified Storage Characteristics | | Column-Level Storage Characteristics | User-Specified Storage Characteristics |
	System Default Value	Specified by the onspaces Utility	Specified by the PUT clause of CREATE TABLE	Specified by a DataBlade API Function
Open-mode information	Default open mode	No	No	Yes

For more information on the open mode and the default open mode, see "Attribute Information" on page 6-5.

Using System-Specified Storage Characteristics: The Database Administrator (DBA) establishes system-specified storage characteristics when he or she initializes the database server and creates an sbspace with the **onspaces** utility, as follows:

- If the **onspaces** utility has specified a value for a particular storage characteristic, the smart-large-object optimizer uses the **onspaces** value as the system-specified storage characteristic.
- If the **onspaces** utility has *not* specified a value for a particular storage characteristic, the smart-large-object optimizer uses the system default as the system-specified storage characteristic.

The system-specified storage characteristics apply to *all* smart large objects that are stored in the sbspace, unless a smart large object specifically overrides them with column-level or user-specified storage characteristics.

The **onspaces** utility establishes storage characteristics for an sbspace. For the storage characteristics that **onspaces** can set as well as the system defaults, see Table 6-9 on page 6-30 and Table 6-10 on page 6-31. For example, the following call to the **onspaces** utility creates an sbspace named **sb1** in the **/dev/sbspace1** partition:

```
onspaces -c -S sb1 -p /dev/sbspace1 -o 500 -s 2000
   -Df "AVG_LO_SIZE=32"
```

Table 6-12 shows the system-specified storage characteristics for *all* smart large objects in the **sb1** sbspace.

Table 6-12. System-Specified Storage Characteristics for the sb1 Sbspace

Storage Characteristic	System-Specified Value	Specified by the onspaces Utility
Disk-storage information:		
Size of extent	Calculated by smart-large-object optimizer	system default
Size of next extent	Calculated by smart-large-object optimizer	system default
Minimum extent size	Calculated by smart-large-object optimizer	system default
Size of smart large object	32 kilobytes (smart-large-object optimizer uses as size estimate)	AVG_LO_SIZE
Maximum size of I/O block	Calculated by smart-large-object optimizer	system default

Table 6-12. System-Specified Storage Characteristics for the sb1 Sbspace (continued)

Storage Characteristic	System-Specified Value	Specified by the onspaces Utility
Name of sbspace	**sb1**	-S option
Attribute information:		
Logging	OFF	system default
Last-access time	OFF	system default

For a smart large object that has system-specified storage characteristics, the smart-large-object optimizer calculates values for all disk-storage information *except* the sbspace name. The DBA can specify a default sbspace name with the SBSPACENAME configuration parameter in the ONCONFIG file. However, you must ensure that the location (the name of the sbspace) is correct for the smart large object that you create. If you do *not* specify an sbspace name for a new smart large object, the database server stores it in this default sbspace. This arrangement can quickly lead to space constraints.

Important: For new smart large objects, use the system-specified values of all disk-storage information except the sbspace name. The smart-large-object optimizer can best determine most of the values of the storage characteristics. Most applications only need to specify an sbspace name for their disk-storage information.

Obtaining Column-Level Storage Characteristics: The DBA can establish column-level storage characteristics for a database table with the CREATE TABLE statement. If the table contains a CLOB or BLOB column, the PUT clause of CREATE TABLE can specify the storage characteristics that Table 6-9 on page 6-30 and Table 6-10 on page 6-31 show. This statement stores column-level storage characteristics in the **syscolattribs** system catalog table.

The column-level storage characteristics apply to *all* smart large objects whose LO handles are stored in the column, unless a smart large object specifically overrides them with user-specified storage characteristics. Column-level storage characteristics override any corresponding system-specified storage characteristics.

For example, if the **sb1** sbspace was defined as Table 6-12 on page 6-32 shows, the following CREATE TABLE statement specifies column-level storage characteristics of a location and last-access time for the **cat_descr** column:

```
CREATE TABLE catalog2
(
     catalog_num INTEGER,
     cat_descr CLOB
) PUT cat_descr IN (sb1) (KEEP ACCESS TIME);
```

Table 6-13 shows the storage characteristics for *all* smart large objects in the **cat_descr** column.

Table 6-13. Storage Characteristics for the cat_descr Column

Storage Characteristic	Column-Level Value	Specified by PUT Clause of CREATE TABLE
Disk-storage information:		

Table 6-13. Storage Characteristics for the cat_descr Column (continued)

Storage Characteristic	Column-Level Value	Specified by PUT Clause of CREATE TABLE
Size of extent	Calculated by smart-large-object optimizer	system-specified
Size of next extent	Calculated by smart-large-object optimizer	system-specified
Minimum extent size	Calculated by smart-large-object optimizer	system-specified
Size of smart large object	32 kilobytes (smart-large-object optimizer uses as size estimate)	system-specified
Maximum size of I/O block	Calculated by smart-large-object optimizer	system-specified
Name of sbspace	**sb1**	IN (sb1)
Attribute information:		
Logging	OFF	system-specified
Last-access time	ON	KEEP LAST ACCESS

For more information on the syntax of the CREATE TABLE statement, see its description in the *IBM Informix Guide to SQL: Syntax*.

The following DataBlade API functions obtain column-level storage characteristics for a specified CLOB or BLOB column:

- The **mi_lo_colinfo_by_name( )** function allows you to identify the column by the table and column name.
- The **mi_lo_colinfo_by_ids( )** function allows you to identify the column by an MI_ROW structure and the relative column identifier.

Both these functions store the column-level storage characteristics for the specified column in an existing LO-specification structure. When a smart-large-object creation function receives this LO-specification structure, it creates a new smart-large-object instance that has these column-level storage characteristics.

Tip: When you use the column-level storage characteristics, you do not usually need to override the name of the sbspace for the smart large object. The sbspace name is specified in the PUT clause of the CREATE TABLE statement.

For example, the following code fragment obtains the column-level storage characteristics for the **emp_picture** column of the **employee** table (Figure 6-2 on page 6-14) and puts them in the LO-specification structure that **LO_spec** references:

```
MI_LO_SPEC *LO_spec = NULL;
MI_CONNECTION *conn;
...
mi_lo_spec_init(conn, &LO_spec);
mi_lo_colinfo_by_name(conn, "employee.emp_picture",
   LO_spec);
```

The call to **mi_lo_colinfo_by_name( )** overwrites the system-specified storage characteristics that the call to **mi_lo_spec_init( )** put in the LO-specification structure. The LO-specification structure that **LO_spec** references now contains the column-level storage characteristics for the **emp_picture** column.

Defining User-Specified Storage Characteristics: You can establish user-specified storage characteristics when you create a new smart large object. DataBlade API functions can specify the storage characteristics that Table 6-9 on page 6-30 and Table 6-10 on page 6-31 show. The user-specified storage characteristics apply *only* to the particular smart-large-object instance that is being created. They override any corresponding column-level or system-specified storage characteristics.

After you have an LO-specification structure allocated, you can use the appropriate LO-specification accessor functions to set fields of this structure. Accessor functions also exist to retrieve storage-characteristic values from the LO-specification structure. When a smart-large-object creation function receives the LO-specification structure, it creates a new smart-large-object instance that has these user-specified storage characteristics.

Important: The *LO*-specification structure, **MI_LO_SPEC**, is an opaque structure to DataBlade API modules. Do not access its internal structure directly. The internal structure of **MI_LO_SPEC** may change in future releases. Therefore, to create portable code, always use the LO-specification accessor functions to obtain and store values from this structure.

The following sections describe how to access each group of storage characteristics in the LO-specification structure.

Accessing Disk-Storage Information: Table 6-14 shows the disk-storage information with the corresponding LO-specification accessor functions.

Table 6-14. Disk-Storage Information in the LO-Specification Structure

Disk-Storage Information	Description	LO-Specification Accessor Function
Estimated number of bytes	An estimate of the final size, in bytes, of the smart large object	**mi_lo_specget_estbytes( )**
		mi_lo_specset_estbytes( )
	The smart-large-object optimizer uses this value to determine the extents in which to store the smart large object. This value provides optimization information. If the value is grossly incorrect, it does not cause incorrect behavior. However, it does mean that the optimizer might not necessarily choose optimal extent sizes for the smart large object.	
	By default, this value is -1, which tells the smart-large-object optimizer to calculate the extent size from a set of heuristics.	
Maximum number of bytes	The maximum size, in bytes, for the smart large object	**mi_lo_specget_maxbytes( )**
	The smart-large-object optimizer does not allow the smart large object to grow beyond this size. By default, this value is -1, which tells the smart-large-object optimizer that there is no preset maximum size.	**mi_lo_specset_maxbytes( )**

Table 6-14. Disk-Storage Information in the LO-Specification Structure (continued)

Disk-Storage Information	Description	LO-Specification Accessor Function
Allocation extent size	The allocation extent size, in kilobytes	**mi_lo_specget_extsz( )**
	It is the size of the page extents for the smart large object. By default, this value is -1, which tells the smart-large-object optimizer to obtain the allocation extent size from the storage-characteristics hierarchy.	**mi_lo_specset_extsz( )**
Name of the sbspace	The name of the sbspace that contains the smart large object	**mi_lo_specget_sbspace( )**
		mi_lo_specset_sbspace( )
	The sbspace name can be at most 18 characters long and must be null terminated. By default, this value is null, which tells the smart-large-object optimizer to obtain the sbspace name from the storage-characteristics hierarchy.	

For most applications, use the values for the disk-storage information that the smart-large-object optimizer determines. If you know the size of the smart large object, it is recommended that you specify this size in the **mi_lo_specset_estbytes( )** function instead of in the **onspaces** utility or the CREATE TABLE or the ALTER TABLE statement. This **mi_lo_specset_estbytes( )** function (and the corresponding Informix ESQL/C **ifx_lo_specset_estbytes( )** function) is the best way to set the extent size because the database server can allocate the entire smart large object as one extent. For more information, see "Disk-Storage Information" on page 6-5.

Accessing Attributes: The LO-specification structure uses a bitmask flag, called an *attributes flag*, to specify the attributes of a smart large object. Table 6-15 shows the attribute constants of an LO-specification structure.

Table 6-15. Attribute Constants in the LO-Specification Structure

Attribute	Attribute Constant	Description
Logging:	MI_LO_ATTR_LOG	Log changes to the smart large object in the system log file.
	MI_LO_ATTR_NO_LOG	Turn off logging for all operations that involve the associated smart large object.
	Consider carefully whether to use the MI_LO_ATTR_LOG flag value. The database server incurs considerable overhead to log smart large objects. For more information, see "Logging" on page 6-5.	
Last-access time:	MI_LO_ATTR_KEEP_LASTACCESS_TIME	Save the last-access time for the smart large object.
	MI_LO_ATTR_NOKEEP_LASTACCESS_TIME	Do *not* maintain the last-access time for the smart large object.
	Consider carefully whether to use the MI_LO_ATTR_KEEP_LASTACCESS_TIME flag value. The database server incurs considerable overhead in logging and concurrency to maintain last-access times for smart large objects. For more information, see "Last-Access Time" on page 6-7.	

Table 6-15. Attribute Constants in the LO-Specification Structure (continued)

Attribute	Attribute Constant	Description
Data integrity:	MI_LO_ATTR_HIGH_INTEG	Use both a page header and a page trailer for the pages of the sbspace.
	MI_LO_ATTR_MODERATE_INTEG	Use only a page header for the pages of the sbspace.
	Consider carefully whether to use the MI_LO_ATTR_MODERATE_INTEG flag value. Although moderate integrity takes less disk space per page, it also reduces the ability of the database server to recover information should disk errors occur. For more information, see "Data Integrity" on page 6-7.	

The **milo.h** header file defines the attribute constants: MI_LO_ATTR_LOG, MI_LO_ATTR_NO_LOG, MI_LO_ATTR_KEEP_LASTACCESS_TIME, and MI_LO_ATTR_NOKEEP_LASTACCESS_TIME, MI_LO_ATTR_HIGH_INTEG, and MI_LO_ATTR_MODERATE_INTEG.

Table 6-16 shows the LO-specification accessor functions for the attribute information.

Table 6-16. Accessor Functions for Attribute Information in the LO-Specification Structure

LO-Specification Accessor Function	Description
mi_lo_specget_flags( )	Overrides system-specified or column-level attributes in the LO-specification structure with the attributes that the attributes flag specifies
mi_lo_specset_flags( )	Retrieves the attributes flag from the LO-specification structure

To set an attributes flag:

1. If you need to set more than one attribute, use the C-language bitwise OR operator (|) to mask attribute constants together.
2. Use the **mi_lo_specset_flags( )** accessor function to store the attributes flag in the LO-specification structure.

Masking mutually exclusive flags results in an error. If you do not specify a value for a particular attribute, the database server uses the storage-characteristics hierarchy to determine this information.

For example, the following code fragment specifies the constants to enable logging the last-access time for the attributes flag in the LO-specification structure that **LO_spec** identifies:

```
MI_CONNECTION *conn;
MI_LO_SPEC *LO_spec = NULL;
mi_integer create_flgs;
...

if ( mi_lo_spec_init(conn, &LO_spec) != MI_OK )
   /* handle error and exit */
create_flgs =
   MI_LO_ATTR_LOG | MI_LO_ATTR_KEEP_LASTACCESS_TIME;
if ( mi_lo_specset_flags(LO_spec, create_flgs) != MI_OK )
   /* handle error and exit */
```

For more information on the attributes of a smart large object, see "Attribute Information" on page 6-5 and the descriptions of the **mi_lo_specset_flags( )** and **mi_lo_specget_flags( )** functions in the *IBM Informix DataBlade API Function Reference*.

Accessing the Default Open Flag: When you open a smart large object, you can specify the *open mode* for the data. The open mode describes the context in which the I/O operations on the smart large object are performed. The LO-specification structure uses a bitmask flag, called a *default-open-mode flag*, to specify the default open mode of a smart large object. Table 6-17 shows the open-mode constants of an LO-specification structure.

Table 6-17. Open-Mode Constants in the LO-Specification Structure

Open-Mode Information	Open-Mode Constant	Description
Access modes	MI_LO_RDONLY	Read-only mode
	MI_LO_DIRTY_READ	Dirty-read mode
	MI_LO_WRONLY	Write-only mode
	MI_LO_APPEND	Write/append mode
	MI_LO_RDWR	Read/write mode
	MI_LO_TRUNC	Truncate

These access-mode flags for a smart large object are patterned after the UNIX System V file-access modes. For more information, see "Access Modes" on page 6-8.

Access methods	MI_LO_RANDOM	Random access
	MI_LO_SEQUENTIAL	Sequential access
	MI_LO_FORWARD	Forward
	MI_LO_REVERSE	Reverse

For more information, see "Access Methods" on page 6-9.

Buffering modes	MI_LO_BUFFER	Buffered access (Buffered I/O)
	MI_LO_NOBUFFER	Unbuffered access (Lightweight I/O)

For more information, see "Buffering Modes" on page 6-9.

Locking modes	MI_LO_LOCKALL	Lock-all locks
	MI_LO_LOCKRANGE	Byte-range locks

For more information, see "Locking Modes" on page 6-11.

The **milo.h** header file defines the open-mode constants: MI_LO_RDONLY, MI_LO_DIRTY_READ, MI_LO_WRONLY, MI_LO_APPEND, MI_LO_RDWR, MI_LO_TRUNC, MI_LO_RANDOM, MI_LO_SEQUENTIAL, MI_LO_FORWARD, MI_LO_REVERSE, MI_LO_BUFFER, MI_LO_NOBUFFER, MI_LO_LOCKALL, and MI_LO_LOCKRANGE.

Table 6-18 shows the LO-specification accessor functions for the default-open-mode information.

Table 6-18. Accessor Functions for Attribute Information in the LO-Specification Structure

LO-Specification Accessor Function	Description
mi_lo_specget_def_open_flags( )	Overrides the system default open mode with the open mode that the default-open-mode flag specifies
mi_lo_specset_def_open_flags( )	Retrieves the default-open-mode flag from the LO-specification structure

To set a default-open-mode flag:

1. Use the appropriate open-mode constants from the list in Table 6-17 on page 6-38. If you need to set more than one default-open-mode value, use the C-language bitwise OR operator (|) to mask open-mode constants together.

2. Use the **mi_lo_specset_def_open_flags( )** accessor function to store the default-open-mode flag in the LO-specification structure.

Masking mutually exclusive flags results in an error. However, you can mask the MI_LO_APPEND constant with another access-mode constant. In any of these OR combinations, the seek operation remains unaffected. The following table shows the effect that each of the OR combinations has on the read and write operations.

OR Operation	Read Operations	Write Operations
MI_LO_RDONLY | MI_LO_APPEND	Starts at the LO seek position and then moves the seek position to the end of the data that has been read	Fails and does not move the LO seek position
MI_LO_WRONLY | MI_LO_APPEND	Fails and does not move the LO seek position	Moves the LO seek position to the end of the smart large object and then writes the data. The LO seek position is at the end of the data after the write operation.
MI_LO_RDWR | MI_LO_APPEND	Starts at the LO seek position and then moves the seek position to the end of the data that has been read	Moves the LO seek position to the end of the smart large object and then writes the data. The LO seek position is at the end of the data after the write operation.

For more information on access modes of a smart large object, see "Access Modes" on page 6-8.

If you do not specify a value for a particular part of the open mode, the database server assumes the following system default open mode when you open a smart large object.

Access Capability	Default Open Mode	Smart-Large-Object Constant
Access mode	Read-only	MI_LO_RDONLY
Access method	Random	MI_LO_RANDOM
Buffering	Buffered access	MI_LO_BUFFER
Locking	Whole-object locks	MI_LO_LOCKALL

You can specify a different open mode for a particular smart large object when you open a smart large object. For more information on how to open a smart large object, see "Opening a Smart Large Object" on page 6-48.

Initializing an LO Handle and an LO File Descriptor

Once you have an LO-specification structure that describes the storage characteristics for the new smart large object, you can create the smart large object with one of the smart-large-object creation functions: **mi_lo_copy( )**, **mi_lo_create( )**, **mi_lo_expand( )**, or **mi_lo_from_file( )**. These smart-large-object creation functions perform the following tasks to create a new smart large object:

1. Initialize the LO handle for the new smart large object

   You provide a pointer to an LO handle as an argument to these functions. The creation functions initialize the LO handle with information about the location of the new smart large object.

2. Store the storage characteristics in a user-supplied LO-specification structure for the new smart large object in the metadata area of the sbspace

   You provide a pointer to an LO-specification structure as an argument to these functions. For more information, see "Obtaining the LO-Specification Structure" on page 6-25.

3. Open the new smart large object in the specified access mode

   You provide the open mode as an argument to the **mi_lo_create( )**, **mi_lo_copy( )**, or **mi_lo_expand( )** function. The **mi_lo_from_file( )** function opens a smart large object in read/write mode. For more information, see "Opening a Smart Large Object" on page 6-48.

4. Write any associated data to the new smart large object

   The **mi_lo_copy( )**, **mi_lo_expand( )**, and **mi_lo_from_file( )** function specifies data to write to the sbspace of the new smart large object.

5. Return an LO file descriptor that identifies the open smart large object

   The LO file descriptor is needed for most subsequent operations on the smart large object. However, this LO file descriptor is only valid within the current database connection.

These smart-large-object creation functions initialize the following data type structures for a smart large object:

- An LO handle, which identifies the location of the smart large object and can be stored in a CLOB, BLOB, or opaque-type column
- An LO file descriptor, which identifies the open smart large object

Obtaining an LO Handle

A DataBlade API module can obtain an LO handle with any of the following methods:

- Any of the smart-large-object creation functions can allocate an LO handle for a new smart large object.
- A DataBlade API module can explicitly allocate an LO handle.
- A SELECT statement can return an LO handle from a CLOB or BLOB column in the database.

For more information, see "Selecting the LO Handle" on page 6-47.

Implicitly Allocating an LO Handle: Any of the smart-large-object creation functions (Table 6-7 on page 6-20) can allocate memory for an LO handle when you specify a NULL-valued pointer for the last argument. For example, the following

code fragment declares a pointer to an LO handle named **LO_hdl**, initializes it to NULL, and then calls the **mi_lo_create( )** function to allocate memory for this LO handle:

```
MI_CONNECTION *conn;
MI_LO_SPEC *LO_spec;
MI_LO_HANDLE *LO_hdl = NULL; /* request allocation */
MI_LO_FD LO_fd;
mi_integer flags;
...
LO_fd = mi_lo_create(conn, &LO_spec, flags, &LO_hdl);
```

After the execution of **mi_lo_create( )**, the **LO_hdl** variable is a pointer to the new LO handle, which identifies the location of the new smart large object.

 Server Only

This new LO handle has a default memory duration of PER_ROUTINE. If you switch the memory duration, the creation function uses the current memory duration for the LO handle that it allocates.

--- **End of Server Only** ---

If you provide an LO-handle pointer that does not point to NULL, the smart-large-object creation function assumes that memory has already been allocated for the LO handle and it uses the LO handle that you provide.

Explicitly Allocating an LO Handle: You can explicitly allocate an LO handle in either of the following ways:

- Dynamically, with one of the DataBlade API memory-management functions such as **mi_alloc( )**:

  ```
  MI_LO_HANDLE *LO_hdl = mi_alloc(sizeof(MI_LO_HANDLE));
  ```

- On the stack, with a direct declaration:

  ```
  MI_LO_HANDLE my_LOhndl;
  MI_LO_HANDLE *LO_hndl2 = &my_LOhndl;
  ```

However, this LO handle is still an opaque C data structure; that is, it is declared as a flat array of undifferentiated bytes and its fields are not available to the DataBlade API module.

Important: The LO handle structure is the only smart-large-object structure that a DataBlade API module can allocate directly. You must allocate other smart-large-object data type structures, such as the LO-specification structure and the LO-status structure, with the appropriate DataBlade API constructor function.

Obtaining an LO File Descriptor

The smart-large-object creation functions (Table 6-7 on page 6-20) return an *LO file descriptor* for a smart large object. The LO file descriptor is needed for most subsequent operations on the smart large object. However, this LO file descriptor is only valid within the current database connection.

The following code fragment uses the **mi_lo_create( )** function to generate an LO file descriptor for a new smart large object:

```
MI_LO_FD LO_fd;
MI_LO_HANDLE *LO_hdl;
MI_LO_SPEC *LO_spec;
```

```
MI_CONNECTION *conn;
...
LO_fd = mi_lo_create(conn, LO_spec, MI_LO_RDONLY, &LO_hdl);
```

> **Tip:** A return value of zero (0) from a smart-large-object creation function does not indicate an error. The value zero (0) is a valid LO file descriptor.

Writing Data to a Smart Large Object

To write data to the sbspace of a smart large object, use one of the following smart-large-object functions:

- The **mi_lo_write( )** function begins the write operation at the *current* LO seek position.

  You can obtain the current LO seek position with the **mi_lo_tell( )** function, or you can set the LO seek position with the **mi_lo_seek( )** function.

- The **mi_lo_writewithseek( )** function performs the seek and write operations with a single function call.

  You specify the seek position at which to begin the write operation as arguments to **mi_lo_writewithseek( )**.

These functions both write a specified number of bytes from a user-defined character buffer to the open smart large object that an LO file descriptor identifies. The smart-large-object optimizer determines the default extent size for the smart large object based on the amount of data that you write. Therefore, try to maximum the amount of data you write in a single call to **mi_lo_write( )** or **mi_lo_writewithseek( )**.

> **Important:** An attempt to write data to an sbspace that does not exist results in an error.

In addition to a write operation, you might also need to perform the following operations on the open smart large object.

Task	Smart-Large-Object Function	More Information
Read data from the sbspace	**mi_lo_read( )**, **mi_lo_readwithseek( )**	page 6-48
Obtain the LO seek position	**mi_lo_tell( )**	page 6-48
Obtain status information	**mi_lo_stat( )**	page 6-54
Obtain storage characteristics	**mi_lo_stat_cspec( )**	page 6-28

Storing an LO Handle

The INSERT or UPDATE statement can store the LO handle of a smart large object into the CLOB, BLOB, or opaque-type column.

To store a smart large object in the database:

1. Provide the LO handle to the INSERT or UPDATE statement, as follows:
   - For a CLOB or BLOB column, provide the LO handle as data for the column.
   - For an opaque-type column, store the LO handle in the internal structure of the opaque data type and pass this internal structure as data for the column.
2. Execute the INSERT or UPDATE statement with a DataBlade API function such as **mi_exec( )** or **mi_exec_prepared_statement( )**.

Tip: The data of the smart large object is stored when you write it to the sbspace of the smart large object.

When you save the LO handle in the CLOB or BLOB column, the smart-large-object optimizer increments the reference count of the smart large object by one. When you save the LO handle in an opaque-type column, the **assign( )** support function for the opaque type must increment the reference count.

If you create a new smart large object but do *not* store it in a database column, the smart large object is a *transient smart large object*. The database server does not guarantee that transient smart large objects remain valid once they are closed. When all references to the smart large objects are deleted, the database server deletes the smart large object. For more information, see "Deleting a Smart Large Object" on page 6-56.

For more information on how to execute an INSERT or UPDATE statement, see Chapter 8, "Executing SQL Statements," on page 8-1.

Freeing Resources

After you store the new smart large object in the database, make sure that any resources you no longer need are freed. When you create a new smart large object, you might need to free resources of the following data type structures:

- The LO-specification structure
- The LO handle

———————— Server Only ————————

If any of the smart-large-object data type structures has a memory duration of PER_ROUTINE, the database server automatically frees the structure when the UDR completes.

———————— End of Server Only ————————

Freeing an LO-Specification Structure

———————— Server Only ————————

The **mi_lo_spec_init( )** function allocates an LO-specification structure in the current memory duration. Therefore, if an LO-specification structure has a memory duration of PER_ROUTINE, the database server automatically frees it when the UDR completes.

———————— End of Server Only ————————

To explicitly free the resources assigned to an LO-specification structure, use the **mi_lo_spec_free( )** function. The **mi_lo_spec_free( )** function is the destructor function for the LO-specification structure. When these resources are freed, they can be reallocated to other structures that your program needs.

Freeing an LO Handle

———————— Server Only ————————

The LO handle structure is allocated with the current memory duration. Therefore, if it has the default memory duration of PER_ROUTINE, the database server

automatically frees it when the UDR completes.

To explicitly free the resources assigned to an LO handle, you can use one of the following DataBlade API functions.

DataBlade API Function	Object Freed
mi_lo_release( )	Frees resources of a transient smart large object
	Frees an LO handle that the DataBlade API allocated
mi_free( )	Frees an LO handle that you have allocated
	If you allocate an LO handle with a DataBlade API memory-management function (such as **mi_alloc( )** or **mi_dalloc( )**), use **mi_free( )** to explicitly free the resources.
mi_lo_delete_immediate( )	Immediately frees the resources of a smart large object (rather than waiting for the end of the transaction)

When these resources are freed, they can be reallocated to other structures that your program needs.

Sample Code to Create a New Smart Large Object

Suppose you want to create a new smart large object for the **cat_descr** column in the **catalog2** table that contains the following data:

```
The rain in Spain stays mainly in the plain. In Hartford, Hereford, and
Hampshire, hurricanes hardly happen.
```

The following code fragment creates a new smart large object, which assumes the storage characteristics of its column, **cat_descr**, and then modifies the logging behavior:

```
#include "int8.h"
#include "mi.h"

#define BUFSZ 10000

{
   MI_CONNECTION *conn;
   MI_LO_SPEC *create_spec = NULL;
   MI_LO_HANDLE *descrip = NULL;
   MI_LO_FD lofd;
   char buf[BUFSZ];
   mi_integer buflen = BUFSZ;
   mi_int8 offset, est_size;
   mi_integer numbytes;
...
/* Allocate and initialize the LO-specification structure */
   if ( mi_lo_spec_init(conn, &create_spec) == MI_ERROR )
      handle_lo_error("mi_lo_spec_init( )");

/* Obtain the following column-level storage characteristics
 * for the cat_desc column:
 *      sbspace name = sb1 (this sbspace must already exist)
 *      keep last access time is ON
 */
   if ( mi_lo_colinfo_by_name(conn, "catalog2.cat_descr",
         create_spec) == MI_ERROR )
      handle_lo_error("mi_lo_colinfo_by_name( )");
```

```
/* Provide user-specified storage characteristics:
 *      logging behavior is ON
 *      size estimate of two kilobytes
 */
  mi_lo_specset_flags(create_spec, MI_LO_ATTR_LOG);
  ifx_int8cvint(2000, &est_size);
  mi_lo_specset_estbytes(create_spec, &est_size)

/* Create an LO handle and LO file descriptor for the new
 * smart large object
 */
  if ( lofd = mi_lo_create(conn, create_spec, MI_LO_RDWR,
        &descrip) == MI_ERROR )
    handle_lo_error("mi_lo_create( )");

/* Copy data into the character buffer 'buf' */
  sprintf(buf, "%s %s %s"
      "The rain in Spain stays mainly in the plain. ",
      "In Hartford, Hereford, and Hampshire, hurricanes",
      " hardly happen.");

/* Write contents of character buffer to the open smart
 * large object to which lofd points.
 */
  ifx_int8cvint(0, &offset);
  if ( numbytes = mi_lo_writewithseek(conn, lofd, buf,
        buflen, &offset, MI_LO_SEEK_SET) == MI_ERROR )
    handle_lo_error("mi_lo_writewithseek( )");

/* Close the LO file descriptor */
  mi_lo_close(conn, lofd);

/* Free LO-specification structure */
  mi_lo_spec_free(conn, create_spec);
```

After the **mi_lo_create( )** function executes, the following items are true:

- The **create_spec** LO handle was allocated and identifies the new smart large object.
- The **lofd** LO file descriptor identifies the open smart large object.
- The new smart large object has user-specified storage characteristics for logging behavior and estimated size.

The smart large object inherits the other storage characteristics. Table 6-19 shows the complete storage characteristics for this new smart large object.

Table 6-19. Storage Characteristics for the New Smart Large Object

Storage Characteristic	Value	Specified By
Disk-storage information:		
Size of extent	Calculated by smart-large-object optimizer	system-specified
Size of next extent	Calculated by smart-large-object optimizer	system-specified
Minimum extent size	Calculated by smart-large-object optimizer	system-specified
Size of smart large object	Two kilobytes (smart-large-object optimizer uses as size estimate)	**mi_lo_specset_estbytes( )**
Maximum size of I/O block	Calculated by smart-large-object optimizer	system-specified
Name of sbspace	**sb1**	column-level
Attribute information:		

Storage Characteristic	Value	Specified By
Logging	ON	**mi_lo_specset_flags( )** with MI_LO_ATTR_LOG
Last-access time	ON	column-level

Table 6-13 on page 6-33 shows the column-level storage characteristics for the **cat_descr** column and Table 6-12 on page 6-32 shows the system-specified storage characteristics for the **sb1** sbspace.

The **mi_lo_writewithseek( )** function writes the **buf** data to the smart large object that **lofd** identifies. When the write operation is successful, the **descrip** LO handle is ready to be stored in the CLOB column with the INSERT statement.

For more information on how to insert a value into a column, see Chapter 8, "Executing SQL Statements," on page 8-1.

Accessing a Smart Large Object

To access an existing smart large object in the database, you need to perform the following steps. For details on a step, see the page listed under "More Information."

Step	Task	Smart-Large-Object Function	More Information
1.	Execute a SELECT statement to obtain the LO handle of the smart large object from the CLOB or BLOB column.	**mi_exec( )**, **mi_exec_prepared_statement( )**, **mi_value( )**	page 6-47
2.	Convert the returned column value into an LO handle.	C Cast	page 6-60
3.	Open the smart large object that the LO handle identifies and return a valid LO file descriptor.	**mi_lo_open( )**	page 6-48
4.	Read a specified number of bytes and store them in a user-defined buffer.	**mi_lo_read( )**, **mi_lo_readwithseek( )**	page 6-48
5.	Close the smart large object.	**mi_lo_close( )**	page 6-49
6.	Free resources.	**mi_lo_release( )**	page 6-43

Figure 6-5 shows the first four of these steps that a DataBlade API module uses to access the smart-large-object data from the **emp_picture** column of the **employee** table (Figure 6-2 on page 6-14).
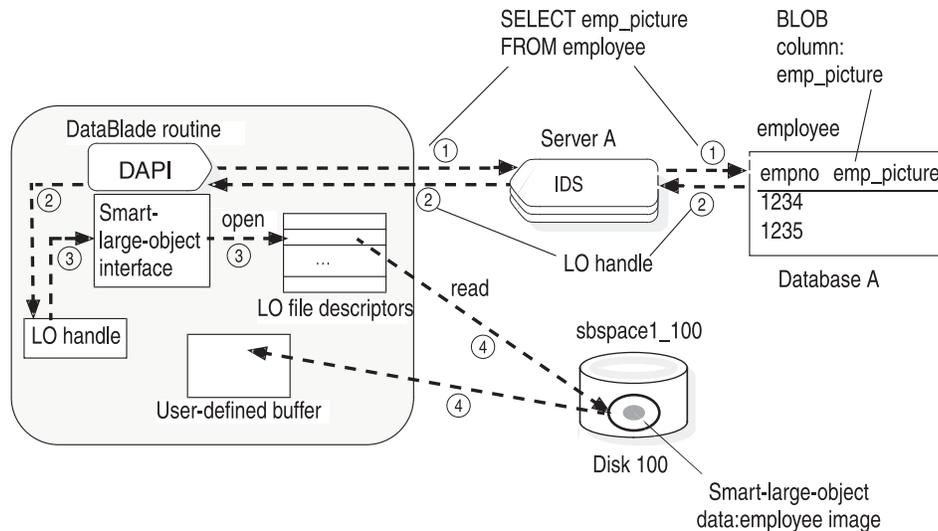
Figure 6-5. Selecting a BLOB Column

Selecting the LO Handle

The SELECT statement can select an LO handle of a smart large object from a CLOB, BLOB, or opaque-type column. Because the desired result of a query is usually the contents of an object, not just its LO handle, the DataBlade API module must then use the LO handle that the **mi_value( )** or **mi_value_by_name( )** function returns to access the smart-large-object data in its sbspace.

To select a smart large object from the database:

1. Execute the SELECT statement with a DataBlade API statement-execution function such as **mi_exec( )** or **mi_exec_prepared_statement( )**.
2. Obtain the column value that the **mi_value( )** or **mi_value_by_name( )** function passes back in the **MI_DATUM** structure as appropriate for the control mode of the query:
   - For binary representation, the **MI_DATUM** structure contains a pointer to an LO handle.
   - For text representation, the **MI_DATUM** structure contains the hexadecimal dump of an LO handle. To access the smart-large-object data, you must convert the LO handle to its binary representation with **mi_lo_from_string( )**.

   For more information, see "Binary and Text Representations of an LO Handle" on page 6-60.
3. Optionally, ensure that the LO handle is valid with **mi_lo_validate( )**.

For more information on how to select smart large objects, see "Accessing Smart Large Objects" on page 8-48.

Validating an LO Handle

An LO handle is valid when it correctly identifies the location of a smart large object in an sbspace. An LO handle might be invalid for either of the following reasons:

- The memory address is invalid or a NULL-valued pointer.
- The LO handle contains invalid reference data.

Use the **mi_lo_validate( )** function to check whether an LO handle is valid. If **mi_lo_validate( )** returns a positive integer, the LO handle is invalid. You can mark this LO handle as invalid with the **mi_lo_invalidate( )** function. The following code fragment checks whether the LO handle that **LO_hdl** references is valid:

```
if ( mi_lo_validate(conn, LO_hdl) > 0 )
   mi_lo_invalidate(conn, LO_hdl);
```

You can use the **mi_lo_validate( )** function in the support function of an opaque data type that contains smart large objects. In the **lohandles( )** support function, this function can determine unambiguously which LO handles are valid for the given instance of the opaque type.

Opening a Smart Large Object

You can open a smart large object with one of the following functions:

- The **mi_lo_open( )** function
- One of the smart-large-object creation functions: **mi_lo_copy( )**, **mi_lo_create( )**, **mi_lo_expand( )**, or **mi_lo_from_file( )**

These functions open the smart large object in a particular *open mode*, which in turn determines the *lock mode* of the smart large object. When you open a smart large object with the **mi_lo_copy( )**, **mi_lo_create( )**, **mi_lo_expand( )**, or **mi_lo_open( )** function, you tell the database server the open mode for the smart large object in either of the following ways:

- Provide an open mode of zero (0) as an argument to specify use of the default open mode of the smart large object.

  For information on how to associate a default open mode with a smart large object, see "Accessing the Default Open Flag" on page 6-38.

- Provide a non-zero open-mode argument to override the default open mode with an open mode you provide.

  Choose the appropriate open-mode constants from the list in Table 6-17 on page 6-38. If you need to set more than one open-mode value, use the C-language bitwise OR operator (|) to mask open-mode constants together.

Tip: The **mi_lo_from_file( )** function does not require an open mode for the smart large object it creates. It always opens a smart large object in read/write access mode. The smart-large-object optimizer determines which method of access is most efficient (buffered I/O or lightweight I/O).

All these open functions return an LO file descriptor, through which you can access the data of a smart large object as if it were in an operating-system file.

Reading Data from a Smart Large Object

To read data from the sbspace of a smart large object, use one of the following smart-large-object functions:

- The **mi_lo_read( )** function begins the read operation at the *current* LO seek position.

  You can obtain the current LO seek position with the **mi_lo_tell( )** function, or you can set the LO seek position with the **mi_lo_seek( )** function.

- The **mi_lo_readwithseek( )** function performs a seek to a *specified* LO seek position and then begins the read operation.

You specify the seek position at which to begin the read operation as arguments to **mi_lo_readwithseek( )**.

These functions both read a specified number of bytes from the open smart large object to a user-defined character buffer. For information on the syntax of the **mi_lo_read( )** and **mi_lo_readwithseek( )** functions, see the *IBM Informix DataBlade API Function Reference*.

You might also need to perform other operations on the open smart large object.

Task	Smart-Large-Object Function	For More Information
Write data to the sbspace	**mi_lo_write( )**, **mi_lo_writewithseek( )**	page 6-42
Obtain the LO seek position	**mi_lo_tell( )**	page 6-48
Obtain status information	**mi_lo_stat( )**	page 6-54
Obtain storage characteristics	**mi_lo_stat_cspec( )**	page 6-28

Freeing a Smart Large Object

A smart large object remains open until it is freed in either of the following ways:
- Explicitly, by a call to the **mi_lo_close( )** function
- Implicitly, when the current session ends

Once you finish the operations on the smart large object, you can close it explicitly with the **mi_lo_close( )** function. This function frees the resources associated with the LO file descriptor and LO handle so that they can be reallocated to other structures that your program needs. In addition, the LO file descriptor can be reassigned to another smart large object.

When you close a smart large object, you release any share-mode or update-mode locks on that object. However, you do not release exclusive locks until the end of the transaction. For more information, see "Locking Modes" on page 6-11.

Important: The end of a transaction does not close any smart large objects that are open. However, it does release any locks on the smart large objects.

If you do not explicitly close a smart large object, the database server closes it automatically at the end of the session. For information on the syntax of the **mi_lo_close( )** function, see the *IBM Informix DataBlade API Function Reference*. For more information on when the database server deletes a smart large object, see "Deleting a Smart Large Object" on page 6-56.

Sample Code to Select an Existing Smart Large Object

Suppose you want to select the following data from a smart large object that was inserted into a CLOB column named **cat_descr** in the **catalog2** table:

```
The rain in Spain stays mainly in the plain. In Hartford, Hereford, and
Hampshire, hurricanes hardly happen.
```

The following code fragment assumes that the **descrip** LO handle identifies the smart large object that was selected from the CLOB column. This LO handle was obtained with the SELECT statement on the **cat_descr** column.

```
#include "int8.h"
#include "mi.h"

#define BUFSZ 1000

{
   MI_CONNECTION *conn;
   MI_LO_HANDLE *descrip;
   MI_LO_FD lofd;
   char buf[BUFSZ];
   mi_integer buflen = BUFSZ;
   mi_int8 offset;
   mi_integer numbytes;
...

/* Use the LO handle to identify the smart large object
 * to open and get an LO file descriptor.
 */
   lofd = mi_lo_open(conn, descrip, MI_LO_RDONLY);
   if ( lofd < 0 )
      handle_lo_error("mi_lo_open( )");

/* Use the LO file descriptor to read the data of the
 * smart large object.
 */
   ifx_int8cvint(0, &offset);
   strcpy(buf, "");
   numbytes = mi_lo_readwithseek(conn, lofd, buf, buflen,
      &offset, MI_LO_SEEK_CUR);
   if ( numbytes == 0 )
      handle_lo_error("mi_lo_readwithseek( )");

/* Close the smart large object */
   mi_lo_close(lofd);
}
```

The **mi_lo_readwithseek( )** function reads 1000 bytes of data from the smart large
object that **lofd** identifies to the **buf** user-defined buffer.

For more information on how to select a value from a column, see "Accessing
Smart Large Objects" on page 8-48.

Modifying a Smart Large Object

Once you have an LO file descriptor for an open smart large object, you can
modify the smart large object, as follows:

• You can update the smart large object.
• You can alter some storage characteristics of the smart large object.

The following sections describe each of these tasks.

Updating a Smart Large Object

A smart large object has two parts: its LO handle and its data in the sbspace. You
can update either of these parts.

The UPDATE statement can store a new LO handle in a CLOB, BLOB, or
opaque-type column. For the steps to update a column, see "Storing an LO
Handle" on page 6-42.

To update an LO handle:

1. Update the column with a new smart large object.

   Overwrite the existing LO handle in the column with the LO handle for the new smart large object.

2. Store an additional reference to an existing smart large object.

   Multiple columns can reference the same smart large object on disk. You can overwrite an existing LO handle in the column with the LO handle for an existing smart large object. Both columns now reference the same smart large object.

To update the data of an existing smart large object:

1. Use the SELECT statement to obtain the LO handle that identifies the location of the data.

   For more information, see "Selecting the LO Handle" on page 6-47.

2. Open the smart large object to obtain an LO file descriptor.

   For more information, see "Opening a Smart Large Object" on page 6-48.

3. Read data from and write data to the open smart large object.

   For more information, see "Reading Data from a Smart Large Object" on page 6-48 and "Writing Data to a Smart Large Object" on page 6-42.

4. Close the smart large object.

   For more information, see "Freeing a Smart Large Object" on page 6-49.

Important: To update data of an existing smart large object, you do not need to use the UPDATE statement to update the CLOB, BLOB, or opaque-type column. The LO handle in the column does not need to change if you modify only the smart-large-object data.

Altering Storage Characteristics

After you create a smart large object, you can change some of its storage characteristics with the **mi_lo_alter( )** function. This function enables you to alter the following storage characteristics:

- Logging behavior
- Last-access time
- Extent size

All other storage characteristics *cannot* be changed once the smart large object is created. For information on the syntax of the **mi_lo_alter( )** function, see the *IBM Informix DataBlade API Function Reference*.

You can alter these storage characteristics in either of the following ways:

- Execute the SQL statement, ALTER TABLE.

  The PUT clause of ALTER TABLE enables you to modify any storage characteristics. However, any changes do not affect existing smart large objects; they only affect smart large objects in rows created after the ALTER TABLE statement executes. For more information, see the description of ALTER TABLE in the *IBM Informix Guide to SQL: Syntax*.

- Call the **mi_lo_alter( )** function.

  This function enables you to modify the logging characteristics, last-access time characteristics, and the extent size.

Obtaining Status Information for a Smart Large Object

To obtain the status information for an existing smart large object, take the following steps.

Step	Task	Smart-Large-Object Function	More Information
1.	Obtain a valid LO file descriptor for the smart large object whose status information you need.	**mi_lo_create( )**, **mi_lo_copy( )**, **mi_lo_expand( )**, **mi_lo_from_file( )** **mi_lo_open( )**	page 6-41
2.	Initialize an LO-status structure with the status information for the smart large object.	**mi_lo_stat( )**	page 6-53
3.	Use the appropriate LO-status accessor function to obtain the status information that you need.	Table 6-20 on page 6-54	page 6-54
4.	Free resources.	**mi_lo_stat_free( )**	page 6-55

Figure 6-6 shows the first three of these steps that a DataBlade API module uses to obtain status information for the smart large object data in the **emp_picture** column of the **employee** table (Figure 6-2 on page 6-14).
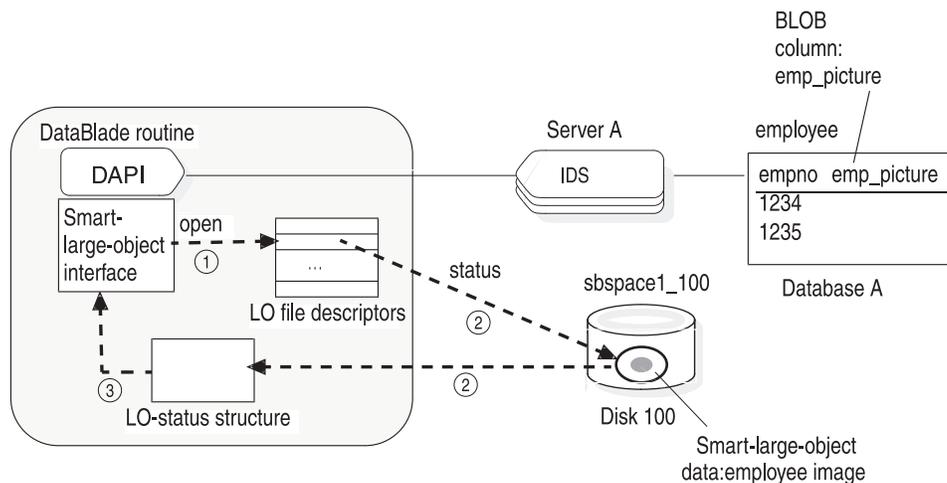
Figure 6-6. Obtaining Status Information

Obtaining a Valid LO File Descriptor

You can obtain status information for any smart large object for which you have a valid LO file descriptor. To obtain an LO file descriptor, you can take any of the following actions:

- Select an existing smart large object from a column in a database and open it

  For more information, see "Accessing a Smart Large Object" on page 6-46.

- Create a new smart large object

  For more information, see "Creating a Smart Large Object" on page 6-24.

- Receive the LO handle as an argument

A DataBlade API module can receive an argument that might provide the LO handle directly or it might provide an opaque data type in which the smart large object is embedded.

Initializing an LO-Status Structure

The **mi_lo_stat( )** function performs the following tasks:

1. It obtains either a new or existing LO-status structure.
2. It fills the LO-status structure with all status information for the smart large object that the specified LO file descriptor identifies.

Important: Do *not* handle memory allocation for an LO-status structure with system memory-allocation routines (such as **malloc( )** or **mi_alloc( )**) or by direct declaration. You must use the LO-status constructor, **mi_lo_stat( )**, to allocate a new LO-status structure.

Obtaining a Valid LO-Status Structure

The **mi_lo_stat( )** function is the constructor for the LO-status structure. The third argument to the **mi_lo_stat( )** function indicates whether to create a new LO-status structure:

- When you pass a NULL-valued pointer, the **mi_lo_stat( )** function allocates a new LO-status structure.

Server Only

This LO-status structure has the current memory duration.

End of Server Only

- When you pass a pointer that does *not* point to NULL, the **mi_lo_stat( )** function assumes that the pointer references an existing LO-status structure that a previous call to **mi_lo_stat( )** has allocated.

    An LO-status pointer that does not point to NULL allows a DataBlade API module to reuse an LO-status structure.

For example, the code fragment in Figure 6-7 uses the **mi_lo_stat( )** function to allocate memory for the LO-status structure *only* when the **first_time** flag is true.

```
MI_CONNECTION *conn;
MI_LO_HANDLE *LO_hdl;
MI_LO_STAT *LO_stat;
MI_LO_FD LO_fd;
mi_integer first_time;
...
LO_fd = mi_lo_open(conn, LO_hdl, MI_LO_RDONLY);
if ( first_time )
    {
    ...
    LO_stat = NULL; /* tell interface to allocate memory */
    first_time = 0; /* set "first_time" flag to false */
    ...
    }
err = mi_lo_stat(conn, LO_fd, &LO_stat);
```

Figure 6-7. Sample mi_lo_stat( ) Call

Filling the LO-Status Structure

Once **mi_lo_stat( )** has a pointer to a valid LO-status structure, it fills this structure with the status information for the open smart large object. You pass an LO file descriptor of the open smart large object as an argument to the **mi_lo_stat( )** function.

After the execution of **mi_lo_stat( )** in Figure 6-7, the **LO_stat** variable points to an allocated LO-status structure that contains status information for the smart large object that the LO file descriptor, **LO_fd**, identifies.

Important: Before you use an LO-status structure, make sure that you either call **mi_lo_stat( )** with the LO-status pointer set to NULL or initialize this pointer with a previous call to **mi_lo_stat( )**.

For more information, see Table 6-20. For the syntax of the **mi_lo_stat( )** function, see the *IBM Informix DataBlade API Function Reference*.

Obtaining Status Information

Once you have a valid LO-status structure, you can use the accessor functions to obtain the status information from this structure. Table 6-20 shows the status information that an LO-status structure contains and the corresponding LO-status accessor functions.

Table 6-20. Status Information in the LO-Status Structure

Status Information	LO-Status Accessor Function
Last-access time	**mi_lo_stat_atime( )**
This value is available *only* if the last-access time attribute (MI_LO_ATTR_KEEP_LASTACCESS_TIME) is set for this smart large object.	
Storage characteristics	**mi_lo_stat_cspec( )**
These characteristics are stored in an LO-specification structure. Use the LO-specification accessor functions (see "Defining User-Specified Storage Characteristics" on page 6-35) to obtain information from this structure.	
Last-change time	**mi_lo_stat_ctime( )**
Last-modification time	**mi_lo_stat_mtime_sec( )**, **mi_lo_stat_mtime_usec( )**
Reference count	**mi_lo_stat_refcnt( )**
Size	**mi_lo_stat_size( )**

Important: The *LO*-status structure, **MI_LO_STAT**, is an opaque structure to DataBlade API modules. Do not access its internal structure directly. The internal structure of **MI_LO_STAT** may change in future releases. Therefore, to create portable code, always use the LO-status accessor functions for this structure.

The following code fragment obtains the reference count from an LO-status structure that the **LO_stat** variable references:

```
MI_CONNECTION *conn;
MI_LO_HANDLE *LO_hdl;
MI_LO_FD LO_fd;
MI_LO_STAT *LO_stat = NULL; /* DataBlade API allocates */
```

```
                 mi_integer ref_count, err;
                 ...
                 /* Open the selected large object */
                 LO_fd = mi_lo_open(conn, LO_hdl, MI_LO_RDONLY);
                 if ( LO_fd == MI_ERROR )
                    /* handle error */

                 /* Allocate LO-specification structure and get status
                  * information for the opened smart large object
                  */
                 if ( mi_lo_stat(conn, LO_fd, &LO_stat) != MI_OK )
                    /* handle error */
                 else
                    {
                    /* get reference count for this smart large object */
                    ref_count = mi_lo_stat_refcnt(LO_stat);

                    /* free the LO-status structure */
                    err = mi_lo_stat_free(LO_stat);
                    }
```

The **mi_lo_open( )** function opens the smart large object that the LO handle,
LO_hdl, identifies. The **mi_lo_stat( )** function then obtains the status information
for this open smart large object. The **mi_lo_stat( )** function performs the following
tasks:

1. Allocates a new LO-status structure because the value of *__LO_stat__ is NULL

   The **mi_lo_stat( )** function assigns a pointer to this new LO-status structure to
   the **LO_stat** variable.

2. Initializes the **LO_stat** structure with the status information for the open smart
   large object that the LO file descriptor, **LO_fd**, identifies

Once the LO-status structure contains the status information, the
mi_lo_stat_refcnt( ) accessor function obtains the reference count from the
LO-status structure and returns it into the **ref_count** variable. When the code no
longer needs the LO-status structure, it frees this structure with the
mi_lo_stat_free( ) function.

Freeing an LO-Status Structure

─────────────────────────────── **Server Only** ───────────────────────────────

The **mi_lo_stat( )** function allocates an LO-status structure in the current memory
duration. Therefore, if the current memory duration is the default duration of
PER_ROUTINE, an LO-status structure has a memory duration of PER_ROUTINE
and the database server automatically frees it when the UDR completes.

─────────────────────────────── **End of Server Only** ───────────────────────────────

To explicitly free the resources assigned to an LO-status structure, use the
mi_lo_stat_free( ) function. The **mi_lo_stat_free( )** function is the destructor
function for an LO-status structure. When the resources are freed, they can be
reallocated to other structures that your program needs.

Deleting a Smart Large Object

The following table shows the methods that can cause a smart large object to be marked for deletion.

Location of Smart Large Object	Task for Deletion	Method for Deletion
For a CLOB or BLOB column	Remove the LO handle as data for the column	The DELETE statement
For an opaque-type column (for an opaque type that contains a smart large object)	Remove the LO handle from the internal structure of the opaque data type and store this revised internal structure as data for the column	The **destroy( )** support function
Transient smart large object	Remove the LO handle	Wait for the end of the session. The **mi_lo_delete_immediate( )** function

Alternatively, you can mark an LO handle as invalid with the **mi_lo_invalidate( )** function to indicate that it no longer identifies a valid smart large object.

When you delete an LO handle, the database server decrements the reference count of the smart large object that the LO handle references by one. The database server cannot delete a smart large object until it meets the following conditions:

- A reference count of zero

  To decrement the reference count of a smart large object, you delete an LO handle that references that smart large object. For more information, see "Managing the Reference Count" on page 6-56.

- No open LO file descriptors

  When the smart large object is closed, its LO file descriptor is freed. For more information, see "Freeing an LO Handle" on page 6-43.

Managing the Reference Count

The *reference count* of a smart large object is the number of LO handles that refer to the smart large object in its sbspace. Each LO handle contains the location of the smart large object in an sbspace. The reference count is stored with the smart-large-object data in an sbspace. (For more information on sbspaces, see your *IBM Informix Administrator's Guide*.) You can obtain the reference count with the **mi_lo_stat_refcnt( )** function.

A smart large object remains allocated as long as its reference count is greater than zero (0). A reference count greater than zero indicates that at least one column contains an LO handle that references the smart large object. In this sense, the smart large object is *permanent*. The management that the database server performs on a reference count depends on the associated smart large object:

- A smart large object whose LO handle is stored in a CLOB or BLOB column
- A smart large object whose LO handle is stored in an opaque data type
- A transient smart large object

Reference Counts for CLOB and BLOB Columns

For smart-large-object columns (CLOB and BLOB), the database server automatically manages the reference count, as follows:

- When you store an LO handle into a CLOB or BLOB column, the smart-large-object optimizer *increments* by one (1) the reference count for the smart large object that the LO handle identifies.
- When you delete an LO handle from a CLOB or BLOB column, the smart-large-object optimizer *decrements* by one (1) the reference count for the smart large object that the LO handle identifies.

At the end of the transaction, the smart-large-object optimizer automatically deletes all smart large objects stored in CLOB or BLOB columns with reference counts of zero and no open LO file descriptors.

Reference Counts for Opaque-Type Columns

The database server does *not* automatically manage the reference count for an opaque type that contains a smart large object (including multirepresentational opaque types). For these opaque-type columns, you must explicitly manage the reference count in special support functions of the opaque data type, as follows.

Support Function	Reference-Count Task	DataBlade API Function
assign( )	Increment the reference count by one each time a new LO handle for the smart large object is saved in the database.	**mi_lo_increfcount( )**
destroy( )	Decrement the reference count by one each time an LO handle that is stored in the database is removed from the database.	**mi_lo_decrefcount( )**
lohandles( )	If the opaque type does *not* have an **lohandles( )** support function, you must handle the reference count in the **assign( )** and **destroy( )** support functions.	
	If the opaque type has an **lohandles( )** support function, you do *not* need to handle the reference count in the **assign( )** and **destroy( )** support functions. The database server handles the decrement of the reference count when it executes the **lohandles( )** support function.	

If you increment or decrement the reference count for a smart large object within a transaction causing it to end up with a value of zero (0), the database server automatically deletes the smart large object at the end of the transaction (as long as it has no open LO file descriptors).

Reference Counts for Transient Smart Large Objects

A *transient* smart large object is one that you created but have *not* stored its LO handle in the database. Transient smart large objects can occur in the following ways:

- You create a smart large object (with **mi_lo_create( )**, **mi_lo_copy( )**, **mi_lo_expand( )**, or **mi_lo_from_file( )**) but do not insert its LO handle into a column of the database.
- You invoke a UDR that creates a smart large object in a query but never assigns its LO handle to a column of the database.

For example, the following query creates one smart large object for each row in the **table1** table:

```
SELECT FILETOBLOB(...) FROM table1;
```

However, the preceding query does not store the LO handles for these smart large objects in any database column. Therefore, each of these smart large objects is transient.

Important: A smart large object is "temporary" in the sense that it will automatically be deleted at the end of the session (unless its LO handle is stored in the database). A transient smart large object is not a smart large object that is stored in a temporary sbspace.

You only increment the reference count to tell the database server that the LO handle for the smart large object is going to be stored in the database (and become a permanent smart large object). Therefore, the reference count of a transient smart large object is zero. The database server deletes the transient smart large object at the end of the session.

You can explicitly deallocate the LO handle for a transient smart large object with the **mi_lo_release( )** function.

You can explicitly delete a transient smart large object with the **mi_lo_delete_immediate( )** function.

Freeing LO File Descriptors

An LO file descriptor exists until one of the following conditions occurs:

- You *explicitly* close a smart large object with the **mi_lo_close( )** function.

  When **mi_lo_close( )** closes a smart large object, the associated LO file descriptor is freed.

- The database server *implicitly* closes any open smart large objects at a session boundary (when the current database or connection closes).

  The resources that an open smart large object uses get automatically released at the end of a session. However, LO handles get released based on their memory duration. For more information on the memory duration of LO handles, see "Freeing an LO Handle" on page 6-43.

The effect of closing the LO file descriptors of a smart large object depends on whether the smart large object is permanent or transient:

- Closing a permanent smart large object

  When you close all its LO file descriptors, a permanent smart large object (one that is referenced by at least one column) remains allocated. The database server does *not* delete the data until the reference count is zero.

- Closing a transient smart large object

  However, when you close the last LO file descriptor for a transient smart large object, the database server marks the smart large object for deletion because both deallocation conditions are true:

  – The reference count of the transient smart large object is zero (0).

    The reference count of any transient smart large object is zero because it has no LO handles stored in the database. For more information, see "Managing the Reference Count" on page 6-56.

  – No LO file descriptors exist for the transient smart large object.

    Once you close the last open LO file descriptor (explicitly or implicitly), no more references to this smart large object exist, and the data is not kept.

Converting a Smart Large Object to a File or Buffer

The DataBlade API provides support for the conversion of a smart large object to or from either of the following structures:

- Operating-system file
- User-defined buffer

Using Operating-System Files

The DataBlade API supports the following types of functions for conversion between operating-system files and smart large objects.

DataBlade API Function	Description
mi_lo_from_file( ), **mi_lo_from_file_by_lofd( )**	Copies data in an operating-system file to a smart large object
mi_lo_to_file( ), **mi_lo_filename( )**	Copies data in a smart large object to an operating-system file

The file functions have a set of file-mode constants that are distinct from the open modes of smart large objects, as the table in Table 6-21 shows.

Table 6-21. File Modes for Operating-System Files

File Mode for Operating-System Files	Purpose
MI_O_EXCL	Fail if the file already exists
MI_O_APPEND	Append to the end of file
MI_O_TRUNC	Truncate to zero if file exists
MI_O_RDWR	Read/write mode (default)
MI_O_RDONLY	Read-only mode (copying from operating-system files only)
MI_O_WRONLY	Write-only mode (copying to operating-system files only)
MI_O_TEXT	Text mode (default off)
MI_O_CLIENT_FILE	Indication that file is on client computer (default)
MI_O_SERVER_FILE	Indication that file is on server computer

You can include an environment variable in the filename path for the **mi_lo_to_file( )**, **mi_lo_from_file( )**, and **mi_lo_from_file_by_lofd( )** functions. This environment variable must be set in the server environment; that is, it must be set *before* the database server starts.

Using User-Defined Buffers

The DataBlade API supports the following functions for conversion between user-defined buffers and smart large objects.

DataBlade API Function	Description
mi_lo_from_buffer( )	Copies data in a user-defined buffer to a smart large object
mi_lo_to_buffer( )	Copies data in a smart large object to a user-defined buffer

Converting an LO Handle Between Binary and Text

The DataBlade API library provides functions that convert between the binary (internal) representation of an LO handle and its text (string) representation.

Binary and Text Representations of an LO Handle

The **MI_LO_HANDLE** data type (for an LO handle) is an opaque C data structure with a length of MI_LO_SIZE. The binary representation of the LO handle is a flat array of MI_LO_SIZE bytes. You can perform the following actions on the binary representation of an LO handle:

- Store it in a C variable of type **MI_LO_HANDLE**.
- Pass it to a UDR.
- Bind it an **MI_LO_HANDLE** variable to hold a smart large object retrieved by a query whose control mode is binary representation (for example, in **mi_exec( )**).
- Store it in a CLOB or BLOB column of the database.
- Send it as part of the internal (binary) representation of an opaque type.

The text representation of an LO handle is the text hexadecimal dump of the flat binary array. To represent the hexadecimal format, each binary byte requires two bytes of characters. You can perform the following actions on the text representation of an LO handle:

- Store it in a C character string or array.
- Bind it to a character-pointer variable to hold a smart large object retrieved by a query whose control mode is text representation (for example, in **mi_exec( )**).
- Store it in a CHAR (or other character-based) column in a database.

DataBlade API Functions for LO-Handle Conversion

The DataBlade API provides the following functions for conversion between binary and text representations of an LO handle.

DataBlade API Function	Converts from	Converts to
mi_lo_to_string( )	LO handle (**MI_LO_HANDLE**)	Text representation of LO handle
mi_lo_from_string( )	Text representation of LO handle	LO handle (**MI_LO_HANDLE**)

────────────────────────────── **Server Only** ──────────────────────────────

The **mi_lo_to_string( )** and **mi_lo_from_string( )** functions are useful in the input and output support function of an opaque data type that contains smart large objects. These functions enable you to convert CLOB and BLOB values (LO handles) between their external format (text) and their internal format (binary) when you transfer them to and from client applications. For more information, see "Conversion of Opaque-Type Data Between Text and Binary Representations" on page 16-16.

────────────────────────────── **End of Server Only** ──────────────────────────────

Transferring an LO Handle Between Computers (Server)

For an LO handle to be portable when transferred across different computer architectures, the DataBlade API provides the following functions to handle type alignment and byte order.

DataBlade API Function	Description
mi_get_lo_handle( )	Copies an aligned LO handle, converting any difference in alignment or byte order on the client computer to that of the server computer
mi_put_lo_handle( )	Copies an aligned LO handle, converting any difference in alignment or byte order on the server computer to that of the client computer

The **mi_get_lo_handle( )** and **mi_put_lo_handle( )** functions are useful in the send and receive support function of an opaque data type that contains a smart large object. They enable you to ensure that BLOB or CLOB values (LO handles) remained aligned when transferred to and from client applications. For more information, see "Conversion of Opaque-Type Data with Computer-Specific Data Types" on page 16-21.

Using Byte-Range Locking

By default, the database server uses whole lock-all locks when it needs to lock a smart large object. Lock-all locks are an "all or nothing" lock; that is, they lock the entire smart large object. When the database server obtains an exclusive lock, no other user can access the data of the smart large object as long as the lock is held. (For more information on the default locking, see "Locking Modes" on page 6-11.)

If this locking is too restrictive for the concurrency requirements of your application, you can use *byte-range locking* instead of lock-all locking. With byte-range locking, you can specify the range of bytes to lock in the smart-large-object data. If other users access other portions of the data, they can still acquire their own byte-range lock.

To use byte-range locking:

1. Enable the byte-range locking feature on the smart large object you need to lock.

   You can specify the byte-range locking feature either when you create the smart large object or when you open it, as follows:

   • At the time of smart-large-object creation

   You can specify the LO_LOCKRANGE lock-mode constant as a default open flag for the new smart large object.

   • When you open the smart large object

   You can specify the LO_LOCKRANGE lock-mode constant in the open-mode argument of **mi_lo_open( )**.

2. Handle the lock requests for the byte-range locks with the appropriate function of the smart-large-object interface.

   The smart-large-object interface provides the following functions for handle lock requests of byte-range locks.

Byte-Range Locking Function   Description

mi_lo_lock( )	Obtains a byte-range lock on the specified number of bytes in a smart large object
mi_lo_unlock( )	Releases a byte-range lock on a smart large object

With the **mi_lo_lock( )** function, you can specify the following information for the lock request of the byte-range lock:

- The location in the smart-large-object data at which to begin the byte-range lock
- The number of bytes to lock
- The type of lock to obtain: shared or exclusive lock

Passing a NULL Connection (Server)

Many functions in the smart-large-object interface take a connection descriptor as a parameter. However, many of the functions also accept a NULL-valued pointer as a connection descriptor. Use of a NULL-valued connection descriptor has the following performance impact:

- The smart-large-object functions do not need to check the validity of a connection descriptor.
- The calling code is not required to open and close a connection.

To improve performance, you can pass a NULL-valued pointer as a connection descriptor to any of the following functions of the smart-large-object interface:

mi_lo_alter( )	**mi_lo_spec_init( )**
mi_lo_close( )	**mi_lo_stat( )**
mi_lo_colinfo_by_ids( )	**mi_lo_stat_free( )**
mi_lo_colinfo_by_name( )	**mi_lo_tell( )**
mi_lo_copy( )	**mi_lo_to_buffer( )**
mi_lo_create( )	**mi_lo_to_file( )**
mi_lo_decrefcount( )	**mi_lo_truncate( )**
mi_lo_delete_immediate( )	**mi_lo_unlock( )**
mi_lo_expand( )	**mi_lo_utimes( )**
mi_lo_filename( )	**mi_lo_validate( )**
mi_lo_from_buffer( )	**mi_lo_write( )**
mi_lo_from_file( )	**mi_lo_writewithseek( )**
mi_lo_from_file_by_lofd( )	**mi_lo_ptr_cmp( )**
mi_lo_increfcount( )	**mi_lo_read( )**
mi_lo_invalidate( )	**mi_lo_readwithseek( )**
mi_lo_lock( )	**mi_lo_release( )**
mi_lo_lolist_create( )	**mi_lo_seek( )**
mi_lo_open( )	**mi_lo_spec_free( )**

The following code fragment passes a valid connection descriptor to the **mi_lo_alter( )** function:

```
conn = mi_open(NULL, NULL, NULL);
if ( mi_lo_alter(conn, LO_ptr, LO_spec) == MI_ERROR )

/* Code execution does not reach here when a database server
 * exception occurs.
 */
   return MI_ERROR;
mi_close(conn);
```

When you specify a NULL-valued pointer as a connection descriptor, you can omit the calls to **mi_open( )** and **mi_close( )**, as the following code fragment shows:

```
      if ( mi_lo_alter(NULL, LO_ptr, LO_spec) == MI_ERROR )

/* Code execution does not reach here when a database server
 * exception occurs.
 */
   return MI_ERROR;
```

Part 3. Database Access

Chapter 7. Handling Connections

In This Chapter

When a DataBlade API module begins execution, it has no communication with a database server; however, for SQL statements to execute, such communication must exist. To establish this communication, a DataBlade API module must take the following steps:

―――――――――――――――― **Client Only** ――――――――――――――――

1. Initialize a connection to the database server and, optionally, a database.

―――――――――――――――― **End of Client Only** ――――――――――――――――

2. Establish the connection and initialize the DataBlade API.

To end the communication, the DataBlade API module must close the connection.

This chapter describes how to initialize, establish, and close connections.

Understanding Session Management

Session management is the handling of a connection to a database server and the associated session within a DataBlade API module. A *session* is the period of time that elapses between when a client application establishes a connection and when this connection is closed.

A *connection* is the mechanism over which a DataBlade API module communicates with the database server and, optionally, a database that this database server manages. The term *connection* has different meanings depending on whether it refers to a connection within a client LIBMI application or a C user-defined routine (UDR), as follows:

- A *client connection* is established by a client application to a particular database server and database.
- A *UDR connection* is established by a C UDR to obtain information about the current session.

Client Connection

For a client application, a *connection* is the mechanism that the application uses to request a synchronization with the database server for the purpose of exchanging data. A client application (such as an Informix ESQL/C or client LIBMI application) takes the following steps to request a connection:

1. Initializes the connection

   A client LIBMI application can set connection, database, and session parameters to determine attributes of the connection.

   For more information, see "Initializing a Client Connection" on page 7-4.

2. Establishes the connection

   An Informix ESQL/C client application uses SQL statements (such as CONNECT or DATABASE) to establish a connection. A client LIBMI application uses the DataBlade API to establish a connection in either of the following ways:

   - Uses the **mi_open( )** function
   - Uses the **mi_server_connect( )** function

   For more information, see "Establishing a Client Connection" on page 7-14.

When a client application connects to the database server, the database server performs the following tasks:

- Creates a session structure, called a *session control block*, to hold information about the connection and the user
- Creates a thread structure, called a *thread-control block* (TCB), to hold information about the current state of the thread
- Determines the server-processing locale, the locale to use for SQL statements during the session
- Initializes a primary thread, called the *session thread* (or **sqlexec** thread), to handle client-application requests

When the client application successfully establishes a connection, it begins a *session*. Only a client application can begin a session. The *session context* consists of data structures and state information that are associated with a specific session, such as cursors, save sets, and user data.

UDR Connection (Server)

A C UDR cannot establish a connection to the database server. It must run within an existing session, which a client application begins when it establishes a client connection. To obtain access to an existing session, a UDR establishes a *UDR connection*. A UDR uses the **mi_open( )** function (with NULL-valued pointers for *all* arguments) to establish a UDR connection. A UDR can have one or more UDR connections to the session.

Because a C UDR cannot establish a connection, it never begins a session. Instead, it inherits the context of an existing session from the SQL statement that invokes the UDR. This SQL statement executes within a session that the client application has begun. Therefore, the UDR is already connected to a particular database server and has access to a database that has already been opened. The UDR is also inside the current transaction.

A UDR connection provides access to the session context. This session context persists across all invocations of a UDR instance because a UDR connection has the duration of an SQL command. That is, the **mi_open( )** function and its corresponding **mi_close( )** function do not necessarily have to be called from within the same invocation of the UDR but they do have to be called within the same instance. For more information, see "Establishing a UDR Connection (Server)" on page 7-11.

Connection Descriptor

A DataBlade API module (C UDR or client LIBMI application) obtains access to the session context through a *connection descriptor*, **MI_CONNECTION**. This descriptor is an opaque C data structure that points to the threadsafe context-sensitive portion of the session information. The session context includes the information in Table 7-1.

Table 7-1. Session-Context Information in a Connection Descriptor

Connection Information	More Information
Save sets	"Using Save Sets" on page 8-59
Statement descriptors: • For SQL statement, last executed with **mi_exec( )** • For prepared statements	"Executing Basic SQL Statements" on page 8-6 "Executing Prepared SQL Statements" on page 8-11
Cursors (implicit and explicit)	"Queries and Implicit Cursors" on page 8-5 and "Defining an Explicit Cursor" on page 8-22
Resources for the current row of the current statement: • Row descriptor • Row structure	"Retrieving Query Data" on page 8-39
Function descriptors	"Obtaining a Function Descriptor" on page 9-17
Callbacks registered for the connection	"Registering a Callback" on page 10-4
User data	"Associating User Data with a Connection" on page 7-16
The integer byte order of the client computer	"Conversion of Opaque-Type Data with Computer-Specific Data Types" on page 16-21

You obtain a connection descriptor when you establish a connection in the DataBlade API module. For more information, see "Establishing a Connection" on page 7-11.

Initializing a Client Connection

Before a client LIBMI application can establish a connection, it must initialize the connection with the name of the database server and database to which it needs to connect. This initialization occurs in the following steps.

Connection-Initialization Steps	DataBlade API Task
1. Indicate the database server to which you want to connect.	Set *connection parameters* in the connection-information descriptor.
2. Indicate the database to which you want to connect and the user you want to log in as.	Set *database parameters* in the database-information descriptor.
3. Indicate settings for session-specific features (*optional*).	Set *session parameters* in the parameter-information descriptor.

Using Connection Parameters

To indicate which database server the client LIBMI application needs to connect to, the application uses *connection parameters*. The DataBlade API provides a *connection-information descriptor*, **MI_CONNECTION_INFO**, to access connection parameters. This data type structure is similar in concept to a file descriptor in UNIX. It identifies the database server for a particular session.

Unlike most DataBlade API structures, the connection-information descriptor is *not* an opaque C data structure. To access connection parameters, you must allocate a connection-information descriptor and directly access its fields. Table 7-2 shows the fields in the connection-information descriptor.

Table 7-2. Fields in the Connection-Information Descriptor

Field	Data Type	Description
server_name	**char ***	The name of the default database server
		This field corresponds to the value of the **INFORMIXSERVER** environment variable.
server_port	**mi_integer**	This value is ignored. It must always be set to zero (0).
		A client LIBMI application does *not* need to specify the server port. It only needs to specify a database server by its name (the **server_name** field).
locale	**char ***	In a C UDR: The name of the server locale
		This field corresponds to the **SERVER_LOCALE** environment variable (as set on the computer with the database server).
		In a client LIBMI application: the name of the database locale
		This field corresponds to the **DB_LOCALE** environment variable (as set on the computer with the client LIBMI application).
reserved1	**mi_integer**	Unused
reserved2	**mi_integer**	Unused

The **milib.h** header file defines the **MI_CONNECTION_INFO** structure.

With the connection-information descriptor, you can use the following DataBlade API functions to perform the connection-parameter tasks.

Connection-Parameter Task	DataBlade API Function
Access the *default* connection parameters to determine the database server for the connection	**mi_set_default_connection_info( )**, **mi_get_default_connection_info( )**
Obtain *current* connection parameters for an open connection	**mi_get_connection_info( )**

Establishing Default Connection Parameters

The default connection parameters identify to which database server to connect. Before you establish a connection, determine which of the following connection parameters to use:

- The system-default connection parameters
- Default connection parameters that you specify

System-Default Connection Parameters: The database server obtains values for the system-default connection parameters from the execution environment of the client LIBMI application. When you use system-default connection parameters, you enable your application to be portable across client/server environments. However, before the application begins execution, you must ensure that the client/server environment is correctly initialized.

Table 7-3 shows the system-default connection parameters that the database server uses to establish a connection.

Table 7-3. System-Default Connection Parameters

System-Default Connection Parameter	System-Default Connection Value
Database server name	**INFORMIXSERVER** environment variable
(Server) Server locale	**SERVER_LOCALE** environment variable or default locale (**en_us**)
(Client) Database locale	**DB_LOCALE** environment variable or default locale (**en_us**)

The system-default connection parameters provide connection information for *all* connections made within a client LIBMI application unless you explicitly override them within the application.

To use the default database server, initialize the **server_name** field to a NULL-valued pointer and **server_port** to 0.

User-Defined Connection Parameters: The database server obtains values for the connection parameters from the connection-information descriptor. The database server initializes the connection-information descriptor with the system-default connection parameters in Table 7-7 on page 7-9. You can initialize your own connection-information descriptor to override the system-default connection parameters. When you override system-default connection parameters, you enable your application to have connection information that is independent of the client/server environment in which it runs.

To override the system-default connection parameters:

1. Allocate a connection-information descriptor.

2. Fill the fields of the connection-information descriptor with the default connection parameters you need.

   To change the database server, specify a value for **server_name**. Any non-zero value for the **server_port** field is ignored. If you do not set a particular field, the database server uses the system-default value in Table 7-3 on page 7-5 for the associated connection parameter.

3. Pass a pointer to this connection-information descriptor to the **mi_set_default_connection_info( )** function.

The user-defined connection parameters provide connection information for *all* connections made within a client LIBMI application *after* these functions execute (unless the functions are called again to set new default values).

You can obtain existing default connection parameters with the **mi_get_default_connection_info( )** function. This function populates a user-defined connection-information descriptor with the current default connection parameters.

───────────────────────── **Server Only** ─────────────────────────

In a C UDR, **mi_get_default_connection_info( )** obtains the same information as **mi_get_connection_info( )**. The **mi_set_default_connection_info( )** function is ignored when it is used in a UDR.

───────────────────────── **End of Server Only** ─────────────────────────

Obtaining Current Connection Parameters

To obtain connection parameters associated with an open connection, use the **mi_get_connection_info( )** function. This function populates a user-defined connection-information descriptor with values from the specified open connection.

───────────────────────── **Server Only** ─────────────────────────

The **mi_get_connection_info( )** function is valid when it is used in a C UDR. For more information, see "Accessing the Session Environment" on page 13-58.

───────────────────────── **End of Server Only** ─────────────────────────

Using Database Parameters

To indicate which database it needs to connect to, the client LIBMI application uses *database parameters*. The DataBlade API provides a *database-information descriptor*, **MI_DATABASE_INFO**, to access database parameters. This data type structure identifies the database for a particular session.

Unlike most DataBlade API structures, the database-information descriptor is *not* an opaque C data structure. To access database information, you must allocate a database-information descriptor and directly access its fields. Table 7-4 shows the fields in the database-information descriptor.

Table 7-4. Fields in the Database-Information Descriptor

Field	Data Type	Description
database_name	char *	The name of the database
user_name	char *	The user account name, as defined by the operating system

Table 7-4. Fields in the Database-Information Descriptor (continued)

Field	Data Type	Description
password	char *	The account password, as defined by the operating system

The **milib.h** header file defines the **MI_DATABASE_INFO** structure.

With the database-information descriptor, you can use the following DataBlade API functions to perform the database-parameter tasks.

Database-Parameter Task	DataBlade API Function
Access the *default* database parameters to determine the database and user for the connection	**mi_set_default_database_info( )**, **mi_get_default_database_info( )**
Obtain *current* database parameters for an open connection	**mi_get_database_info( )**

Establishing Default Database Parameters

The default database parameters identify the database and user for the connection. Before you establish a connection, you can determine which of the following database parameters to use:

- The system-default database parameters
- Default database parameters that you specify

System-Default Database Parameters: The database server obtains values for the system-default database parameters from the execution environment of the client LIBMI application. When you use system-default database parameters, you enable your application to be portable across client/server environments. However, you must ensure that the client/server environment is correctly initialized to provide the system-default values.

Table 7-5 shows the system-default database parameters that the database server uses to open a database.

Table 7-5. System-Default Database Parameters

System-Default Database Parameter	System-Default Value
Database name	None
User-account name	Account name of user that invoked the client LIBMI application
Account password	Account password of user that invoked the client LIBMI application

The system-default database parameters provide database information for *all* connections made within a client LIBMI application unless you explicitly override them within the application.

User-Defined Database Parameters: The database server obtains values for the database parameters from the database-information descriptor. The database server initializes the database-information descriptor with the system-default database parameters in Table 7-5. You can initialize your own database-information descriptor to override the default database parameters. When you override system-default database parameters, you enable your application to have database information that is independent of the client/server environment in which it runs.

To override the system-default database parameters:

1. Allocate a database-information descriptor.
2. Fill the fields of the database-information descriptor with the default database parameters you need.

   If you do not set a particular field, the database server uses the system-default value in Table 7-5 for the associated database parameter.
3. Pass a pointer to this database-information descriptor to the **mi_set_default_database_info( )** function.

The user-defined database parameters provide database information for all connections made within a client LIBMI application *after* these functions execute (unless the functions are called again to set new default values).

You can obtain existing default database parameters with the **mi_get_default_database_info( )** function. This function populates a user-defined database-information descriptor with the current default database parameters.

─────────────────────────── **Server Only** ───────────────────────────

In a C UDR, **mi_get_default_database_info( )** obtains the same information as **mi_get_database_info( )**. The **mi_set_default_database_info( )** function is ignored.

─────────────────────────── **End of Server Only** ───────────────────────────

Obtaining Current Database Parameters

To obtain database parameters associated with an open connection, use the **mi_get_database_info( )** function. This function populates a user-defined database-information descriptor with values from the specified open connection.

─────────────────────────── **Server Only** ───────────────────────────

The **mi_get_database_info( )** function is valid with a C UDR.

─────────────────────────── **End of Server Only** ───────────────────────────

Using Session Parameters

The *parameter-information descriptor*, MI_PARAMETER_INFO, allows you to set the following session parameters for the client LIBMI application:

- Disables invocation of callbacks
- Enables checking of pointers

Unlike most DataBlade API structures, the parameter-information descriptor is *not* an opaque C data structure. To access session-parameter information, you must directly access the fields of a parameter-information descriptor that you allocate. Table 7-6 shows the fields in the MI_PARAMETER_INFO structure.

Table 7-6. Fields in the Parameter-Information Descriptor

Field	Data Type	Description
callbacks_enabled	**mi_integer**	Indicates whether callbacks are enabled:
		• A value of 1 indicates that callbacks are *enabled*.
		• A value of 0 indicates that callbacks are *disabled*.

Table 7-6. Fields in the Parameter-Information Descriptor (continued)

Field	Data Type	Description
pointer_checks_enabled	**mi_integer**	Indicates whether pointers (such as MI_ROW pointers) that the client LIBMI application passes to the database server are checked to ensure that they are within the heap space of the process:
		• A value of 1 indicates that pointers are checked.
		• A value of 0 indicates that pointers are *not* checked.

The **milib.h** header file defines the MI_PARAMETER_INFO structure.

Before you establish a connection, you can determine which of the following session parameters to use:
• The system-default session parameters
• Default session parameters that you specify

Using System-Default Session Parameters

When the database server establishes a connection, it uses the values in Table 7-7 as the system-default session parameters.

Table 7-7. System-Default Session Parameters

System-Default Session Parameter	System-Default Value
Callbacks Enabled?	Yes
Pointers Checked?	Yes

The system-default session parameters provide session-parameter information for *all* connections made within a client LIBMI application unless you explicitly override them within the application.

Using User-Defined Session Parameters

The database server obtains values for the session parameters from the parameter-information descriptor. The database server initializes the parameter-information descriptor with the system-default session parameters in page 7-9. To override these system-default values, you can initialize your own parameter-information descriptor to set session parameters.

The following DataBlade API functions access default session parameters for a client LIBMI application.

DataBlade API Function	Purpose
mi_set_parameter_info( )	Sets session parameters for the current session
mi_get_parameter_info( )	Obtains session parameters for the current session

To override the system-default session parameters:
1. Allocate a parameter-information descriptor.
2. Fill the fields of the parameter-information descriptor with the default session parameters you need.

If you do not set a particular field, the database server uses the system-default value in Table 7-7 on page 7-9 for the associated session parameter.

3. Pass a pointer to this parameter-information descriptor to the **mi_set_parameter_info( )** function.

You can examine existing session parameters with the **mi_get_parameter_info( )** function. This function populates a user-defined parameter-information descriptor with the current session parameters.

Setting Connection Parameters for a Client Connection

The following example shows one way to set default connection parameters. Assume that the system-default connection parameters are as follows.

System-Default Parameter	Parameter Value
Default database server	**joe**
	(**INFORMIXSERVER** environment variable is set to **joe**.)
Default user	**tester**
Default user password	No password
Callbacks enabled?	Yes (system default)
Pointers checked?	Yes (system default)

The following code fragment uses DataBlade API functions to change the following default system values.

DataBlade API Function	Default Parameter	Default Value
mi_set_default_connection_info( )	Database server name	**beth**
	Server port = 0	None
mi_set_default_database_info( )	Database name	**template1**
	User-account name	**miadmin**
	User-account password	No password

```
extern void MI_PROC_CALLBACK all_callback( );
MI_CONNECTION                   *conn;
MI_CONNECTION_INFO               conn_info;
MI_DATABASE_INFO                db_info;

/* Initialize DataBlade API */
mi_register_callback(conn, MI_Exception, all_callback,
   NULL, NULL);

/* Assign default connection parameter in the
 * connection-information descriptor
 */
conn_info.server_name = "beth";
conn_info.server_port = 0;

/* Set default connection parameters for the application */
if ( mi_set_default_connection_info(&conn_info) == MI_ERROR )
     printf("FAILED: mi_set_default_connection_info( )\n");

/* Assign default database parameters in the
 * database-information descriptor
 */
db_info.user_name = "miadmin";
db_info.database_name = "template1";
```

```
db_info.password = NULL;

/* Set default database parameters for the application */
if ( mi_set_default_database_info(&db_info) == MI_ERROR )
     printf("FAILED: mi_set_default_database_info( )\n");

/* Get default connection and database parameters for
 * application
 */
mi_get_default_connection_info(&conn_info);
mi_get_default_database_info(&db_info);

/* Make sure the right database server is set as the default */
if ( strcmp("beth", conn_info.server_name) != 0 )
   printf("FAILED: got server_name %s, should be beth\n",
      conn_info.server_name);

/* Connect to database server 'beth' */
conn = mi_server_connect(&conn_info);
if ( conn == NULL )
   printf("FAILED: CONNECT to beth\n");
else
   {
   printf("OK: connected to %s\n", conn_info.server_name);
}
```

After these new defaults are established, the application calls **mi_server_connect( )**
to request a connection to the **beth** database server. If this request is successful, the
application opens the **template1**. For more information on **mi_server_connect( )**,
see "Connections with mi_server_connect( )" on page 7-16.

Establishing a Connection

The following DataBlade API functions are constructor functions for a connection
descriptor:

- The **mi_open( )** function

───────────────── **Client Only** ─────────────────

- The **mi_server_connect( )** function

───────────────── **End of Client Only** ─────────────────

These functions establish a connection and return a pointer to a connection
descriptor, which holds information from the session context. You can then pass
this connection descriptor to subsequent DataBlade API functions that need to
access the session context.

The DataBlade API supports the establishment of two kinds of connections:
- UDR connection
- Client connection

Establishing a UDR Connection (Server)

A *UDR connection* is the way that a C UDR obtains access to the session context;
that is, to the database server and database that the calling client application has
already established. For a summary of restrictions that the UDR imposes on a
session, see "Session Restrictions" on page 12-6.

A C UDR can establish one of two kinds of connections to a session:

- The public connection descriptor provides the C UDR invocations within an SQL command with access to the session context.
- The session-duration connection descriptor provides the C UDR invocations within a session with access to the session context.

Obtaining a Connection Descriptor

A *public connection descriptor* (usually just called a *connection descriptor*) provides a local copy of session information for the use of the UDR. Because it has a PER_STMT_EXEC memory duration, *all* UDR invocations in the same SQL statement can share the session-context information (see Table 7-1 on page 7-3). The following table summarizes the memory operations for a connection descriptor in a C UDR.

Memory Duration	Memory Operation	Function Name
PER_STMT_EXEC	Constructor	**mi_open( )**
	Destructor	**mi_close( )**

To establish a UDR connection, pass all three arguments of **mi_open( )** as NULL-valued pointers. The following code fragment uses **mi_open( )** to establish a connection for a UDR:

```
mi_integer func1( )
{
   MI_CONNECTION *conn;

/* Open a connection from C UDR to database server
 * of current session context:
 *    database = currently open database
 *    user = operating-system user account which is running
 *             the SQL statement that called this
 *             user-defined routine
 *    password = default specified for this user
 */
   conn = mi_open(NULL, NULL, NULL);

/* If connection descriptor is NULL, there was an error
 * connecting to the session context.
 */
   if ( conn == NULL )
      {
      mi_db_error_raise(conn, MI_EXCEPTION,
         "func1: cannot establish connection", NULL);
      }

... /* Code for use of this connection goes here */
}
```

Important: When called within a C UDR, many DataBlade API functions do not use the connection descriptor. You can pass a NULL-valued pointer as a connection descriptor to the DataBlade API functions for smart large objects, which have the **mi_lo_** prefix. The *IBM Informix DataBlade API Function Reference* describes these functions. Exceptions to this rule are listed in the documentation. Instead, pass in the connection descriptor that the **mi_open( )** function obtains.

The **mi_open( )** call can be expensive in a C UDR. If the UDR instance contains many invocations, you can obtain the connection descriptor the *first* time the UDR

is invoked and store it as part of the **MI_FPARAM** state information, as Figure 10-6 on page 10-29 shows. For more information, see "Saving a User State" on page 9-8.

Tip: It is not valid for a UDR to cache a connection descriptor at a memory duration higher than PER_COMMAND. If you need session-context information with a higher duration, use a session-duration connection descriptor. For more information, see "Obtaining a Session-Duration Connection Descriptor" on page 7-13.

Obtaining a Session-Duration Connection Descriptor

A *session-duration connection descriptor* provides a public copy of connection information, providing access to the actual session information of the client application. Because this connection descriptor has a PER_SESSION memory duration, all UDR invocations in the session can share the session-context information (see Table 7-1 on page 7-3). (For more information on a session, see "PER_SESSION Memory Duration" on page 14-15.)

The following table summarizes the memory operations for a session-duration connection descriptor in a C UDR.

Memory Duration	Memory Operation	Initiator of Operation
PER_SESSION	Constructor	**mi_get_session_connection( )**
	Destructor	End of session

Warning: The advanced **mi_get_session_connection( )** function can adversely affect your UDR if you use it incorrectly. Use it only when a regular function cannot perform the task you need done.

The **mi_get_session_connection( )** function is *not* a true constructor, in the sense that it does not actually allocate a connection descriptor in a PER_SESSION duration. Instead, it returns a handle to the actual session connection, which has a PER_SESSION duration. Therefore, the **mi_get_session_connection( )** is often faster than **mi_open( )** (which does allocate a connection descriptor in PER_COMMAND memory).

The **minmprot.h** header file defines the restricted-access **mi_get_session_connection( )** function. The **minmmem.h** header file automatically includes the **minmprot.h** header file. However, the **mi.h** header file does *not* automatically includes **minmmem.h**. To use **mi_get_session_connection( )**, you must include **minmmem.h** in any DataBlade API routine that calls these functions.

A session-duration connection descriptor is useful in the following cases:

- As an alternative to frequent calls to **mi_open( )**

   The **mi_open( )** function is a relatively expensive call. If you need to open connections frequently in your UDR, **mi_get_session_connection( )** is the preferred alternative. With a session-duration connection descriptor, the database server caches a connection for you.

- To obtain access to session-duration function descriptors

   One of the DataBlade API data type structures that the connection descriptor holds is a function descriptor. When you pass a Fastpath look-up function (see Table 9-5 on page 9-18) a public connection descriptor, the function descriptor that these functions allocate is valid until the SQL command completes. If you pass these look-up functions a session-duration connection descriptor instead of

a public connection descriptor, you can obtain a session-duration function descriptor, which is valid until the session ends. In this way, other UDRs can use Fastpath to execute the same UDR without having to create and destroy its function descriptor for each execution. For more information, see "Reusing a Function Descriptor" on page 9-30.

Keep the following restrictions in mind when you decide to use a session-duration connection:

- Do *not* use **mi_close( )** to free a session-duration connection descriptor.

  A session-duration connection descriptor has the duration of the session. An attempt to free a session-duration connection with **mi_close( )** generates an error.

- Do *not* cache a session-duration connection descriptor in the user state of an **MI_FPARAM** structure.

  You must obtain a session-duration connection descriptor in *each* UDR that uses it.

- Do *not* call **mi_get_session_connection( )** in a parallelizable UDR.

  If the UDR must be parallelizable, use **mi_open( )** to obtain a connection descriptor.

Establishing a Client Connection

A client LIBMI application can establish a client connection in either of the following ways:

- The **mi_open( )** function
- The **mi_server_connect( )** function

These DataBlade API functions obtain a connection descriptor for the client connection. The following table summarizes the memory operations for a connection descriptor in a client LIBMI application.

Memory Duration	Memory Operation	Function Name
For the duration of the session	Constructor	**mi_open( )**, **mi_server_connect( )**
	Destructor	**mi_close( )**

Important: When called within a client LIBMI application, DataBlade API functions *always* use the connection descriptor. Therefore, *never* send in a NULL-valued pointer as a connection descriptor to DataBlade API functions. Instead, pass in the connection descriptor that the **mi_open( )**, **mi_server_connect( )**, or **mi_server_reconnect( )** function obtains.

After the client LIBMI application has established a connection, the session begins.

Connections with mi_open( )

The **mi_open( )** function establishes a *default connection* for the calling DataBlade API module and returns a connection descriptor. A default connection is a connection to the default database server (which the **INFORMIXSERVER** environment variable specifies) and a specified database.

To establish a default connection, the **mi_open( )** function accepts the following information as arguments.

mi_open( ) Argument	Purpose	Default Used When Argument is NULL
Database name	The name of the database to open	None
User account name	The name of the login account for the user who is to open the database This account must be valid on the server computer.	The name of the system-defined user account (See Table 7-5 on page 7-7.)
Account password	The password of the login account for the user who is to open the database This account must be valid on the server computer.	The password of the system-defined user account (See Table 7-5 on page 7-7.)

All of these arguments are passed as pointers to character strings. You can specify NULL for any of these arguments, in which case **mi_open( )** uses the specified default values. If the client LIBMI application uses a shared-memory communication, it can only establish one connection per application.

The following code fragment demonstrates the simplest way for a client LIBMI application to initiate a connection to the default database server and to open a database:

```
/*
 * Use mi_open( ) to connect to the database passed on the
 * client application command line. Close the connection with
 * mi_close( ).
 */

#include <mi.h>
#include <stdio.h>

main( mi_integer argc, char *argv[] )
{
  MI_CONNECTION  *conn;

/* Check incomming parameters from command line */
   if ( argc != 2 )
      {
      printf(stderr, "Usage:%s <db name>\n", argv[0]);
      exit(2);
      }

/* Open a connection from client LIBMI application to
 * database server.
 *          database = parameter on command line
 *          user = operating-system user account which is
 *                   running this application
 *          password = default specified for this user
 */
   conn = mi_open(argv[1], NULL, NULL);

/* If connection descriptor is NULL, there was an error
 * attempting to connect to the database server and database
 * specified.  Exit application.
 */
   if ( NULL == conn )
      {
      fprintf(stderr, "Cannot open database: %s\n",
        argv[1]);
```

```
        exit(3);
        }

    /* Code for application use of this connection goes here */
    ...

    /* Valid connection has occurred. Close the connection
     * and exit the application.
     */
      mi_close(conn);
      exit(0);
}
```

In this example, the name of the database to be opened is passed on the command line. The *user_name* and the *user_password* arguments to **mi_open( )** are both passed as NULL, which indicates that **mi_open( )** uses the default user and password.

Connections with mi_server_connect( )

To exercise more control over which connection to establish, a client LIBMI application can use **mi_server_connect( )**, which establishes a connection to a specified database server. The **mi_server_connect( )** function obtains information about which database server to connect to from a connection-information descriptor. This function does *not* open a database.

This DataBlade API function provides greater flexibility for client LIBMI applications that run against different database servers. You can pass information about the connection through descriptors.

Associating User Data with a Connection

The connection descriptor provides information about various data type structures associated with the current connection. (For a list of this information, see Table 7-1 on page 7-3.) In addition, you can store the address of private information, called *user data*, in the connection descriptor. The connection descriptor can hold this *user-data pointer*, which points to the private user-data information.

┌─────────────────── **Server Only** ───────────────────┐

You allocate the user data with a DataBlade API memory-management function from the shared memory of the database server. The memory duration of this user data must correspond with the connection descriptor that holds the user-data pointer, as the following table shows.

Type of Connection Descriptor	Memory Duration of User Data	Which UDRs Can Access User Data
Public connection descriptor (with **mi_open( )**)	MI_COMMAND	All UDR invocations in the same SQL command have access to the connection descriptor that **mi_open( )** returns.
Session-duration connection descriptor (with **mi_get_session_connection( )**)	MI_SESSION	All C UDR invocations in the session have access to the connection descriptor that **mi_get_session_connection( )** returns.

Therefore, your user data is available to all UDRs that can access its connection descriptor.

Important: A session-duration connection descriptor is a restricted feature that can adversely affect your UDR if used incorrectly. Use it only when a public connection descriptor will not support the task you need to perform. For more information, see "Obtaining a Session-Duration Connection Descriptor" on page 7-13.

───────────────── **End of Server Only** ─────────────────

───────────────── **Client Only** ─────────────────

The user data is allocated in client-side memory. Therefore, your user data is available to *all* DataBlade API functions that execute in the session.

───────────────── **End of Client Only** ─────────────────

Table 7-8 shows the functions that the DataBlade API provides to access the user data of a connection descriptor.

Table 7-8. DataBlade API Accessor Functions for User Data in the Connection Descriptor

DataBlade API Accessor Function	User-State Information
mi_get_connection_user_data( )	Obtains the user-data pointer from the connection descriptor
mi_set_connection_user_data( )	Sets the user-data pointer in the connection descriptor

The size of the connection user data is the size of a pointer of type "**void ***". The DataBlade API does not interpret or touch the associated user-data address, other than to store and retrieve it from the connection descriptor.

Initializing the DataBlade API

Before you can use the DataBlade API to communicate with the database server, you must make sure that it is initialized. When you establish a connection, the DataBlade API function automatically initializes the DataBlade API. However, if your DataBlade API module does *not* establish a connection, it must still ensure that it initializes the DataBlade API.

Important: If the DataBlade API was not initialized, calls to subsequent DataBlade API functions generate errors.

Table 7-9 lists the functions that can initialize the DataBlade API.

Table 7-9. DataBlade API Functions That Initialize the DataBlade API

DataBlade API Initialization Function	Valid in Client LIBMI Application?	Valid in User-Defined Routine?
mi_client_locale( )	Yes	Yes
mi_get_default_connection_info( )	Yes	Yes
mi_get_default_database_info( )	Yes	Yes
mi_get_next_sysname( )	Yes	No
mi_get_parameter_info( )	Yes	Yes
mi_init_library( )	Yes	No

Table 7-9. DataBlade API Functions That Initialize the DataBlade API (continued)

DataBlade API Initialization Function	Valid in Client LIBMI Application?	Valid in User-Defined Routine?
mi_open( )	Yes	Yes
mi_register_callback( )	Yes	Yes
mi_server_connect( )	Yes	No
mi_set_default_connection_info( )	Yes	Ignored
mi_set_default_database_info( )	Yes	Ignored
mi_set_parameter_info( )	Yes	No
mi_sysname( )	Yes	Yes

One of the functions listed in Table 7-9 must be the first DataBlade API function called in a DataBlade API session. If you do *not* call one of these functions, the DataBlade API is not initialized and all subsequent DataBlade API calls return error status.

Closing a Connection

To close a connection, free the associated connection descriptor. When the connection descriptor is freed, the DataBlade API also frees the session-context resources, including the following:

- Save sets
- Prepared statements (explicit statement descriptors)
- For an SQL statement executed with **mi_exec( )** (also called the current statement):
    - The implicit statement descriptor for the current statement
    - The row structure and associated row descriptor for the current statement
- Cursors (implicit and explicit)
- Function descriptors
- Callbacks registered for the connection
- Connection user data

To conserve resources, use **mi_close( )** to deallocate the connection descriptor explicitly once your DataBlade API module no longer needs it. The **mi_close( )** function is the destructor function for a connection descriptor. It frees the connection descriptor and any resources that are associated with it.

Server Only

In a C UDR, a public connection descriptor has a memory duration of PER_STMT_EXEC. Therefore, a connection descriptor remains active until one of the following events occurs:

- The **mi_close( )** function closes the specified UDR connection.
- The current SQL statement completes execution.

When a UDR connection is closed, the UDR can no longer access the associated connection information (see Table 7-1 on page 7-3). However, the session remains open until the client application ends it. Therefore, a UDR can obtain a new UDR connection with another call to **mi_open( )**.

Tip: After a C UDR closes a connection, the UDR can no longer access the connection resources in Table 7-1 on page 7-3. Any open smart large objects and operating-system files, however, remain valid for the duration of the session. You can explicitly close these descriptors with the **mi_lo_close( )** and **mi_file_close( )** functions, respectively.

A session-duration connection descriptor has a memory duration of PER_SESSION. Therefore, it and its associated connection information remain valid until the end of the session. However, a session-duration connection is a *restricted* feature of the DataBlade API. Use it only when a public connection descriptor will not perform the task you need. For more information, see "Obtaining a Session-Duration Connection Descriptor" on page 7-13.

───────────────── **End of Server Only** ─────────────────

───────────────── **Client Only** ─────────────────

In a client LIBMI application, a connection descriptor has a scope of the session. When the client connection closes, the session ends. Therefore, a connection descriptor remains active until one of the following events occurs:

- The **mi_close( )** function closes the specified connection, ending the session.
- The client LIBMI application completes.

Tip: Once a client LIBMI application closes a connection, it can no longer access the connection information. In addition, any open smart large objects and files are closed.

───────────────── **End of Client Only** ─────────────────

Chapter 8. Executing SQL Statements

In This Chapter

One basic task of a DataBlade API module is to send SQL statements to the database server for execution. To execute an SQL statement, a DataBlade API module must perform the following tasks:

- Assemble the SQL statement and send it to the database server for execution
- Process results that the database server returns to the module
- If the SQL statement (such as a SELECT) returns rows, obtain each row of data
- For each row, obtain the column value or values of interest
- Complete the execution of the statement

This chapter describes each of these execution steps in detail.

Executing SQL Statements

To execute an SQL statement, a DataBlade API module must send the SQL statement to the database server, where the statement is actually executed. The DataBlade API provides the following statement-execution functions for use in a DataBlade API module:

- **mi_exec( )**
- **mi_exec_prepared_statement( )**
- **mi_open_prepared_statement( )**

All of these functions perform the same basic task: they send a string representation of an SQL statement to the database server, which executes it and returns statement results. The **mi_exec( )** function is the simplest way to execute an SQL statement.

───────────────────── **Server Only** ─────────────────────

A C user-defined routine (UDR) that executes SQL statements must be registered as a variant function; that is, its CREATE FUNCTION statement must either include the VARIANT routine modifier or omit both the NOT VARIANT and VARIANT routine modifiers (VARIANT is the default).

───────────────────── **End of Server Only** ─────────────────────

This section provides a summary of factors to consider when choosing the DataBlade API statement-execution function to use. It then describes the two methods for statement execution.

Method of Statement Execution	More Information
Parse, optimize, and execute the statement in one step	"Executing Basic SQL Statements" on page 8-6
Parse and optimize the statement to create a *prepared statement* Execute the prepared statement	"Executing Prepared SQL Statements" on page 8-11

Tip: Before you use a DataBlade API function that sends an SQL statement to the database server, make sure you obtain a valid connection descriptor.

Choosing a DataBlade API Function

Table 8-1 shows the functions that the DataBlade API provides to send SQL statements to the database server for execution.

Table 8-1. Statement-Execution Functions of the DataBlade API

DataBlade API Function	Type of Statement	When to Use Function	
		Statement Executed Many Times or Contains Input Parameters?	Query Can Use Implicit Cursor?
mi_exec( )	Query Other valid SQL statements	No	Yes
mi_exec_prepared_statement( )	Query Other valid SQL statements	Yes	Yes
mi_open_prepared_statement( )	Query only	Yes	No

As the preceding table shows, you need to consider the following factors when deciding which DataBlade API statement-execution function to use:

- What type of SQL statement do you need to send?
- Does your SQL statement contain input parameters?
- If the SQL statement is a query, can you use an implicit cursor to access the retrieved rows?

Choose the DataBlade API statement-execution function that is appropriate for the needs of your DataBlade API application.

Type of Statement

The DataBlade API statement-execution functions can execute the following types of SQL statements:

- An SQL statement that does *not* return rows of data (is *not* a SELECT statement and not an EXECUTE FUNCTION statement that executes an iterator function)

  Most SQL statements do *not* return rows. For example, all data definition (DDL) statements and most data manipulation (DML) statements return only a status to indicate the statement's success.

- An SQL statement that does return one or more rows of data

  The following SQL statements return rows:

  – SELECT statement
  – EXECUTE FUNCTION statement, when the user-defined function returns more than one row of data

  An SQL statement that returns rows is often called a *query* because it asks the database server to answer a question: which rows match?

Tip: The term "query" is sometimes used to refer to any SQL statement. However, this publication uses the more specific definition of "query": an SQL statement that returns rows.

The following table shows how to choose a DataBlade API statement-execution function based on the type of SQL statement.

Type of Statement	DataBlade API Function
Query, Other valid statements	**mi_exec( )**, **mi_exec_prepared_statement( )**

Type of Statement	DataBlade API Function
Query only	**mi_open_prepared_statement( )**

Prepared Statements and Input Parameters

A *prepared SQL statement* is the parsed version of an SQL statement. The database server *prepares* an SQL statement for execution at a later time. Preparing a statement enables you to separate the parsing and execution phases of the statement execution. When you prepare a statement, you send the statement to the database server to be parsed. The database server checks the statement for syntax errors and creates an optimized version of the statement for execution.

You need to prepare an SQL statement only once. You can then execute the statement multiple times. Each time you execute the statement, you avoid the parsing phase. Prepared statements are useful for SQL statements that execute often in your DataBlade API module.

SQL statements that have missing column or expression values are called *parameterized statements* because you use *input parameters* as placeholders for missing column or expression values. An input parameter is a placeholder in an SQL statement that indicates that the actual column value is provided at runtime. You can specify input parameters in the statement text representation of an SQL statement for either of the following reasons:

- A column or expression value is unknown at the time you prepare the SQL statement.
- A column or expression value changes for each execution of the SQL statement.

For a parameterized SQL statement, your DataBlade API module must provide the following information to the database server for each of its input parameters.

Input-Parameter Information	More Information
Specify the input parameter in the text of the SQL statement	"Assembling a Prepared Statement" on page 8-11
Specify the value for the input parameter when the statement executes	"Assigning Values to Input Parameters" on page 8-27

You can also obtain information about the input parameters after the parameterized statement is prepared. For more information, see "Obtaining Input-Parameter Information" on page 8-15.

A DataBlade API module can prepare an SQL statement for the following reasons:

- To increase performance by reducing the number of times that the database server parses and optimizes the statement
- To execute a parameterized SQL statement and provide different input-parameter values each time the statement executes

The following table shows how to choose a DataBlade API statement-execution function based on whether the SQL statement needs to be prepared.

Statement Needs To Be Prepared?	DataBlade API Function
No	**mi_exec( )**
Yes	**mi_exec_prepared_statement( )**, **mi_open_prepared_statement( )**

The **mi_exec_prepared_statement( )** or **mi_open_prepared_statement( )** function provides argument values for specifying the input-parameter values when the function executes the statement. You can also use these functions to execute prepared statements that do not have input parameters.

Queries and Implicit Cursors

When a DataBlade API statement-execution function executes a query, the function must create a place to hold the resulting rows. Each of these functions (**mi_exec( )**, **mi_exec_prepared_statement( )**, or **mi_open_prepared_statement( )**) automatically creates a *row cursor* (often called simply a *cursor*). The row cursor is an area of memory that serves as a holding place for rows that the database server has retrieved.

The simplest way to hold the rows of a query is to use an *implicit cursor*, which is defined with the following characteristics.

Cursor Characteristic	Restriction
Read-only	You can only examine the contents of the row cursor. You cannot modify these contents.
Sequential	A sequential cursor allows movement through the rows of the cursor in the forward direction only. You cannot go backward through the cursor. To reaccess a row that you have already accessed, you must close the cursor, reopen it, and move to the desired row.

Most DataBlade API modules can use an implicit cursor for accessing rows. However, if the cursor characteristics of the implicit cursor are not adequate for the needs of your DataBlade API module, you can define an *explicit cursor* with any of the following cursor characteristics.

Cursor Characteristic	Description
Cursor type	In which direction does the cursor enable you to access rows? You can choose a sequential cursor or a scroll cursor.
Cursor mode	Which operations are valid on the rows in the cursor? You can choose read-only or update mode.
Cursor lifespan	How long does the cursor remain open? You can choose whether to use a hold cursor.

For more information on these cursor characteristics, see "Defining an Explicit Cursor" on page 8-22.

The following table shows how to choose a DataBlade API statement-execution function based on the type of cursor that the query requires.

Can Query Use Implicit Cursor?	DataBlade API Function
Yes	**mi_exec( )**, **mi_exec_prepared_statement( )**
No	**mi_open_prepared_statement( )**

With the **mi_open_prepared_statement( )** function, you can specify an explicit cursor to hold the query rows. In addition, you can assign a name to the cursor that you can use in other SQL statements.

Executing Basic SQL Statements

The **mi_exec( )** function provides the simplest way to send a basic SQL statement to the database server for execution. A basic SQL statement is one that does not need to be prepared. That is, the statement does *not* execute many times in the DataBlade API module or it does not contain input parameters. To send a basic SQL statement to the database server for execution, take the following steps:

- Assemble a statement string, which contains the SQL statement to execute.
- Send the statement string to the database server with **mi_exec( )**.

The database server parses the statement string, optimizes it, executes it, and sends back the statement results.

Assembling a Statement String

The **mi_exec( )** function passes the SQL statement to the database server as a *statement string*, which is a text representation of the SQL statement. To execute a statement with **mi_exec( )**, the statement string must include the entire SQL statement; that is, it *cannot* contain any input parameters.

You can assemble this statement string in the following ways:

- If you know all the information at compile time, assemble the statement as a fixed string.

  If you know the whole statement structure, you can specify the string itself as the argument to **mi_exec( )**, as the following line shows:

```
mi_exec(conn,
   "select company from customer where \
    customer_num = 101;", MI_QUERY_BINARY);
```

- If you do *not* know all the information about the statement at compile time, you can use the following features to assemble the statement string:
  - Character variables can hold the identifiers in the SQL statement (column names or table names) or parts of the statement like the WHERE clause. They can also contain keywords of the statement.

    You can then build the SQL statement as a series of string operations, as Figure 8-1 shows.

```
mi_string stmt_txt[30];
mi_string fld_name[15];
...
stcopy("select ", stmt_txt);
fld_name = obtain_fldname(...);
stcat(fld_name, stmt_txt);
stcat("from customer where customer_num = 101", stmt_txt);
...
mi_exec(conn, stmt_txt, MI_QUERY_BINARY);
```

Figure 8-1. Assembling a SELECT Statement from a Character String

- If you know what column values the statement specifies, you can declare program variables to provide column values that are needed in a WHERE clause or to hold column values that database server returns.

Figure 8-2 shows the SELECT statement of Figure 8-1 changed so that it uses a variable to determine the customer number dynamically.

```
mi_string stmt_txt[30];
mi_integer cust_num;
...
stcopy("select company from customer where customer_num = ",
   stmt_txt);
cust_num = obtain_custnum(...);
stcat(cust_num, stmt_txt);
...
stmt_desc = mi_exec(conn, stmt_txt, MI_QUERY_BINARY);
```

Figure 8-2. Using a Variable to Assemble a SELECT Statement

The statement string can contain multiple SQL statements. Each SQL statement must be terminated with the semicolon (;) symbol. For more information, see "Executing Multiple SQL Statements" on page 8-32.

Sending an SQL Statement

The **mi_exec( )** function is for the execution of basic SQL statements, both queries and other valid SQL statements. In a DataBlade API module, use the following **DataBlade API** functions to execute a basic SQL statement.

Step in Execution of Basic SQL Statement	DataBlade API Function
Send the basic SQL statement to the database server for execution and open any cursor required	**mi_exec( )**
Release statement resources	**mi_query_finish( )**, **mi_query_interrupt( )**

Once the database server executes the statement that **mi_exec( )** sends, the statement becomes the *current statement*. The current statement is the most recent SQL statement on the connection. Only one statement per connection is current. The database server sends back the results of the current statement, including whether the current statement was successful.

The **mi_exec( )** function creates an *implicit statement descriptor* to hold the information about the current statement. The following table summarizes the memory operations for an implicit statement descriptor.

Memory Duration	Memory Operation	Function Name
Not allocated from memory-duration pools	Constructor	**mi_exec( )**
	Destructor	**mi_query_finish( )**

Table 8-2 lists the DataBlade API accessor functions for the implicit statement descriptor that **mi_exec( )** creates.

Table 8-2. Accessor Functions for an Implicit Statement Descriptor

Statement-Descriptor Information	DataBlade API Accessor Function
The name of the SQL statement that is the current statement	**mi_result_command_name( )**

Table 8-2. Accessor Functions for an Implicit Statement Descriptor (continued)

Statement-Descriptor Information	DataBlade API Accessor Function
A row descriptor for the columns in the current statement	**mi_get_row_desc_without_row( )** From the row descriptor, you can use the row-descriptor accessor functions to obtain information about a particular column (see Table 5-3 on page 5-30).

You obtain the status of the current statement with the **mi_get_result( )** function. For more information, see "Processing Statement Results" on page 8-33.

Tip: The return value that the **mi_exec( )** function returns does not indicate the success of the current statement. It indicates if **mi_exec( )** was able to successfully send the statement to the database server.

When **mi_exec( )** executes a query, it performs the following additional steps:
1. Opens an implicit cursor to hold the query rows
2. Reads the query rows into the open cursor

The Implicit Row Cursor: When **mi_exec( )** executes a query, it automatically opens an *implicit cursor* to hold the resulting rows. This cursor is associated with the current statement and is stored as part of the connection descriptor. Therefore, only one cursor per connection can be current. For more information, see "Queries and Implicit Cursors" on page 8-5.

Tip: If the implicit cursor that **mi_exec( )** creates does not adequately meet the needs of your DataBlade API module, you can use the **mi_open_prepared_statement( )** function to define other types of cursors. For more information, see "Defining an Explicit Cursor" on page 8-22.

When the **mi_exec( )** function successfully fetches the query results into the cursor, the *cursor position* points to the *first* row of the cursor, and the **mi_get_result( )** function returns a status of MI_ROWS to indicate that the cursor contains rows.

You can access these rows one at a time with the **mi_next_row( )** function. Each access obtains the row to which the cursor position points. After each access to the cursor, the cursor position moves to the next row. For more information, see "Retrieving Query Data" on page 8-39.

Control Modes for Query Data: The data that the database server returns for a query can be in one of two *control modes*:
- In text representation, the query data is represented as null-terminated strings. Data in its text representation is often called a *literal value*.
- In binary representation, the query data is represented in its internal format; that is, in the format that the database server uses to store the value.

Table 8-3 shows the format of different data types in the two control modes.

Table 8-3. Control Modes for Data

Type of Data	Text Representation	Binary Representation
Character	Null-terminated string	Varying-length structure: **mi_lvarchar**

Table 8-3. Control Modes for Data (continued)

Type of Data	Text Representation	Binary Representation
Date	"mm/dd/yyyy" Nondefault locale: End-user date format	Integer number of days since December 31, 1899 (DATE, **mi_date**)
Date/time	"yyyy-mm-dd HH:MM:SS" Nondefault locale: End-user date and time format	**dtime_t** (DATETIME, **mi_datetime**)
Interval	"yyyy-mm" "dd HH:MM:SS" Nondefault locale: End-user date and time format	**intrvl_t** (INTERVAL, **mi_interval**)
Integer	Integer value as a string: thousands separator = "," Nondefault locale: End-user numeric format	Internal format: • Two-byte integer (SMALLINT, **mi_smallint**) • Four-byte integer (INTEGER, **mi_integer**) • Eight-byte integer: **ifx_int8_t** (INT8, **mi_int8**)
Decimal	Fixed-point value as a string: thousands separator = "," decimal separator = "." Nondefault locale: End-user numeric format	**dec_t** (DECIMAL, **mi_decimal**)
Monetary	Fixed-point value as a string: thousands separator = "," decimal separator = "." currency symbol = "$" Nondefault locale: End-user monetary format	**dec_t** (MONEY, **mi_money**)
Floating-point	Floating-point value as a string: thousands separator = "," decimal separator = "." Nondefault locale: End-user numeric format	Internal format: • single-precision floating point (SMALLFLOAT, **mi_real**) • double-precision floating point (FLOAT, **mi_double_precision**)
Boolean	"t" or "T" "f" or "F"	MI_TRUE, MI_FALSE (BOOLEAN, **mi_boolean**)
Smart large object	Text representation of the LO handle (obtained with **mi_lo_to_string( )**)	LO handle (CLOB, BLOB; **MI_LO_HANDLE**)
Row type	Unnamed row type: "ROW(*fld_value1*, *fld_value2*, ...)" Named row type: "*row_type*(*fld_value1*, *fld_value2*, ...)"	Row structure (ROW, named row type; **MI_ROW**)

Table 8-3. Control Modes for Data (continued)

Type of Data	Text Representation	Binary Representation
Collection type	"SET{*elmnt_value, elmnt_value, ...*}" "MULTISET{*elmnt_value, elmnt_value, ...*}" "LIST{*elmnt_value, elmnt_value, ...*}"	Collection structure (SET, LIST, MULTISET; **MI_COLLECTION**)
Varying-length opaque type	External format of opaque type (as returned by output support function)	Varying-length structure: **mi_bitvarying** (which contains the internal C data type)
Fixed-length opaque type	External format of opaque type (as returned by output support function)	Internal C data type
Distinct type	Text representation of its source data type	Binary representation of its source data type

The **mi_exec( )** function indicates the control mode of the query with a bit-mask *control* argument, which is one of the following flags.

Control Mode	Control-Flag Value
Text representation	MI_QUERY_NORMAL
Binary representation	MI_QUERY_BINARY

In the **send_statement( )** function (page 8-10), **mi_exec( )** sets the control mode of the query data to text representation.

To determine the control mode for query data, use the **mi_binary_query( )** function. The **mi_binary_query( )** function determines the control mode for data of the current statement.

Example: The send_statement( ) Function: The **send_statement( )** function takes an existing open connection and an SQL statement string as arguments and sends the statement to the database server with the **mi_exec( )** function. It specifies text representation for the query results.

```
/* FUNCTION: send_statement( )
 * PURPOSE: To send an SQL statement to the database server for
 *          execution
 *
 * CALLED BY: Called from within a C user-defined function to
 *            execute a basic SQL statement
 */

mi_integer
send_statement(MI_CONNECTION *conn, mi_string *stmt)
{
   mi_integer count;

   /* Send the statement, specifying results be sent
    * in their text representation (MI_QUERY_NORMAL)
    */
   if ( MI_ERROR == mi_exec(conn, stmt, MI_QUERY_NORMAL) )
      {
       mi_db_error_raise(conn, MI_EXCEPTION,
         "mi_exec failed\n");
      }

   /* Get the results of the current statement */
   count = get_results(conn);
```

```
    /* Release statement resources */
    if ( mi_query_finish(conn) == MI_ERROR )
        {
        mi_db_error_raise(conn, MI_EXCEPTION,
            "mi_query_finish failed\n");
        }

    return ( count );
}
```

The **send_statement( )** function calls another user function, **get_results( )**, to examine the status of the current statement. For the implementation of the **get_results( )** function, see "Example: The get_results( ) Function" on page 8-39.

Executing Prepared SQL Statements

A *prepared statement* is an SQL statement that is parsed and ready for execution. For these statements, you prepare the statement once and execute it as many times as needed. The DataBlade API provides the following functions to execute a prepared SQL statement.

DataBlade API Function	Step in Prepared-Statement Execution
mi_prepare( )	Prepares a text representation of the SQL statement to execute
mi_statement_command_name( ), **mi_get_statement_row_desc( )**, or input-parameter accessor function (Table 8-5 on page 8-15)	Obtains information about the prepared statement
mi_exec_prepared_statement( ) or **mi_open_prepared_statement( )**	Sends the prepared statement to the database server for execution
mi_drop_prepared_statement( )	Releases prepared-statement resources

Preparing an SQL Statement

To turn a statement string for an SQL statement into a format that the database server can execute, use the **mi_prepare( )** statement. The **mi_prepare( )** function performs the following tasks to create a prepared statement:

- Sends a statement string to the database server for parsing
- Assigns an optional name to the SQL statement
- Returns a pointer to a statement descriptor for the prepared statement

Tip: The **mi_prepare( )** function performs the same basic task for a DataBlade API module as the SQL PREPARE statement does for an IBM Informix ESQL/C application.

Assembling a Prepared Statement: The **mi_prepare( )** function passes the SQL statement to the database server as a statement string. For the **mi_prepare( )** function, a statement string can contain either of the following formats of an SQL statement:

- An unparameterized SQL statement (the same as the **mi_exec( )** function accepts)
- A parameterized SQL statement, which contains input parameters

Assembling Unparameterized Statements: If you know all the statement information before the statement is prepared, you assemble an *unparameterized statement* as the

statement string. Pass the SQL statement as a string (or a variable that contains a string) to the **mi_prepare( )** function. For example, Figure 8-3 prepares an unparameterized SELECT statement that obtains column values from the **customer** table.

```
stmt_desc = mi_prepare(conn,
    "SELECT * FROM customer;", NULL)
```

Figure 8-3. Preparing an Unparameterized Statement

For more information, see "Assembling a Statement String" on page 8-6.

Assembling Parameterized Statements: If some column or expression value is provided when the statement actually executes, you assemble the *parameterized* statement as the statement string. Specify input parameters in the statement text representation of an SQL statement. For a description of an input parameter, see "Prepared Statements and Input Parameters" on page 8-4.

You indicate the presence of an input parameter with a question mark (?) anywhere within a statement where an expression is valid. You cannot list a program-variable name in the text of an SQL statement because the database server knows nothing about variables declared in the DataBlade API module. You cannot use an input parameter to represent an identifier such as a database name, a table name, or a column name.

For example, Figure 8-4 shows an INSERT statement that uses input parameters as placeholders for two column values in the **customer** table.

```
insrt_stdesc = mi_prepare(conn,
    "INSERT INTO customer (customer_num, company) \
    VALUES (?,?);", NULL
```

Figure 8-4. Preparing a Statement That Contains Input Parameters

In Figure 8-4, the first input parameter is defined for the value of the **customer_num** column and the second for the value of the **company** column.

Before the prepared statement executes, your DataBlade API module must assign a value to the input parameter. You pass these input-parameter values as arguments to the **mi_exec_prepared_statement( )** or **mi_open_prepared_statement( )** function. For more information, see "Assigning Values to Input Parameters" on page 8-27.

Assigning an Optional Name: You can obtain access to a prepared statement through its statement descriptor. However, other SQL statements that need to reference the prepared statement cannot use a statement descriptor. Therefore, you can assign an optional string name to a prepared SQL statement. Specify a name as the third argument of the **mi_prepare( )** function.

─────────────── **Server Only** ───────────────

The last argument to **mi_prepare( )** specifies the *cursor name* for the prepared statement. Assigning a cursor name is useful for a statement that includes an update cursor so that an UPDATE or DELETE statement that contains the following clause can reference the cursor in this clause:

```
WHERE CURRENT OF cursor_name
```

You can specify an update cursor in the syntax of the SELECT statement that you prepare, as the following versions of the SELECT statement show:

```
SELECT customer_num, company FROM customer
WHERE customer_num = 104 FOR UPDATE OF company;

SELECT customer_num, company FROM customer
WHERE customer_num = 104;
```

For more information on the FOR UPDATE keywords of SELECT with databases that are ANSI compliant and not ANSI compliant, see "Defining a Cursor Mode" on page 8-22.

─────────────────────── **End of Server Only** ───────────────────────

The following code fragment uses the **mi_prepare( )** statement to assign a name to a cursor and an UPDATE WHERE CURRENT OF statement to update the fifth row in this cursor:

```
/* Prepare the FOR UPDATE statement */
if ( (stmt1 = mi_prepare(conn,
          "select * from tab1 for update;",
          "curs1")) == NULL )
   return MI_ERROR;

/* Open the cursor */
if ( mi_open_prepared_statement(stmt1, MI_BINARY,
      MI_QUERY_BINARY, num_params, values, lengths, nulls,
      types, NULL, 0, NULL) != MI_OK )
   return MI_ERROR;

/* Fetch the 5th row */
if ( mi_fetch_statement(stmt1, MI_CURSOR_NEXT, 0, 5)
      != MI_OK )
   return MI_ERROR;

/* Get values from 5th row */
if ( mi_get_result(conn) != MI_ROWS
      || mi_next_row(conn, &res) == NULL )
   return MI_ERROR;

/* Update 5th row */
if ( mi_exec("update tab1 set int_col = int_col + 2 \
          where current of curs1;", NULL) != MI_OK )
   return MI_ERROR;

/* Clean up */
if ( mi_close_statement(stmt1) != MI_OK )
   return MI_ERROR;
if ( mi_drop_prepared_statement(stmt1) != MI_OK )
   return MI_ERROR;
```

The **mi_open_prepared_statement( )** function also provides the ability to name the cursor. However, if you specify a cursor name in **mi_prepare( )**, make sure that you pass a NULL-valued pointer as the cursor name to **mi_open_prepared_statement( )**. Conversely, if you want to specify the cursor name in **mi_open_prepared_statement( )**, use a NULL-valued pointer as the cursor name in **mi_prepare( )**. If you specify a cursor name in *both* **mi_prepare( )** and **mi_open_prepared_statement( )**, the DataBlade API uses the cursor name that **mi_open_prepared_statement( )** provides.

If your prepared statement does not fetch rows, pass a NULL-valued pointer as the third argument to **mi_prepare( )**.

Client Only

The last argument to **mi_prepare( )** specifies the statement name for the prepared statement. The *cursor_name* argument of **mi_open_prepared_statement( )** specifies the cursor name for the prepared statement. If you do not need to assign a statement name, pass a NULL-valued pointer as the last argument to **mi_prepare( )**.

End of Client Only

Returning a Statement Descriptor: The **mi_prepare( )** function sends the contents of an SQL statement string to the database server, which parses the statement and returns it in an optimized executable format. The function returns a pointer to an *explicit statement descriptor* (usually called just a *statement descriptor*). A statement descriptor, MI_STATEMENT, is a DataBlade API structure that contains the information about a prepared SQL statement, including the executable format of the SQL statement.

The following table summarizes the memory operations for a statement descriptor.

Memory Duration	Memory Operation	Function Name
Not allocated from memory-duration pools	Constructor	**mi_prepare( )**
	Destructor	**mi_drop_prepared_statement( )**, **mi_close_statement( )**

A statement descriptor can be identified in either of the following ways:

- As a pointer to an MI_STATEMENT structure, which **mi_prepare( )** returns

  The **mi_prepare( )** function is a constructor function for a statement descriptor.
- As an integer *statement identifier*, which the **mi_get_id( )** function returns when passed MI_STATEMENT_ID as its second argument

Table 8-4 lists the DataBlade API accessor functions for an explicit statement descriptor.

Table 8-4. Accessor Functions for an Explicit Statement Descriptor

Statement-Descriptor Information	DataBlade API Accessor Function
The name of the SQL statement that was prepared	**mi_statement_command_name( )**
Information about any input parameters in the prepared statement	The input-parameter accessor functions (Table 8-5 on page 8-15)
A row descriptor for the columns in the prepared statement	**mi_get_statement_row_desc( )** From the row descriptor, you can use the row-descriptor accessor functions to obtain information about a particular column (see Table 5-3 on page 5-30).

Important: To DataBlade API modules, the statement descriptor (MI_STATEMENT) is an opaque C structure. Do not access the internal fields of this structure directly. The internal structure of the MI_STATEMENT may

change in future releases. Therefore, to create portable code, always use these accessor functions to obtain prepared-statement information.

You pass a statement descriptor to the other DataBlade API functions that handle prepared statements, including **mi_exec_prepared_statement( )**, **mi_open_prepared_statement( )**, **mi_fetch_statement( )**, **mi_close_statement( )**, and **mi_drop_prepared_statement( )**.

Obtaining Input-Parameter Information

From a statement descriptor, you can obtain information about an input parameter once an SQL statement has been prepared. An input parameter indicates a value that is provided when the prepared statement executes. Table 8-5 lists the DataBlade API accessor functions that obtain input-parameter information from the statement descriptor.

Table 8-5. Input-Parameter Information in the Statement Descriptor

Column Information	DataBlade API Accessor Function
The *number* of input parameters in the prepared statement	**mi_parameter_count( )**
The *precision* (total number of digits) of the column associated with an input parameter	**mi_parameter_precision( )**
The *scale* of a column that is associated with the input parameter	**mi_parameter_scale( )**
Whether the column associated with each input parameter was defined with the *NOT NULL* constraint	**mi_parameter_nullable( )**
The *type identifier* of the column that is associated with the input parameter	**mi_parameter_type_id( )**
The *type name* of the column that is associated with the input parameter	**mi_parameter_type_name( )**

Important: To DataBlade API modules, the input-parameter information in the statement descriptor (MI_STATEMENT) is part of an opaque C data structure. Do not access the internal fields of this structure directly. The internal structure of the MI_STATEMENT structure may change in future releases. Therefore, to create portable code, always use these accessor functions to obtain input-parameter information.

Input-parameter information is available *only* for the INSERT and UPDATE statements. Support for the UPDATE statement includes the following forms of UPDATE:

- UPDATE with or without a WHERE clause
- UPDATE WHERE CURRENT OF

If you attempt to request input-parameter information for other SQL statements, the input-parameter functions in Table 8-5 raise an exception.

The statement descriptor stores input-parameter information in several parallel arrays.

Input-Parameter Array	Contents
Parameter-type ID array	Each element is a pointer to a type identifier (MI_TYPEID) that indicates the data type of the input parameter.

Input-Parameter Array	Contents
Parameter-type name array	Each element is a pointer to the string name of the data type for each input parameter.
Parameter-scale array	Each element is the scale of the column associated with the input parameter.
Parameter-precision array	Each element is the precision of the column associated with the input parameter.
Parameter-nullable array	Each element is either MI_FALSE or MI_TRUE: • MI_FALSE indicates that the input parameter is associated with a column that *cannot* contain SQL NULL values. • MI_TRUE indicates that the input parameter is associated with a column that *can* contain SQL NULL values.

All of the input-parameter arrays in the statement descriptor have zero-based indexes. Within the statement descriptor, each input parameter in the prepared statement has a *parameter identifier*, which is the zero-based position of the input parameter within the input-parameter arrays. When you need information about an input parameter, specify its parameter identifier to one of the statement-descriptor accessor functions in Table 8-5 on page 8-15.

Figure 8-5 shows how the information at index position 1 of these arrays holds the input-parameter information for the second input parameter of a prepared statement.

All information for the second input parameter
(at index position 1)
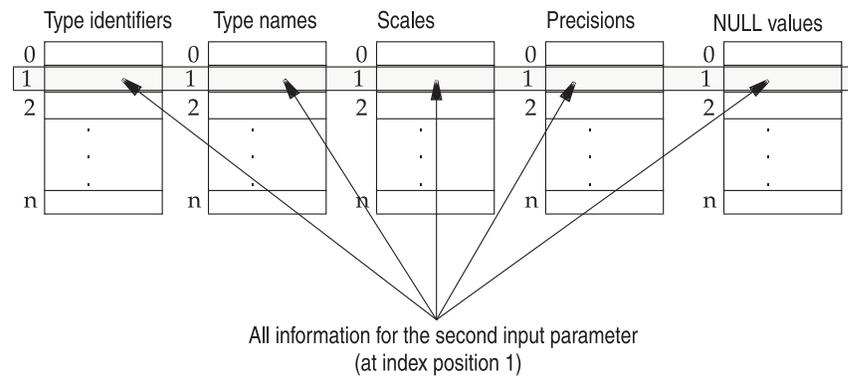
Figure 8-5. Input-Parameter Arrays in the Statement Descriptor

To access information for the *n*th input parameter, provide an index value of *n*-1 to the appropriate accessor function in Table 8-5 on page 8-15. The following calls to the **mi_parameter_type_id( )** and **mi_parameter_nullable( )** functions obtain from the statement descriptor that **stmt_desc** identifies the type identifier (**param_type**) and whether the column is nullable (**param_nullable**) for the *second* input parameter:

```
MI_STATEMENT *stmt_desc;
MI_TYPEID *param_type;
mi_integer param_nullable;
...
param_type = mi_parameter_type_id(stmt_desc, 1);
param_nullable = mi_parameter_nullable(stmt_desc, 1);
```

To obtain the number of input parameters in the prepared statement (which is also the number of elements in the input-parameter arrays), use the **mi_parameter_count( )** function.

Sending the Prepared Statement

For a prepared statement to be executed, you must send it to the database server with one of the following DataBlade API functions.

DataBlade API Function	When To Use
mi_exec_prepared_statement( )	If the prepared statement does *not* return rows
	If the prepared statement does return rows but you only need to access these rows sequentially (with an implicit cursor)
mi_open_prepared_statement( )	If the prepared statement *does* return rows and you need to perform one of the following tasks: • Access these rows with a scroll, update, or hold cursor (instead of a read-only sequential cursor) • Control how many rows the database server puts into the cursor at one time

Both these functions support the following parameters.

Parameter	Description
stmt_desc	Is a pointer to a statement descriptor for the prepared statement The **mi_prepare( )** function generates this statement descriptor.
control flag	Determines whether any query rows are in binary or text representation
params_are_binary	Indicates whether the input-parameter values are in binary or text representation
n_params	Is the number of input-parameter values in the input-parameter-value arrays
Input-parameter-value arrays: • *values* • *types* • *lengths* • *nulls*	Arrays that contain the following information for each input-parameter value: • Value • Data type • Length (for varying-length data types) • Whether the input-parameter value is an SQL NULL value For more information, see "Assigning Values to Input Parameters" on page 8-27.
retlen	The number of column values that are in each retrieved row
rettypes	An array that contains the data types of any returned column values

Once the database server executes the prepared statement, the statement becomes the *current statement*. The database server sends back the statement results, including whether the current statement was successful. Obtain the status of the current statement with the **mi_get_result( )** function. For more information, see "Processing Statement Results" on page 8-33.

Tip: The return value that the **mi_exec_prepared_statement( )** or **mi_open_prepared_statement( )** function returns does not indicate the success of the current statement. It indicates if **mi_exec_prepared_statement( )** or **mi_open_prepared_statement( )** was able to successfully send the prepared statement to the database server.

Statements with mi_exec_prepared_statement( ): The **mi_exec_prepared_statement( )** function is for the execution of prepared statements, both queries and other valid SQL statements. In a DataBlade API module, use the following **DataBlade API** functions to execute a prepared SQL statement with **mi_exec_prepared_statement( )**.

DataBlade API Function	Step in Prepared-Statement Execution
mi_prepare( )	Prepares the statement string for execution
mi_statement_command_name( ), **mi_get_statement_row_desc( )**, or input-parameter accessor function (Table 8-5 on page 8-15)	Obtains information about the prepared statement (*optional*)
mi_exec_prepared_statement( )	Sends the prepared statement to the database server for execution and opens any cursor required
mi_drop_prepared_statement( )	Releases prepared-statement resources

The **mi_exec_prepared_statement( )** function performs the following tasks for the prepared SQL statement:

- Binds any input-parameter values to the appropriate input parameters in the prepared statement

  For more information, see "Assigning Values to Input Parameters" on page 8-27.

- Sends the prepared statement to the database server for execution

  The *control* flag supports the MI_BINARY flag to indicate that query rows are to be returned in binary representation. For more information, see "Determining Control Mode for Query Data" on page 8-30.

- When it executes a query, it performs the following additional steps:
  - Opens an implicit cursor to hold the query rows
  - Reads the query rows into the open cursor

  The DataBlade API stores the cursor as part of the statement descriptor. For more information on this row cursor, see "Queries and Implicit Cursors" on page 8-5.

Tip: If the implicit cursor that **mi_exec_prepared_statement( )** creates does not adequately meet the needs of your DataBlade API module, you can use the **mi_open_prepared_statement( )** function to define other types of cursors. For more information, see "Defining an Explicit Cursor" on page 8-22.

When the **mi_exec_prepared_statement( )** function successfully fetches the query rows into the cursor, the *cursor position* points to the *first* row of the cursor, and the **mi_get_result( )** function returns a status of MI_ROWS to indicate that the cursor contains rows.

You can access these rows one at a time with the **mi_next_row( )** function. Each access obtains the row to which the cursor position points. After each access to the cursor, the cursor position moves to the next row. For more information, see "Retrieving Query Data" on page 8-39.

The following variation of the **send_statement( )** function (page 8-10) uses **mi_exec_prepared_statement( )** instead of **mi_exec( )** to send an SQL statement to the database server:

```
mi_integer send_statement2(conn, stmt)
   MI_CONNECTION *conn;
   mi_string *stmt;
{
   mi_integer count;
   MI_STATEMENT *stmt_desc;

   /* Prepare the statement */
   if ( (stmt_desc = mi_prepare(conn, stmt, NULL)) == NULL )
      mi_db_error_raise(conn, MI_EXCEPTION,
         "mi_prepared failed\n");

   /* Send the basic statement, specifying that query
    * be sent in its text representation
    */
   if ( mi_exec_prepared_statement(stmt_desc, 0, MI_FALSE,
         0, NULL, NULL, NULL, 0, NULL) == MI_ERROR )
      mi_db_error_raise(conn, MI_EXCEPTION,
         "mi_exec_prepared_statement failed\n");

   /* Get the results of the current statement */
   count = get_results(conn);

   /* Release statement resources */
   if ( mi_drop_prepared_statement(stmt_desc) == MI_ERROR )
      mi_db_error_raise(conn, MI_EXCEPTION,
         "mi_drop_prepared_statement failed\n");
   if ( mi_query_finish(conn) == MI_ERROR )
      mi_db_error_raise(conn, MI_EXCEPTION,
         "mi_query_finish failed\n");

   return ( count );
}
```

The **mi_exec_prepared_statement( )** function allocates type descriptors for each of the data types of the input parameters. If the calls to **mi_exec_prepared_statement( )** are in a loop in which these data types do not vary between loop iterations, **mi_exec_prepared_statement( )** can reuse the type descriptors, as follows:

- On the first call to **mi_exec_prepared_statement( )**, specify in the *types* array the correct data type names for the input parameters.
- On subsequent calls to **mi_exec_prepared_statement( )**, replace the array of data type names with a NULL-valued pointer.

This method saves on the number of type descriptors that **mi_exec_prepared_statement( )** must allocate, thereby reducing memory usage.

In Figure 8-6, **mi_exec_prepared_statement( )** in the initial pass of the **for** loop specifies the INTEGER data type for the single input parameter in an INSERT statement. For subsequent passes of the **for** loop, **mi_exec_prepared_statement( )** receives a NULL-valued pointer for its types array. When it receives this NULL-valued pointer, **mi_exec_prepared_statement( )** reuses the type descriptor

that it has already created.

```
mi_string *types[1] = {"int"};
mi_string **types_exec;
...
sprintf(command, "insert into tabA values(?, %d);", j);
if ( (stmt_desc = mi_prepare(conn, command, NULL)) == NULL )
    {
    return -1;
    }

types_exec = types;
for (j=0; j < numLoop; j++)
    {
    values[0] = (MI_DATUM) j;

    if ( (ret = mi_exec_prepared_statement(stmt_desc,
            MI_BINARY, 1, 1, values, lengths, nulls,
            types_exec, 0, NULL)) )
        {
        return -2;
        }

    if ( (ret = mi_get_result(conn)) == MI_ERROR )
        return -4;

    if ( ret == MI_DML || MI_DDL )
        row_count += mi_result_row_count(conn);

    types_exec = NULL; /* reuse data types from 1st pass */
    }
```

Figure 8-6. Reusing Type Descriptors in Repeated Calls to mi_exec_prepared_statement( )

Statements with mi_open_prepared_statement( ): The
mi_open_prepared_statement( ) function is for the execution of queries. In a
DataBlade API module, use the following **DataBlade API** functions to execute a
prepared SQL statement with **mi_open_prepared_statement( )**.

DataBlade API Function	Step in Execution of Prepared Statement
mi_prepare( )	Prepares the statement string for execution
mi_statement_command_name( ), **mi_get_statement_row_desc( )**, or input-parameter accessor function (Table 8-5 on page 8-15)	Obtains information about the prepared statement (optional)
mi_open_prepared_statement( )	Sends the prepared statement to the database server for execution and open the cursor
mi_fetch_statement( )	Retrieves any data that the query returns
mi_close_statement( ), **mi_drop_prepared_statement( )**	Releases prepared-statement resources

The **mi_open_prepared_statement( )** function performs the following tasks for the
prepared SQL statement:
• Binds any input-parameter values to the appropriate input parameters in the
  prepared statement

  For more information on how to assign input-parameter values, see "Assigning
  Values to Input Parameters" on page 8-27.

- Sends the prepared statement to the database server for execution
- Creates and opens an explicit cursor with characteristics specified in the *control* argument

  The DataBlade API stores the cursor as part of the statement descriptor. For more information on this cursor, see "Defining an Explicit Cursor" on page 8-22.

Tip: The **mi_open_prepared_statement( )** function performs the same basic task for a DataBlade API module as the SQL OPEN statement does for an IBM Informix ESQL/C application.

The main difference between **mi_exec_prepared_statement( )** and **mi_open_prepared_statement( )** is that the latter allows more flexibility in the definition of the cursor used for the query rows. With **mi_open_prepared_statement( )**, you can define an explicit cursor. In particular, **mi_open_prepared_statement( )** allows you to specify:

- A string name to assign to the cursor

  The *cursor_name* parameter is a pointer to the string name that you want to assign to the cursor. You can use this *cursor_name* for an update cursor so that the UPDATE or DELETE statement can reference the cursor in its clause:

  WHERE CURRENT OF *cursor_name*

  For more information, see "Assigning an Optional Name" on page 8-12.

  ───────────────────── Client Only ─────────────────────

  To use an internally-generated unique name for the cursor, specify a NULL-valued pointer for the *cursor_name* argument.

  ───────────────────── End of Client Only ─────────────────────

- The type of cursor to use for holding the query rows

  The **mi_open_prepared_statement( )** function supports several flag values in its *control* flag that determine the type of cursor it creates. For more information, see "Defining an Explicit Cursor" on page 8-22.

  In addition, the *control* flag also supports the MI_BINARY flag to indicate that query rows are to be returned in binary representation. For more information, see "Determining Control Mode for Query Data" on page 8-30.

- The number of rows to read into the cursor at one time

  Unlike **mi_exec( )** and **mi_exec_prepared_statement( )**, **mi_open_prepared_statement( )** does *not* read any retrieved rows into the open cursor. To fetch rows into the explicit cursor, use the **mi_fetch_statement( )** function. For more information, see "Fetching Rows Into a Cursor" on page 8-23.

The **mi_open_prepared_statement( )** function allocates type descriptors for each of the data types of the input parameters. If the calls to **mi_open_prepared_statement( )** are in a loop in which these data types do not vary between loop iterations, **mi_open_prepared_statement( )** can reuse the type descriptors, as follows:

- On the first call to **mi_open_prepared_statement( )**, specify in the *types* array the correct data type names for the input parameters.
- On subsequent calls to **mi_open_prepared_statement( )**, replace the array of data type names with a NULL-valued pointer.

This method saves on the number of type descriptors that **mi_open_prepared_statement( )** must allocate, thereby reducing memory usage. For sample code in which the **mi_exec_prepared_statement( )** function reuses type descriptors, see Figure 8-6 on page 8-20.

Defining an Explicit Cursor: The *control* flag of **mi_open_prepared_statement( )** allows you to define an *explicit cursor* to hold the rows that the prepared query returns. You can choose the following cursor characteristics when you define the cursor:

- The cursor type
- The cursor mode
- The cursor lifespan

Defining a Cursor Type: The **mi_open_prepared_statement( )** function supports the following types of cursors for holding query rows.

Cursor Type	Description
Sequential cursor	Enables you to move through the rows of the cursor in the *forward* direction only
	You can pass only once through the rows.
Scroll cursor	Enables you to move through the rows of the cursor in the *forward and backward* directions
	You can move back in the rows without having to reopen the cursor; however, the database server stores the data for a scroll cursor in a temporary table. The data can become stale; that is, the data in the cursor is consistent with the data in the database when the cursor is filled, but if the data in the database changes, the data in the cursor does *not* reflect these changes.

Table 8-6 shows the control-flag values that determine cursor type and cursor mode.

Defining a Cursor Mode: You can specify one of the following *cursor modes* for the cursor with the *control*-flag bit mask.

Cursor Mode	Description	SELECT Statement
Update	Enables you to read and modify the data within the cursor	SELECT...FOR UPDATE
Read-only	Enables you to read the data within the cursor; does not allow you to update or delete any row it fetches	SELECT...FOR READ ONLY

When you execute a prepared SELECT statement with no FOR UPDATE or FOR READ ONLY clause, the cursor mode you need depends on whether your database is ANSI-compliant, as follows:

- In a database that is *not* ANSI compliant, the SELECT statement specifies a read-only mode by default.

   You do not need to specify the FOR READ ONLY keywords in the SELECT statement. The only advantage of specifying the FOR READ ONLY keywords explicitly is for better program documentation. To specify an update mode, you

must specify the FOR UPDATE keywords in the SELECT statement.

For more information on the FOR UPDATE and FOR READ ONLY clauses, see the
description of the SELECT statement in the *IBM Informix Guide to SQL: Syntax*.

By default, both the sequential and scroll cursor types have a cursor mode of
update (also called *read/write*). Table 8-6 shows the cursor types and cursor modes,
with the required bit-mask values for the *control* flag.

Table 8-6. Control-Flag Values for Cursor Type and Mode

Cursor	Control-Flag Value
Update sequential cursor	None (default)
Read-only sequential cursor	MI_SEND_READ
Update scroll cursor	MI_SEND_SCROLL
Read-only scroll cursor	MI_SEND_READ + MI_SEND_SCROLL

Defining a Cursor Lifespan: You can define the lifespan of the cursor with the
control-flag bit mask. By default, the database server closes all cursors at the end of
a transaction. If your DataBlade API module requires uninterrupted access to a set
of rows across transaction boundaries, you can define a *hold* cursor. A hold cursor
can be either a sequential or a scroll cursor.

To define a hold cursor, you specify the MI_SEND_HOLD constant in the
control-flag bit mask of the **mi_open_prepared_statement( )** function, as the
following table shows.

Cursor	Control-Flag Value
Update sequential cursor *with hold*	MI_SEND_HOLD
Read-only sequential cursor *with hold*	MI_SEND_READ + MI_SEND_HOLD
Update scroll cursor *with hold*	MI_SEND_SCROLL + MI_SEND_HOLD
Read-only scroll cursor *with hold*	MI_SEND_READ + MI_SEND_SCROLL + MI_SEND_HOLD

Fetching Rows Into a Cursor: When **mi_open_prepared_statement( )** successfully
opens a cursor, the cursor is *empty*, with the *cursor position* pointing to the *first*
location of the cursor, and the **mi_get_result( )** function returns a status of
MI_NO_MORE_RESULTS to indicate that the cursor does *not* contain rows.

Figure 8-7 shows the state of the explicit cursor that contains one integer column
after **mi_open_prepared_statement( )** executes.
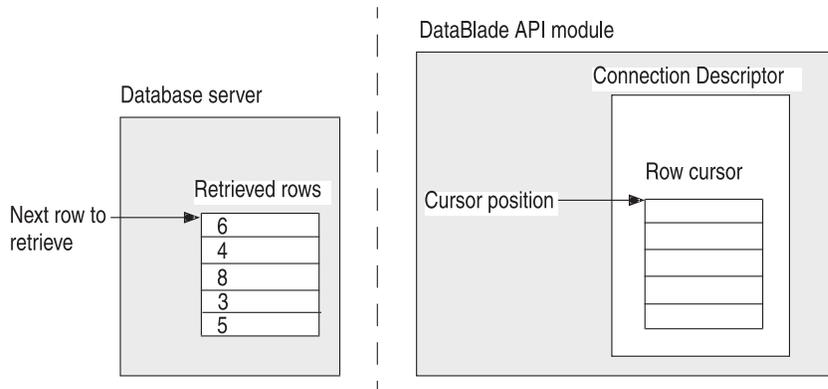
Figure 8-7. Row Cursor After mi_open_prepared_statement( )

To populate the open cursor, use the **mi_fetch_statement( )** function, which *fetches* the specified number of retrieved rows from the database server into the cursor. You can perform a fetch operation on an update or a read-only cursor. To fetch rows into a cursor, you must specify the following information to **mi_fetch_statement( )**:

- The statement descriptor for the prepared statement that returns rows
- The location in the rows on the database server at which to begin the fetch
- The number of rows to fetch into the cursor

The **mi_fetch_statement( )** function requests the specified number of retrieved rows from the database server and copies them into the cursor, which is associated with the specified statement descriptor. When **mi_fetch_statement( )** completes successfully, the following items are true:

- The cursor contains the number of rows that the *num_rows* argument specifies.
- The cursor position points to the first of the fetched rows in the cursor.
- The **mi_get_result( )** function returns a status of MI_ROWS to indicate that the cursor *does* contain rows.

With **mi_fetch_statement( )**, you can request rows at different locations based on the type of cursor that **mi_open_prepared_statement( )** has defined. To specify location, **mi_fetch_statement( )** has an *action* argument of type MI_CURSOR_ACTION, which supports the cursor-action constants in the following table.

Cursor-Action Flag	Description	Type of Cursor
MI_CURSOR_NEXT	Fetches the next *num_rows* rows, starting at the current retrieved row on the database server	Sequential Scroll
MI_CURSOR_PRIOR	Fetches the previous *num_rows* rows, starting at the current retrieved row	Scroll
MI_CURSOR_FIRST	Fetches the first *num_rows* rows	Sequential Scroll
MI_CURSOR_LAST	Fetches the last *num_rows* rows	Sequential Scroll

Cursor-Action Flag	Description	Type of Cursor
MI_CURSOR_ABSOLUTE	Moves *jump* rows into the retrieved rows and fetches *num_rows* rows	Sequential (as long as the *jump* argument does not move the cursor position backward)
		Scroll
MI_CURSOR_RELATIVE	Moves *jump* rows from the current retrieved row and fetch *num_rows* rows	Sequential (as long as the *jump* argument is a positive number)
		Scroll

Figure 8-8 shows the state of a row cursor that Figure 8-7 on page 8-24 defines after the following **mi_fetch_statement( )** executes:

```
mi_fetch_statement(stmt_desc, MI_CURSOR_NEXT, 0, 0);
```
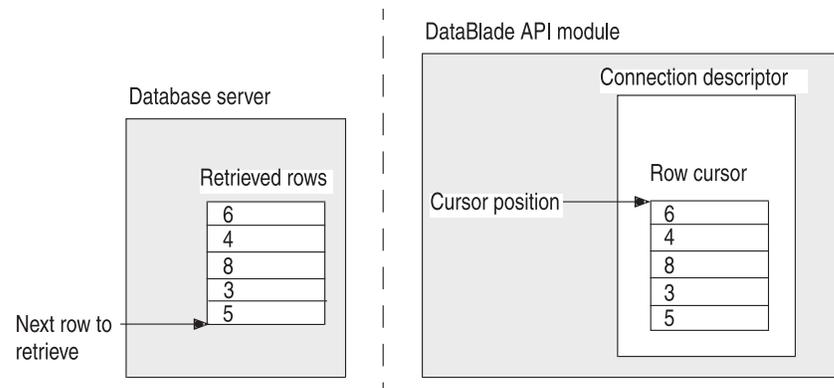

Figure 8-8. Fetching All Retrieved Rows Into a Cursor

Once the rows reside in the cursor, your DataBlade API module can access these rows one at a time with the **mi_next_row( )** function. For more information, see "Retrieving Query Data" on page 8-39.

If you specify a non-zero value for the *num_rows* argument, **mi_fetch_statement( )** fetches the requested number of rows into the cursor. Specify a non-zero value for *num_rows* if your DataBlade API module needs to handle rows in smaller groups. In this case, you retrieve *num_rows* number of query rows from the cursor with **mi_next_row( )**. When **mi_next_row( )** indicates that no more rows are in the cursor, you must determine whether to fetch any remaining rows from the database server into the cursor, as follows:

- If you do *not* need to examine additional rows, exit the **mi_next_row( )** and **mi_get_result( )** loops normally and close the cursor with **mi_close_statement( )**.
- If you *do* need to fetch any rows remaining on the database server into the cursor, execute the **mi_fetch_statement( )** function *again* after the following conditions occur:
  – The **mi_get_result( )** function returns MI_DML (for a SELECT statement).
  – The number of query rows that **mi_next_row( )** obtains is less than the number of rows that **mi_fetch_statement( )** fetches (*num_rows*) from the database server.

You can obtain the number of query rows with the **mi_result_row_count( )** function.

The **mi_fetch_statement( )** for Figure 8-8 on page 8-25 specified a value of zero (0) as the number of rows to fetch, which tells the function to fetch *all* retrieved rows. Figure 8-9 shows the state of the row cursor that Figure 8-7 on page 8-24 defines when the **mi_fetch_statement( )** function specifies a *num_rows* argument of three instead of zero, as follows:

```
mi_fetch_statement(stmt_desc, MI_CURSOR_NEXT, 0, 3);
```
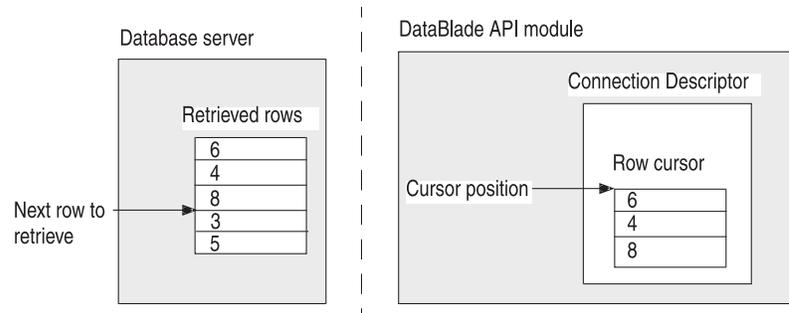

Figure 8-9. Fetching First Three Rows into a Cursor

--- Server Only ---

The following code fragment uses the **mi_open_prepared_statement( )** function to assign an input-parameter value, execute a SELECT statement, and retrieve the query rows:

```
mi_string *cmd =
    "select order_num from orders \
     where customer_num = ?;";
MI_STATEMENT *stmt;
...
if ( (stmt = mi_prepare(conn, cmd, NULL)) == NULL )
   mi_db_error_raise(NULL, MI_EXCEPTION,
      "mi_prepare( ) failed");

values[0] = 104;
types[0] = "integer";
lengths[0] = 0;
nulls[0] = MI_FALSE;

/* Open the read-only cursor to hold the query rows */
if ( mi_open_prepared_statement(stmt, MI_SEND_READ,
      MI_TRUE, 1, values, lengths, nulls, types,
      "cust_select", retlen, rettypes)
      != MI_OK )
   mi_db_error_raise(NULL, MI_EXCEPTION,
      "mi_open_prepared_statement( ) failed");

/* Fetch the retrieved rows into the cursor */
if ( mi_fetch_statement(stmt, MI_CURSOR_NEXT, 0, 3) != MI_OK )
   mi_db_error_raise(NULL, MI_EXCEPTION,
      "mi_fetch_statement( ) failed");

if ( mi_get_result(conn) != MI_ROWS )
   mi_db_error_raise(NULL, MI_EXCEPTION,
   "mi_get_result( ) failed or found nonquery statement");

/* Retrieve the query rows from the cursor */
```

```
if ( !(get_data(conn)) )
   mi_db_error_raise(NULL, MI_EXCEPTION,
      "get_data( ) failed");

/* Close the cursor */
if ( mi_close_statement(stmt) == MI_ERROR )
   mi_db_error_raise(NULL, MI_EXCEPTION,
      "mi_close_statement( ) failed");

/* Release resources */
if ( mi_drop_prepared_statement(stmt) == MI_ERROR )
   mi_db_error_raise(NULL, MI_EXCEPTION,
      "mi_drop_prepared_statement( ) failed");
if ( mi_close(conn) == MI_ERROR )
   mi_db_error_raise(NULL, MI_EXCEPTION,
      "mi_close( ) failed");
```

This code fragment sends its input-parameter value in binary representation. The code fragment is part of a C UDR because it passes the INTEGER input-parameter value by value. For more information, see "Assigning Values to Input Parameters" on page 8-27.

────────────────── **End of Server Only** ──────────────────

Assigning Values to Input Parameters: For a parameterized SQL statement, your DataBlade API module must perform the following steps:

1. Specify input parameters in the text of the SQL statement.
2. Send the SQL statement to the database server for parsing.
3. Provide input-parameter values when the SQL statement executes.

The **mi_prepare( )** statement performs these first two steps for a parameterized statement. For more information, see "Preparing an SQL Statement" on page 8-11.

The **mi_exec_prepared_statement( )** and **mi_open_prepared_statement( )** functions assign values to input parameters when they send a parameterized SQL to the database server for execution. To provide a value for an input parameter, you pass information in several parallel arrays:

• Parameter-value array
• Parameter-value type array
• Parameter-value length array
• Parameter-value null array

These input-parameter value arrays are similar to the input-parameter arrays in the statement descriptor (see Figure 8-5 on page 8-16). They have an element for each input parameter in the prepared statement. However, they are unlike the input-parameter arrays in the statement descriptor in the following ways:

• An input-parameter value array describes the actual value for an input parameter.

   An input-parameter array describes the column with which the input parameter is associated.

• You must allocate and manage an input-parameter value array.

   The DataBlade API does *not* provide accessor functions for input-parameter value arrays. For each input parameter, your DataBlade API module must declare, allocate, and assign a value to the array.

All of the input-parameter-value arrays have zero-based indexes. Figure 8-10 shows how the information at index position 1 of these arrays holds the input-parameter-value information for the second input parameter of a prepared statement.
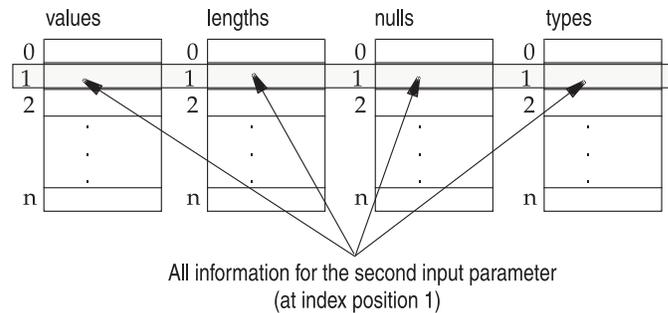
Figure 8-10. Arrays for Initialization of Input Parameters

You specify the number of input-parameter values in the input-parameter value arrays with the *nparams* argument of **mi_exec_prepared_statement( )** or **mi_open_prepared_statement( )**.

The following sections provide additional information about each of the input-parameter-value arrays.

Parameter-Value Array: The parameter-value array, *values*, is the fifth argument of the **mi_exec_prepared_statement( )** and **mi_open_prepared_statement( )** functions. Each element of the parameter-value array is a pointer to an **MI_DATUM** structure that holds the value for each input parameter. The format of this value depends on:

- Whether the control mode for the input-parameter data is text or binary representation:

  The *params_are_binary* argument of **mi_exec_prepared_statement( )** or **mi_open_prepared_statement( )** indicates this control mode. For more information on the format of data for different control modes, see Table 8-3 on page 8-8.

- In binary representation, whether the **MI_DATUM** value is passed by reference or by value:

  Server Only

  – For C UDRs, the data type of the value determines the passing mechanism.

  End of Server Only

  Client Only

  – For client LIBMI applications, pass *all* values (regardless of data type) by reference.

  End of Client Only

For more information, see "Contents of an MI_DATUM Structure" on page 2-33.

For the prepared INSERT statement in Figure 8-4 on page 8-12, the code fragment in Figure 8-11 assigns values to the input parameters for the **customer_num** and

company columns. These values are in text representation because the *params_are_binary* argument of **mi_exec_prepared_statement( )** is MI_FALSE.

```
/* Initialize input parameter for customer_num column */
values[0] = "0"; /* value of '0' for SERIAL customer_num */
lengths[0] = 0; /* SERIAL is built-in type: no length */
types[0] = "serial";
nulls[0] = MI_FALSE;

/* Initialize input parameter for company column */
stcopy("Trievers Inc.", strng);
values[1] = strng;
lengths[1] = stleng(strng); /* CHAR types need length! */
types[1] = "char(20)";
nulls[1] = MI_FALSE;

/* Send INSERT statement to database server for execution along
 * with the input-parameter-value arrays
 */
mi_exec_prepared_statement(insrt_stdesc, 0, MI_FALSE, 2
   values, lengths, nulls, types, 0, NULL);
```

Figure 8-11. Executing a Statement That Contains Text-Representation Input Parameters

---
Server Only

The following code fragment initializes the input parameters to the same values but it assigns these values in binary representation instead of text representation:

```
/* Initialize input parameter for customer_num column */
values[0] = 0; /* value of 0 for SERIAL customer_num */
lengths[0] = 0; /* SERIAL is built-in type: no length */
types[0] = "serial";
nulls[0] = MI_FALSE;

/* Initialize input parameter for company column */
values[1] = mi_string_to_lvarchar("Trievers Inc.");
lengths[1] = 0; /* CHAR types need length! */
types[1] = "char(20)";
nulls[1] = MI_FALSE;

/* Send INSERT statement to database server for execution along
 * with the input-parameter-value arrays
 */
mi_exec_prepared_statement(insrt_stdesc, 0, MI_TRUE, 2
   values, lengths, nulls, types, 0, NULL);
```

In the preceding code fragment, the first element of the **values** array is assigned an integer value. Because this code executes in a C UDR, the integer value in the **MI_DATUM** structure of this array must be passed by value.

End of Server Only

Client Only

In a client LIBMI application, *all* values in **MI_DATUM** structure must be passed by reference. Therefore, to assign values to input parameters within a client LIBMI application, you must assign all values in the *values* array as pointers to the actual values.

The preceding code fragment assumes that it executes within a C UDR because it passes the value for the first input parameter (an INTEGER column by value). In a

client LIBMI application, you cannot use the pass-by-value mechanism. Therefore, the assignment to the first input parameter must pass a pointer to the integer value, as follows:

```
col1 = 0;
values[0] = &col1;
```

<hr>

────────────────────── **End of Client Only** ──────────────────────

Parameter-Value Length Array: The parameter-value length array, *lengths*, is the sixth argument of the **mi_exec_prepared_statement( )** and **mi_open_prepared_statement( )** functions. Each element of the parameter-value length array is the integer length (in bytes) of the data type for the input-parameter value.

The meaning of the values in the *lengths* array depends on the control mode of the input-parameter values, as follows:

- For input-parameter values that are in text representation, the *lengths* array lists the lengths of the text strings.

  Make sure the *lengths* value matches the length of the null-terminated string of the input-parameter value (minus the null terminator). Use a library function, such as **strlen( )** or **stleng( )**, to determine the string length.

- For input-parameter values that are in binary representation, the database server does not need to access the entry of the parameter-value length array.

  Lengths are *not* needed in the following special cases:

  – Input parameters whose data types (such as **mi_integer**) are passed with fixed lengths

  – Input parameters with string values are passed in varying-length structures

    A varying-length structure holds its own data length.

Important: Even though there are some cases in which the database server does not read the length of the input-parameter value, it is recommended that you always specify lengths to maintain consistency of code.

Parameter-Value Null Array: The parameter-value null array, *nulls*, is the seventh argument of the **mi_exec_prepared_statement( )** and **mi_open_prepared_statement( )** functions. Each element of the parameter-value null array is either:

- MI_FALSE: the input-parameter value is *not* an SQL NULL value
- MI_TRUE: the input-parameter value is an SQL NULL value

Parameter-Value Type Array: The parameter-value type array, *types*, is the eighth argument of the **mi_exec_prepared_statement( )** and **mi_open_prepared_statement( )** functions. Each element of the parameter-value type array is a pointer to a string that identifies the data type of the input-parameter value. The type names must match those that the **mi_type_typename( )** function would generate for the data type.

If the prepared statement has input parameters and is *not* an INSERT statement, you must use the *types* array to supply the data types of the input parameters. Otherwise, you can pass in a NULL-valued pointer as the *types* argument.

Determining Control Mode for Query Data: The **mi_exec_prepared_statement( )** and **mi_open_prepared_statement( )** functions specify the control mode for the

data of a prepared query in their bit-mask *control* argument. To determine the control mode, set the *control* argument as the following table shows.

Control Mode	Control Argument
Text representation	Zero *(default)*
Binary representation	MI_BINARY

For **mi_exec_prepared_statement( )**, MI_BINARY is the only valid control-flag constant for the *control* argument. Therefore, a default value of zero (0) as the *control* argument indicates text representation of the data. The following **mi_exec_prepared_statement( )** call specifies a control mode of binary representation:

```
mi_open_prepared_statement(stmt_desc, MI_BINARY, ...);
```

For **mi_open_prepared_statement( )**, the *control* argument indicates the cursor characteristics in addition to the control mode. To specify a text representation, omit the MI_BINARY control-flag constant from the *control* argument. Including MI_BINARY in the *control* argument indicates that results are to be returned in binary representation. The following **mi_open_prepared_statement( )** call specifies an update scroll cursor and a control mode of binary representation:

```
mi_open_prepared_statement(
   stmt_desc,
   MI_BINARY + MI_SEND_SCROLL,
   ...);
```

For information on how to specify cursor characteristics to **mi_open_prepared_statement( )**, see "Defining an Explicit Cursor" on page 8-22. For more information on the control mode, see "Control Modes for Query Data" on page 8-8.

Releasing Prepared-Statement Resources

When your DataBlade API module no longer needs a prepared statement, you can release the resources that it uses with the following DataBlade API functions.

Prepared-Statement Resource	DataBlade API Function
Explicit cursor	**mi_close_statement( )**
Statement descriptor (including any cursor)	**mi_drop_prepared_statement( )**

Closing a Statement Cursor: For prepared queries (SQL statements that return rows), the statement descriptor has a cursor associated with it. The scope of this cursor is from the time it is opened, with **mi_exec_prepared_statement( )** or **mi_open_prepared_statement( )**, until one of the following events occurs:

- The **mi_close_statement( )** function closes the cursor (explicit cursors only).
- The **mi_drop_prepared_statement( )** function frees the statement descriptor.
- The **mi_close( )** function closes the connection.

--- Server Only ---

- The SQL statement that invoked the C UDR ends.

--- End of Server Only ---

To conserve resources, use the **mi_close_statement( )** function to explicitly close an explicit cursor once your DataBlade API module no longer needs it. The

mi_close_statement( ) function is the destructor function for an explicit cursor that is associated with a statement descriptor. That is, it frees the cursor that the **mi_open_prepared_statement( )** function opens. Until you drop the prepared statement with **mi_drop_prepared_statement( )**, you can still reopen an explicit cursor with another call to **mi_open_prepared_statement( )**.

Tip: The **mi_close_statement( )** function performs the same basic task for a DataBlade API module as the SQL CLOSE statement does for an IBM Informix ESQL/C application.

The **mi_close_statement( )** function is *not* the destructor function for an implicit cursor that is associated with a statement descriptor. That is, it does *not* free the cursor that the **mi_exec_prepared_statement( )** function opens. To close an implicit cursor, use the **mi_drop_prepared_statement( )** function.

Dropping a Prepared Statement: A statement descriptor describes a prepared statement. However, this DataBlade API structure is not allocated from the memory-duration pools. Instead, its scope is from the time it is created with **mi_prepare( )** until whichever of the following events occurs first:

- The **mi_drop_prepared_statement( )** function frees the statement descriptor.
- The **mi_close( )** function closes the connection.

—————————————— **Server Only** ——————————————

- The SQL statement that invoked the C UDR ends.

—————————————— **End of Server Only** ——————————————

To conserve resources, use the **mi_drop_prepared_statement( )** function to explicitly deallocate the statement descriptor once your DataBlade API module no longer needs it. The **mi_drop_prepared_statement( )** function is the destructor function for a statement descriptor. It frees the statement descriptor and any resources (such as an implicit or explicit cursor) that are associated with the statement descriptor. These resources include the prepared statement and any associated row cursor. Once you drop a prepared statement, you must reprepare it before it can be executed again.

Executing Multiple SQL Statements

A DataBlade API statement-execution function can send more than one SQL statement to the database server at a time. In this case, the statement string contains several SQL statements, each terminated by a semicolon (;). When the statement string contains more than one SQL statement, the **mi_get_result( )** function executes for *each* statement in the string.

Suppose you send the following statement string for execution:

```
"insert into tab1 (id) values (1); \
 insert into tab1 (id) values (2); \
 insert into tab1 (id) values (3); "
```

For the preceding statement string, the **mi_get_result( )** function executes four times, three times returning an MI_DML status for each INSERT statement and once to return the final MI_NO_MORE_RESULTS status. For more information on **mi_get_result( )**, see "Processing Statement Results" on page 8-33.

Keep in mind that the effects of one part of the statement string are not visible to other parts. If one SQL statement depends on an earlier one, do *not* put them both in the same statement string. For example, the following statement string causes an error:

```
mi_exec(myconn, "create table tab1(a int, b int); \
   insert into tab1 values (1,2);",
   MI_QUERY_NORMAL);
```

The preceding **mi_exec( )** call generates an error because the INSERT statement cannot see the result of the CREATE TABLE statement. The solution is to call **mi_exec( )** twice, as follows:

```
/* Execute CREATE TABLE statement in first mi_exec( ) call */
mi_exec(myconn, "create table tab1 (a integer, b integer);",
   MI_QUERY_NORMAL);
mi_query_finish(myconn);

/* Execute INSERT statement in second mi_exec( ) call */
mi_exec(myconn, "insert into tab1 values (1,2);",
   MI_QUERY_NORMAL);
mi_query_finish(myconn);
```

Processing Statement Results

Once a DataBlade API statement-execution function (see Table 8-1 on page 8-3) executes, the SQL statement that it sent to the database server is the most recent SQL statement on the connection. This most recent SQL statement is called the *current statement*. Information about the current statement is associated with a connection. Only one statement is current at a time.

After you send the current statement to the database server for execution, your DataBlade API module must process the statement results by:

- Determining the status of the current statement, including whether results are available
- Retrieving any results

Retrieving the results of an SQL statement is a multiphase process that involves several levels of nested iteration, as the following table shows.

Statement-Processing Loop	Description	More Information
mi_get_result( ) loop	Outermost loop of the row-retrieval code iterates through each current statement.	"Executing the mi_get_result( ) Loop" on page 8-34
mi_next_row( ) loop	Middle loop of the row-retrieval code iterates through each row that the current statement has retrieved.	"Executing the mi_next_row( ) Loop" on page 8-42
Column-value loop	Innermost loop of the row-retrieval code iterates through each column value of a query row. This loop uses the **mi_value( )** or **mi_value_by_name( )** function to obtain the column values.	"Executing the Column-Value Loop" on page 8-43

The first step in processing statement results is to determine the status of the current statement with the **mi_get_result( )** function, as follows:

- Execute the **mi_get_result( )** function in a loop that iterates for each current statement.

- Interpret the statement status that **mi_get_result( )** returns for each current statement.

For a sample function that shows one way to use **mi_get_result( )** to process statement results, see "Example: The get_data( ) Function" on page 8-54.

Executing the mi_get_result( ) Loop

The **mi_get_result( )** function is usually executed in a loop after one of the DataBlade API statement-execution functions (in Table 8-1 on page 8-3) sends a statement to the database server. The function is normally called in the outermost loop of row-retrieval code. This loop executes for each of several states of the database server as it processes statement results. These states are represented as the status of the current statement. The **mi_get_result( )** function can return the following status information.

Information About Current SQL Statement	Statement-Status Constant
The current statement has generated an error.	MI_ERROR
The current statement is a data definition (DDL) statement that has completed successfully.	MI_DDL
The current statement is a data manipulation (DML) statement that has completed successfully.	MI_DML
The current statement is a query that has executed successfully.	MI_ROWS
No more results are pending for the current statement.	MI_NO_MORE_RESULTS

You can use a **switch** statement based on these statement-status constants to determine how to handle the status of the current statement. The **mi_get_result( )** loop terminates when **mi_get_result( )** returns the status MI_NO_MORE_RESULTS. Think of the **mi_get_result( )** loop as an iteration over the states of the database server.

Handling Unsuccessful Statements

The **mi_get_result( )** function returns a status of MI_ERROR to indicate that the current statement did *not* execute successfully. When **mi_get_result( )** returns this status, you can use the **mi_db_error_raise( )** function to raise a database server exception. If you have registered a callback on the MI_Exception event type, you can obtain an SQL status variable (**SQLCODE** or **SQLSTATE**) from the error descriptor that the database server passes to the callback. This SQL status variable can help determine the cause of the failure. For more information on how to handle the MI_Exception event, see "Database Server Exceptions" on page 10-20.

Handling a DDL Statement

The **mi_get_result( )** function returns a status of MI_DDL to indicate that the current statement was a DDL statement that has successfully executed. When you receive the MI_DDL statement status, you can use the **mi_result_command_name( )** function to obtain the name of the DDL statement that executed as the current statement.

The **mi_get_result( )** function returns an MI_DDL status for *any* SQL statement that is valid in a UDR and is *not* a DML statement (see Table 8-7 on page 8-36). For example, **mi_get_result( )** returns the MI_DDL status for a GRANT statement, even though SQL does not strictly consider GRANT as a DDL statement. However, the following SQL statements are *not* valid with a UDR that is called from within an INSERT, UPDATE, or DELETE statement in SQL:

ALTER ACCESS_METHOD	CREATE TRIGGER
ALTER FRAGMENT	CREATE VIEW
ALTER FUNCTION	CREATE XADATASOURCE
ALTER INDEX	CREATE XADATASOURCE TYPE
ALTER PROCEDURE	DROP ACCESS_METHOD
ALTER ROUTINE	DROP AGGREGATE
ALTER SEQUENCE	DROP CAST
ALTER SECURITY LABEL COMPONENT	DROP DATABASE
ALTER TABLE	DROP DUPLICATE
CLOSE DATABASE	DROP FUNCTION
CREATE ACCESS_METHOD	DROP INDEX
CREATE AGGREGATE	DROP OPCLASS
CREATE CAST	DROP PROCEDURE
CREATE DATABASE	DROP ROLE
CREATE DISTINCT TYPE	DROP ROUTINE
CREATE DUPLICATE	DROP ROW TYPE
CREATE EXTERNAL TABLE	DROP SECURITY
CREATE FUNCTION	DROP SEQUENCE
CREATE FUNCTION FROM	DROP SYNONYM
CREATE INDEX	DROP TABLE
CREATE OPAQUE TYPE	DROP TRIGGER
CREATE OPCLASS	DROP TYPE
CREATE PROCEDURE	DROP VIEW
CREATE PROCEDURE FROM	DROP XADATASOURCE
CREATE ROLE	DROP XADATASOURCE TYPE
CREATE ROUTINE FROM	MOVE
CREATE ROW TYPE	RENAME COLUMN
CREATE SCHEMA	RENAME DATABASE
CREATE SECURITY LABEL	RENAME INDEX
CREATE SECURITY LABEL COMPONENT	RENAME SECURITY
CREATE SECURITY POLICY	RENAME SEQUENCE
CREATE SEQUENCE	RENAME TABLE
CREATE SYNONYM	TRUNCATE
CREATE TABLE	UPDATE STATISTICS
CREATE Temporary TABLE	

For any valid DDL statement, the **mi_get_result( )** loop returns the following states of the database server:

1. An MI_DDL status indicates that the SQL statement has successfully completed.
2. In the next iteration of the **mi_get_result( )** loop, **mi_get_result( )** returns MI_NO_MORE_RESULTS.

Handling a DML Statement

The **mi_get_result( )** function returns a status of MI_DML to indicate that the current statement is a data manipulation (DML) statement that has successfully executed. SQL contains the DML statements that Table 8-7 lists.

Table 8-7. SQL Statements with an MI_DML Status

DML Statement	Purpose	Statement-Status Constant
DELETE	Remove a row from a database table	MI_DML
INSERT	Add a new row to a database table	MI_DML
UPDATE	Modify values in an existing row of a database table	MI_DML
SELECT	Fetch a row or group of rows from the database	MI_ROWS, MI_DML
EXECUTE FUNCTION	Execute a user-defined function	MI_ROWS, MI_DML

Tip: The **mi_get_result( )** function returns the MI_DML status when the current statement is the EXECUTE FUNCTION statement. This SQL statement can return rows of data and therefore is handled in the same way as the SELECT statement. However, execution of the EXECUTE PROCEDURE statement causes a statement status of MI_DDL because this SQL statement never returns rows.

When you receive the MI_DML statement status, you can use the DataBlade API functions in the following table to obtain information about the results of the current statement.

Result Information	DataBlade API Function	Additional Information
The name of the DML statement that executed as the current statement	**mi_result_command_name( )**	This function might be useful in an interactive application in which the statement sent is not determined until runtime. Use this routine only when **mi_get_result( )** reports that a DML or DDL statement has completed.
The number of rows that the current statement affected	**mi_result_row_count( )**	This function is applicable only when **mi_get_result( )** reports that a DML statement completed.

Important: If you want a count of the numbers of rows that satisfy a given query, but you do not want the data in the rows, you can run a query that

uses the COUNT aggregate more efficiently than you can run a query that returns the actual rows. For example, the following query counts the number of rows in mytable:

```
SELECT COUNT(*) FROM mytable;
```

Figure 8-12 shows a sample function, named **handle_dml( )**, that handles the MI_DML statement status that **mi_get_result( )** returns.

```
mi_integer handle_dml(conn)
   MI_CONNECTION *conn;
{
   char        *cmd;
   mi_integer   count;

   /* What kind of statement was it? */
   cmd = mi_result_command_name(conn);
   DPRINTF("trc_class", 11,
       ("Statement executed was %s", cmd));

   /* Get # rows affected by statement */
   if ( (count = mi_result_row_count(conn)) == MI_ERROR )
       {
       DPRINTF("trc_class", 11,
           ("Cannot get row count\n"));
       return( -1 );
       }
   else if ( count = 0 )
       {
       DPRINTF("trc_class", 11,
           ("No rows returned from query\n"));
       }
   else
       DPRINTF("trc_class", 11,
           ("Rows Returned\n"));

   return ( count );
}
```

Figure 8-12. Sample Function to Handle MI_DML Statement Status

The **handle_dml( )** function in Figure 8-12 on page 8-37 uses the **mi_result_command_name( )** and **mi_result_row_count( )** functions to obtain additional information about the DML statement. The function returns the number of rows affected (from **mi_result_row_count( )**) to the calling routine.

─── **Server Only** ───

The **handle_dml( )** function in Figure 8-12 assumes it is called from within a C UDR because it uses the DPRINTF statement. The DPRINTF statement is part of the DataBlade API tracing feature and available *only* to C UDRs. The first DPRINTF statement in Figure 8-12 sends the name of the current statement to a trace file when the current trace level is 11 or higher. For more information on tracing, see "Using Tracing" on page 12-28.

─── **End of Server Only** ───

─── **Client Only** ───

For the **handle_dml( )** function to execute in a client LIBMI application, it would need to replace the DPRINTF statements with a client-side output function such as

printf( ) or **fprintf( )**. The following line shows the use of the **printf( )** function to display the name of the current statement:

```
printf("Statement executed was %s", cmd);
```

──────────────────── **End of Client Only** ────────────────────

For an example of how to call **handle_dml( )**, see the MI_DML case of the **switch** statement in "Example: The get_results( ) Function" on page 8-39.

For a successful UPDATE, INSERT, and DELETE statement, the **mi_get_result( )** loop returns the following states of the database server:

1. An MI_DML status indicates that the DML statement has successfully completed.
2. In the next iteration of the **mi_get_result( )** loop, **mi_get_result( )** returns MI_NO_MORE_RESULTS.

For a successful SELECT (or EXECUTE FUNCTION) statement, the **mi_get_result( )** loop returns the following states of the database server:

1. An MI_ROWS statement status indicates that the current statement is a query that has executed successfully and whose cursor is ready for processing of query rows.
2. After all query rows are retrieved, the next iteration of the **mi_get_result( )** loop returns an MI_DML statement status to indicate that the SELECT (or EXECUTE FUNCTION) has successfully completed.
3. The next iteration of the **mi_get_result( )** loop returns the MI_NO_MORE_RESULTS status to indicate that statement processing is complete.

For more information, see "Handling Query Rows" on page 8-38.

Handling Query Rows

The **mi_get_result( )** function returns a status of MI_ROWS to indicate that a current statement is a query that has successfully executed and a cursor is open. A query can be instigated by a SELECT or an EXECUTE FUNCTION statement. The MI_ROWS statement status does *not* indicate that rows are in the cursor. If the query has not found any matching rows (the NOT FOUND condition), **mi_get_result( )** still returns MI_ROWS. To retrieve rows from the cursor, use the **mi_next_row( )** statement. If no rows exist in the cursor, **mi_next_row( )** returns a NULL-valued pointer. For more information, see "Retrieving Query Data" on page 8-39.

Handling No More Results Status

The **mi_get_result( )** function returns a status of MI_NO_MORE_RESULTS to indicate that statement processing for the current statement is complete. The function can return MI_NO_MORE_RESULTS when any of the following conditions occur for the current statement:

- When the cursor is empty after **mi_open_prepared_statement( )** has opened the cursor and before **mi_fetch_statement( )** has fetched rows into this cursor
- After **mi_next_row( )** has retrieved the last row from the cursor
- When the query is complete, after **mi_query_finish( )** or **mi_query_interrupt( )** has executed
- When any non-query statement is complete: after **mi_get_result( )** has returned the MI_DML or MI_DDL statement status

Tip: When a SELECT or FETCH statement encounters NOT FOUND (or END OF DATA), the database server sets **SQLSTATE** to "02000" (class = "02"). However, the NOT FOUND condition does not generate a database server exception.

Example: The get_results( ) Function

The following user function, **get_results( )**, demonstrates the **mi_get_result( )** row-retrieval loop, controlled with the **mi_get_result( )** function. It also demonstrates the use of the **mi_result_command_name( )** function to get the name of the current statement and the **mi_result_row_count( )** function to get the number of rows affected by this statement.

```
/*
 * FUNCTION: get_results( )
 * PURPOSE: Get results of current statement.
 *          Obtain the kind of statement and the number of
 *              rows affected.
 *          Return the number of rows affected.
 *
 * CALLED BY: send_statement( ), see page
```

8-10. */ #include "mi.h" mi_integer get_results(MI_CONNECTION *conn) { mi_integer count; mi_integer result; char cmd[25]; while ((result = mi_get_result(conn)) != MI_NO_MORE_RESULTS) { switch(result) { case MI_ERROR: mi_db_error_raise(conn, MI_EXCEPTION, "Could not get statement results (mi_get_result)\n"); case MI_DDL: count = 0; break; case MI_DML: count = handle_dml(conn); break; case MI_ROWS: count = get_data(conn); break; default: mi_db_error_raise(conn, MI_EXCEPTION, "Unknown statement results (mi_get_result)\n"); } /* end switch */ } /* end while */ return (count); }

When a query returns rows of data, the **mi_get_result( )** loop in **get_results( )** executes three times:

1. The first iteration of the **mi_get_result( )** loop returns MI_ROWS to indicate that the query has successfully opened a cursor.

   The **get_results( )** function executes the MI_ROWS case of the **switch** statement. This function then calls another user function, **get_data( )**, to iterate over all query rows. For the implementation of the **get_data( )** function, see "Example: The get_data( ) Function" on page 8-54.

2. The second iteration of the **mi_get_result( )** loop returns MI_DML to indicate that the cursor processing has completed and the query has successfully completed.

   The **get_data( )** function has handled the rows in the cursor so no more query rows remain to be processed. The **get_results( )** function executes the MI_DML case of the switch statement, which calls the **handle_dml( )** function to obtain the name and number of statements from the current statement. For the implementation of the **handle_dml( )** function, see Figure 8-12 on page 8-37.

3. The third iteration of the **mi_get_result( )** loop returns MI_NO_MORE_RESULTS to indicate that processing for the query is complete.

   The MI_NO_MORE_RESULTS value from **mi_get_result( )** causes the **mi_get_result( )** loop to terminate.

Retrieving Query Data

When **mi_get_result( )** returns the MI_ROWS statement status, the query has executed and a cursor is open, as follows:

- For SQL statements sent with **mi_exec( )** or **mi_exec_prepared_statement( )**, the database server opens an implicit cursor. This cursor contains the retrieved rows, with the database server controlling the rows that are fetched.
- For SQL statements sent with **mi_open_prepared_statement( )**, the database server opens an explicit cursor. This cursor is empty, with the **mi_fetch_statement( )** controlling the rows that are fetched.

The DataBlade API module receives the query data on a row-by-row basis. To handle the rows that the current statement has retrieved, the DataBlade API creates the following data type structures:

- The *row descriptor* is the data-description portion, which contains information such as row size and column data types.
- The *row structure* is the data portion, which holds one row of data that the query returns.

A one-to-one correspondence occurs between row descriptors and rows. Each row descriptor has an associated row structure.

Server Only

In a C UDR, the row structure and row descriptor are part of the same data type structure. The row structure is just a data buffer in the row descriptor that holds copies of the column values of a row.

End of Server Only

The row descriptor and row structure are valid until the next row is fetched. A row descriptor might need to change on a row-to-row basis for jagged rows. (For more information, see "Obtaining Jagged Rows" on page 8-51.) A row structure holds each row, one row at a time.

To retrieve the row of query data:
1. Get a copy of the row descriptor for a query row.
2. Get the number of columns from the row descriptor.
3. Retrieve query rows, one row at a time.
4. For every query row, get the value of any desired column.

Obtaining Row Information

A row descriptor (MI_ROW_DESC) contains information about the columns in a row. For example, the row descriptor for the following query would contain two columns, **order_num** and **order_date**:

```
SELECT order_num, order_date FROM orders
WHERE ship_date > "07/15/98";
```

To obtain a row descriptor for a row, you can use one of the DataBlade API functions in the following table.

Use	DataBlade API Function	Description
For rows with the same type and size:	**mi_get_row_desc_without_row( )**	Returns a row descriptor for the current statement
	mi_get_statement_row_desc( )	Returns a row descriptor for a prepared statement

Use	DataBlade API Function	Description
For rows of different types or sizes (jagged rows):	**mi_get_row_desc( )**	Returns a row descriptor associated with a particular row structure
	mi_get_row_desc_from_type_desc( )	Returns a row descriptor based on a type descriptor

These functions allocate the memory for the row descriptor that they allocate.

To obtain a row descriptor for the query rows in an implicit or explicit cursor, use the **mi_get_row_desc_without_row( )** function. To free this row descriptor, complete the query. For more information, see "Completing Execution" on page 8-57.

Obtaining Column Information

Once you have a row descriptor for the row, you can obtain information about the columns with the row-descriptor access functions, which Table 5-3 on page 5-30 shows. For each column in an SQL statement, you can obtain information about the column (such as its data type) from the row descriptor.

You can use the **mi_column_count( )** function to determine how many columns are in the row. The number of columns in the row descriptor is the number of columns that the query has retrieved. Use this value to control the number of times to call the **mi_value( )** or **mi_value_by_name( )** function. Each call to **mi_value( )** or **mi_value_by_name( )** passes back one column value from the row structure to the DataBlade API module. For more information, see "Obtaining Column Values" on page 8-42.

Retrieving Rows

After a query executes, a cursor holds the query rows. The **mi_next_row( )** function takes the following actions to obtain the rows from a cursor:

- Obtains access to the current row
- Executes the **mi_next_row( )** function in a loop that iterates for each query row

For a sample function that shows one way to use **mi_next_row( )** to retrieve query rows, see "Example: The get_data( ) Function" on page 8-54.

Accessing the Current Row

The **mi_next_row( )** function accesses rows in the cursor that is associated with the current statement. Because a current statement is associated with a connection, you must pass a connection descriptor into **mi_next_row( )** to identify the cursor to access. From this cursor, **mi_next_row( )** obtains the *current row*. The *current row* is the row in the cursor that the cursor position identifies. Each time **mi_next_row( )** retrieves a row, this cursor position moves by one. One cursor per connection is current and within this cursor, only one row at a time is current.

The **mi_next_row( )** function returns the current row in an implicit *row structure* (MI_ROW structure). The row structure stores the column values of a single query row. The contents of this row structure are valid until **mi_next_row( )** returns the next row. You can obtain column values from the row structure with the **mi_value( )** or **mi_value_by_name( )** function. For more information, see "Obtaining Column Values" on page 8-42.

This implicit row structure is freed when the query is completed, which can occur in any of the following ways:

- When **mi_next_row( )** returns the NULL-valued pointer to indicate no more rows exist in the cursor
- When the **mi_query_finish( )** function executes
- When the connection is closed

Executing the mi_next_row( ) Loop

The **mi_next_row( )** function is usually the middle loop of row-retrieval code. In the **mi_next_row( )** loop, each call to **mi_next_row( )** returns one query row from the cursor that is associated with the current statement. This query row is the current row only until the next iteration of the loop, when **mi_next_row( )** retrieves another row from the cursor. This loop terminates when **mi_next_row( )** returns a NULL-valued pointer and its *error* argument is zero (0). These conditions indicate either that no more rows exist in the cursor or that the cursor is empty. Think of the **mi_next_row( )** loop as an iteration over the matching rows of the query.

The contents of a row structure become invalid as soon as you fetch a new row into it with **mi_next_row( )**. If you want to save the row values that you obtain with **mi_value( )** or **mi_value_by_name( )**, copy the values that these functions pass back before the next call to **mi_next_row( )**.

Tip: If your DataBlade API module requires simultaneous access to several rows at a time, you can use a save set to hold rows. Save sets are useful for comparing or processing multiple rows. For more information, see "Using Save Sets" on page 8-59.

The **mi_next_row( )** function allocates memory for the row structure that it returns. To free this row structure, you must complete the query. For more information, see "Completing Execution" on page 8-57.

As long as rows remain to be retrieved from the cursor, the **mi_get_result( )** function returns a statement status of MI_ROWS. Therefore, you cannot exit the **mi_get_result( )** loop until one of the following actions occurs:

- The **mi_next_row( )** loop continues until no more rows exist in the cursor. That is, **mi_next_row( )** returns a NULL-valued pointer.
- You terminate the **mi_get_result( )** loop prematurely with a call to **mi_query_finish( )** or **mi_query_interrupt( )**.

Obtaining Column Values

When the **mi_next_row( )** function retrieves a query row from the cursor, it returns this row in a row structure. The DataBlade API provides the following functions to get actual column values from a row structure.

DataBlade API Function	Obtaining a Column Value
mi_value( )	Obtains a column value, as identified by its column identifier, from a row structure
mi_value_by_name( )	Obtains a column value, as identified by its column name, from a row structure

Use **mi_value( )** or **mi_value_by_name( )** to retrieve columns from the current row as follows:

- Execute the **mi_value( )** or **mi_value_by_name( )** function in a loop that iterates for each desired column value.
- Interpret the value status that these functions return to correctly access the column value.

The final section, "Example: The get_data( ) Function" on page 8-54, contains sample code that shows one way to use **mi_value( )** to get column values.

Executing the Column-Value Loop

The **mi_value( )** or **mi_value_by_name( )** function is usually called in the innermost loop of row-retrieval code. In the column-value loop, **mi_value( )** or **mi_value_by_name( )** retrieves a column value from the current row. This loop terminates when a value is retrieved for every column in the row (or every column your DataBlade API module needs to access). You can obtain the number of columns in a row with the **mi_column_count( )** function.

Accessing the Columns

The **mi_value( )** and **mi_value_by_name( )** functions access the row structure for the current row. The current row is in the cursor that is associated with the current statement. Because a current statement is associated with a connection, you must pass a connection descriptor into **mi_value( )** or **mi_value_by_name( )** to identify the row to access.

These functions pass back the column value as an **MI_DATUM** value. The format of this value depends on whether the control mode for the query data is text or binary representation. Each of the DataBlade API statement-execution functions indicates the control mode for query data. For more information, see "Control Modes for Query Data" on page 8-8 and "Determining Control Mode for Query Data" on page 8-30.

To obtain this column value, your DataBlade API module must perform the following steps:
- Declare a value buffer to hold the column value that **mi_value( )** or **mi_value_by_name( )** passes back.
- Obtain the column value from the value buffer, based on the value status that **mi_value( )** or **mi_value_by_name( )** returns.

Passing In the Value Buffer: To obtain the column value, you must pass in a pointer to a *value buffer* as an argument to **mi_value( )** or **mi_value_by_name( )**. The value buffer is the place that these functions put the column value that they retrieve from the current row. Both **mi_value( )** and **mi_value_by_name( )** represent a column value as a pointer to an **MI_DATUM** structure.

You can declare the value buffer in either of the following ways:
- If you know the data type of the column value, declare the value buffer of this data type.

  Declare the value buffer as *a pointer to* the column data type, regardless of whether the data type is passed by reference or by value.
- If you do *not* know the data type of the column value, declare the value buffer with the **MI_DATUM** data type.

  Your code can dynamically determine column type with the **mi_column_type_id( )** or **mi_column_typedesc( )** function. You can then convert (or cast) the **MI_DATUM** value to the data type that you need.

The **mi_value( )** and **mi_value_by_name( )** functions allocate memory for the value buffer. However, this memory is only valid until a new SQL statement executes or until the query completes. In addition, the DataBlade API might overwrite the value-buffer data in any of the following cases:

- The **mi_next_row( )** function is called on the same connection.
- A call to **mi_row_create( )** uses the row descriptor.
- The **mi_row_free( )** function is called on the row structure.
- The **mi_row_desc_free( )** function is called on the row descriptor.

If you need to save the value-buffer data for later use, you must create your own copy of the data in the value buffer.

Tip: If your DataBlade API module requires simultaneous access to several rows at a time, you can use a save set to hold rows. Save sets are useful for comparing or processing multiple rows. For more information, see "Using Save Sets" on page 8-59.

Interpreting Column-Value Status: The **mi_value( )** and **mi_value_by_name( )** functions return a value status, which identifies how to interpret the column value that these functions pass back. The following table shows the kinds of column values that these functions can identify.

Type of Column Value	Value-Status Constant	More Information
A built-in, opaque, or distinct data type	MI_NORMAL_VALUE	"Obtaining Normal Values" on page 8-44
An SQL NULL value	MI_NULL_VALUE	"Obtaining NULL Values" on page 8-49
A row type	MI_ROW_VALUE	"Obtaining Row Values" on page 8-50
A collection	MI_COLLECTION_VALUE	"Obtaining Collection Values" on page 8-52

You can use a **switch** statement based on these value-status constants to determine how to handle the column value.

Obtaining Normal Values

The **mi_value( )** and **mi_value_by_name( )** functions return the MI_NORMAL_VALUE value status for a column with any data type *other than* a row type or collection. Therefore, these functions return MI_NORMAL_VALUE for columns that have a built-in data type, smart large object, opaque type, or distinct type.

When the **mi_value( )** or **mi_value_by_name( )** function returns MI_NORMAL_VALUE, the contents of the **MI_DATUM** structure that holds the column value depends on whether the control mode for the query data is text or binary representation, as follows:

- Text representation: the **MI_DATUM** structure contains a pointer to a null-terminated string, which has the text representation of the column value.
- Binary representation: the **MI_DATUM** structure contains a value whose interpretation depends on the passing mechanism used, as follows:

──────── **Server Only** ────────

– When **mi_value( )** or **mi_value_by_name( )** passes back a column value to a C UDR, it can pass the value by reference or by value, depending on the data type of the column value. If the function passes back the value by value, the **MI_DATUM** structure contains the value. If the function passes back the value by reference, the **MI_DATUM** structure contains a pointer to the value.

───────────────── **End of Server Only** ─────────────────

───────────────── **Client Only** ─────────────────

– When **mi_value( )** or **mi_value_by_name( )** passes back a column value to a client LIBMI application, it *always* passes the value by reference. Even for values that you can pass by value in a C UDR (such as an INTEGER value), these functions return the column value by reference. The **MI_DATUM** structure contains a pointer to the value.

───────────────── **End of Client Only** ─────────────────

For a list of the text and binary representations of built-in, opaque, and distinct data types, see Table 8-3 on page 8-8. For more information on the passing mechanism for an **MI_DATUM** value, see "Contents of an MI_DATUM Structure" on page 2-33.

Important: The difference in behavior of **mi_value( )** and **mi_value_by_name( )** between C UDRs and client LIBMI applications means that row-retrieval code is not completely portable between these two types of DataBlade API modules. When you move your DataBlade API code from one of these uses to another, you must change the row-retrieval code to use the appropriate passing mechanism for column values that **mi_value( )** or **mi_value_by_name( )** returns.

Column Values Passed Back to a C UDR (Server): Within a C UDR, the value buffer that **mi_value( )** or **mi_value_by_name( )** fills can contain either of the following values:

- For data types that are passed by value, the value buffer contains the actual column value.
- For data types that are passed by reference, the value buffer contains a pointer to this column value.

Tip: The value buffer also contains a pointer to the column value if the return status of **mi_value( )** or **mi_value_by_name( )** is MI_ROW_VALUE or MI_COLLECTION_VALUE. For more information, see "Obtaining Row Values" on page 8-50 and "Obtaining Collection Values" on page 8-52.

Therefore, within your C UDR, you cannot assume what the **MI_DATUM** value contains without checking its data type (or length).

───────────────── **Windows Only** ─────────────────

If the **mi_value( )** or **mi_value_by_name( )** function passes back a value *smaller* than the size of an **MI_DATUM** structure, the DataBlade API cast promotes the smaller value to the size of **MI_DATUM**. (For more information, see "MI_DATUM in a C UDR (Server)" on page 2-33). If you now want to pass a pointer in the **MI_DATUM** structure, you might have problems on Windows if you passed the address of the **MI_DATUM** value, as the following pseudocode shows:

```
MI_DATUM datum;
mi_integer length;
mi_char bool;
mi_short small;
mi_int large;
void *pointer;

switch ( mi_value( ..., &datum, &length ) )
   {
   ....
   } /* end switch */
/* Assume that 'datum' contains a BOOLEAN value
 * (which uses only one byte of the MI_DATUM storage space).
 * Pass the address of the actual data to another function.
 * YOU CANNOT ALWAYS DO THIS!
 *       my_func( &datum, length );
 * This address might point to the wrong byte! */
```

The preceding code fragment works if **datum** always contains a pointer to a
column value or contains data the size of **MI_DATUM**. It might *not* work on some
computer architectures, however, for data that is smaller than **MI_DATUM** (such
as **mi_boolean**).

To convert the **MI_DATUM** value into a pointer to the data value, you must be
sure that the address points to the starting position of the cast-promoted data. The
following code fragment determines what the **MI_DATUM** value in **datum**
contains and then correctly copies the value and obtains its address, based on the
length of **datum**:

```
MI_ROW_DESC *row_desc;
MI_ROW *row;
MI_DATUM datum;
mi_integer length;
mi_boolean *bool;
mi_smallint *small_int;
mi_integer *full_int;
mi_date *date_val;
mi_string *col_type_name;
void *ptr_to_value;
...
switch ( mi_value(row, i, col_id, &datum, &length) )
   {
   ...
   case MI_NORMAL_VALUE:
      col_type_name =
         mi_type_typename(
            mi_column_typedesc(row_desc, i));

/* To obtain the datum value and its address, first check
 * if the value is passed by value. If not, assume that
 * the value is passed by reference.
 */
     switch( length )
        {
     /*  Case 1: Assume that a length of one byte means
      *          that 'datum' contains a BOOLEAN value.
      */
        case 1:
           bool = (mi_boolean) datum;
           ptr_to_value = &bool;
           break;

     /*  Case 2: Assume that a length of two bytes means
      *          that 'datum' contains a SMALLINT value
      */
        case 2:
```

```
                       small_int = (mi_smallint) datum;
                       ptr_to_value = &small_int;
                       break;

            /*  Case 4: Assume that a length of four bytes means
             *          that 'datum' contains an INTEGER or DATE value
             */
               case 4:
                  if ( stcopy(col_type_name, "date") == 0 )
                     {
                     date_val = (mi_date) datum;
                     ptr_to_value = &date_val;
                     }
                  else /* data type is INTEGER */
                     {
                     full_int = (mi_integer) datum;
                     ptr_to_value = &full_int;
                     }
                  break;

            /*  Default Case: Assume that any for any other lengths,
             *                'datum' contains a pointer to the value.
             */
               default:
                  ptr_to_value = &datum;
                  break;
      } /* end switch */

my_func( ptr_to_value );
```

The preceding code fragment handles only built-in data types that are passed by value. It was *not* written to handle all possible user-defined types (such as small fixed-length opaque types) because these do not have unique lengths.

────────────────────────── **End of Windows Only** ──────────────────────────

In a C UDR, if the data type of the column value can fit into an **MI_DATUM** structure, the value buffer contains the actual column value. If you know the data types of the column values, the value buffers you declare to hold the column values must be declared as *pointers to* their data type. For example, the code fragment declares the value buffer that holds a SMALLINT value can be declared as follows:

```
mi_integer *small_int_ptr;
```

After the call to **mi_value( )**, the C UDR must cast the contents of the value buffer from a pointer variable (as **small_int_ptr** is declared) to the actual data type. For the SMALLINT value, the code can perform the following cast to create a copy of the column value:

```
small_int = (mi_smallint) small_int_ptr;
```

This cast is necessary *only* for column values whose data types are passed by value because the **MI_DATUM** structure contains the actual column value, not a pointer to the value.

You can use the **mi_type_byvalue( )** function to determine the passing mechanism of the column value that **mi_value( )** passes back, as the following code fragment shows:

```
row_desc = mi_get_row_desc_without_row(conn);
...
switch ( mi_value(row, i, col_id, &datum, &length) )
   {
```

```
...
case MI_NORMAL_VALUE:

    if ( mi_type_byname(mi_column_typedesc(row_desc, i))
        == MI_TRUE )
        {
        /* handle pass-by-value data types */;
```

The **mi_type_byvalue( )** function helps to determine if a one-, two-, or four-byte
value is actually passed by value. You can use this function to determine the
passing mechanism of a fixed-length opaque data type.

Column Values Passed Back to a Client LIBMI Application: The **mi_value( )**
and **mi_value_by_name( )** functions pass back by reference column values for *all*
data types; therefore, the returned **MI_DATUM** structure *always* contains a pointer
to the actual value, never the value itself. Even column values that can fit into an
MI_DATUM structure are passed by reference. For example, a SMALLINT value
could have the same value-buffer declaration as it would in a C UDR, as follows:

```
mi_integer *small_int_ptr;
```

Unlike a C UDR, however, the column value in the value buffer does not require a
cast to create a copy:

```
small_int = *small_int_ptr;
```

Accessing Smart Large Objects: In a database, smart large objects are in columns
with the data type CLOB or BLOB. A smart-large-object column contains an *LO
handle* that describes the smart large object, including the location of its data in an
sbspace. This LO handle does *not* contain the actual smart-large-object data.

When a query retrieves a smart large object (a BLOB or CLOB column), the
mi_value( ) and **mi_value_by_name( )** functions return the MI_NORMAL_VALUE
value status. For a BLOB or CLOB column, the **MI_DATUM** structure that these
functions pass back contains the LO handle for the smart large object. The control
mode of the query data determines whether this LO handle is in text or binary
representation, as follows.

Query Control Mode	Contents of Value Buffer
Text representation	Character string that contains the hexadecimal dump of the LO-handle structure
Binary representation	Pointer to an LO-handle structure (**MI_LO_HANDLE** *)

When query data is in binary representation, the **mi_value( )** and
mi_value_by_name( ) functions pass back the LO handle by reference. Regardless
of whether you obtain a smart large object in a C UDR or a client LIBMI
application, the **MI_DATUM** structure that these functions pass back contains a
pointer to an LO handle (**MI_LO_HANDLE** *).

To make a copy of the LO handle within your DataBlade API module, you can
copy the contents of the value buffer, as follows:

```
MI_LO_HANDLE *blob_col, my_LO_hdl;
...
switch ( mi_value(row, i, col_id, &blob_col, &length) )
    {
```

```
    ...
    case MI_NORMAL_VALUE:

        my_LO_hdl = *blob_col;
```

To obtain the smart-large-object data, use the binary representation of the LO handle with the functions of the smart-large-object interface. The smart-large-object interface allows you to access smart-large-object data through its LO handle. You access the smart-large-object data with read, write, and seek operations similar to an operating-system file.

The following code fragment implements the **get_smart_large_object( )** function, which reads smart-large-object data in 4,000-byte chunks:

```
#define BUFSIZE 4000;

mi_integer get_smart_large_object(conn, LO_hndl)
    MI_CONNECTION *conn;
    MI_LO_HANDLE *LO_hndl;
{
    MI_LO_FD LO_fd;
    mi_char read_buf[BUFSIZE];

/* Open the selected smart large object */
LO_fd = mi_lo_open(conn, LO_hndl, MI_LO_RDONLY);
if ( LO_fd == MI_ERROR )
    /* handle error */
    return (-1);
else
    {
    while ( mi_lo_read(conn, LO_fd, read_buf, BUFSIZE)
        != MI_ERROR )
        {
        /* perform processing on smart-large-object data */
        ...
        }
    mi_lo_close(conn, LO_fd);
    return ( 0 );
    }
}
```

For a description of the smart-large-object interface, see Chapter 6, "Using Smart Large Objects," on page 6-1.

Obtaining NULL Values

The **mi_value( )** and **mi_value_by_name( )** functions return the MI_NULL_VALUE value status for a column that contains the SQL NULL value. These functions return MI_NULL_VALUE for columns of *any* data type. When the **mi_value( )** or **mi_value_by_name( )** function returns MI_NULL_VALUE, the contents of the **MI_DATUM** structure that these functions pass back depends on whether the control mode for the query data is text or binary representation, as the following table shows.

Control Mode	Contents of Value Buffer (From mi_value( ) or mi_value_by_name( ))
Text representation	No valid value
Binary representation	The internal representation of the SQL NULL value for the data type

Obtaining Row Values

The **mi_value( )** and **mi_value_by_name( )** functions return the MI_ROW_VALUE value status for a column that meets either of the following conditions:

- The column has a row type (a named or unnamed row type) as its data type.
- The item being selected is a correlation variable that represents an entire row.

  A correlation variable is used in a select list when jagged rows are selected from a supertable in an inheritance hierarchy, as in the query:

  ```
  SELECT p FROM parent p;
  ```

When the **mi_value( )** or **mi_value_by_name( )** function returns MI_ROW_VALUE, the **MI_DATUM** structure that these functions pass back contains a pointer to the row structure, regardless of whether the query data is in binary or text representation. You can extract the individual values from the row structure by passing the returned MI_ROW pointer to **mi_value( )** or **mi_value_by_name( )** for each value you need to retrieve.

Obtaining Row Types: The **mi_value( )** and **mi_value_by_name( )** functions can return the MI_ROW_VALUE value status for a column with a row data type: unnamed or named. The contents of the **MI_DATUM** structure that these functions pass back is a pointer to a row structure that contains the fields of the row type. The format of the field values depends on whether the control mode for the query data is text or binary representation, as the following table shows.

Control Mode	Contents of Fields Within Row Structure
Text representation	Null-terminated strings
Binary representation	Internal formats of field values

For a list of the text and binary representations of data types, see Table 8-3 on page 8-8.

You can extract the individual field value from the row type by passing the returned MI_ROW pointer to **mi_value( )** or **mi_value_by_name( )** for each field value you need to retrieve.

The **get_data( )** function calls the **get_row_data( )** function for an **mi_value( )** return value of MI_ROW_VALUE (see the example on page 8-54). This function takes the pointer to a row structure as an argument and uses **mi_value( )** on it to obtain field values in text representation.

```
mi_integer get_row_data(row)
   MI_ROW *row;
{
   mi_integer numflds, fldlen;
   MI_ROW_DESC *rowdesc;
   mi_integer i;
   char *fldname, *fldval;
   mi_boolean is_nested_row;

/* Get row descriptor */
   rowdesc = mi_get_row_desc(row);

/* Get number of fields in row type */
   numflds = mi_column_count(rowdesc);

/* Display the field names of the row type */
   for ( i=0; i < numflds; i++ )
      {
```

```
      fldname = mi_column_name(rowdesc, i);
      DPRINTF("trc_class, 11, ("%s\t", fldname));
      }
   DPRINTF("trc_class", 11, ("\n"));

/* Get field values for each field of row type */
   for ( i=0, i < numflds; i++ )
      {
      is_nested_row = MI_FALSE;
      switch( mi_value(row, i, &fldval, &fldlen) )
         {
         case MI_ERROR:
            ...

         case MI_NULL_VALUE:
            fldval = "NULL";
            break;

         case MI_NORMAL_VALUE:
            break;

         case MI_ROW_VALUE:
         /* have nested row type - make recursive call */
            is_nested_row = MI_TRUE;
            get_row_data((MI_ROW *)fldval);
            break;

         default:
            ...

         }
      if ( is_nested_row == MI_FALSE )
         DPRINTF("trc_class", 11, ("%s\t", fldval));
      }
   return (0);
}
```

Obtaining Jagged Rows: When all the rows that a query retrieves are *not* the same type and length, the rows are called *jagged rows*. Jagged rows occur as a result of a query that uses the following syntax to request all the rows in a supertable and all its subtables:

```
SELECT correlation_variable
FROM table_name correlation_variable;
```

In the preceding query, *table_name* represents a supertable in an inheritance hierarchy. Suppose you create the following schema in which the table **parent** has one column, **child** has two columns, and **grandchild** has three columns:

```
CREATE TABLE parent OF TYPE parent_t (num1 INTEGER);
INSERT INTO parent VALUES (10);

CREATE TABLE child OF TYPE child_t (num2 SMALLFLOAT)
   UNDER parent;
INSERT INTO child VALUES (20, 3.5);

CREATE TABLE grandchild OF TYPE grandchild_t (name TEXT)
   UNDER child;
INSERT INTO grandchild VALUES (30, 7.8, 'gundrun');
```

The following SELECT statement queries the **parent** supertable:

```
SELECT p FROM parent p;
```

This query returns the following three jagged rows:

p (parent_t)		
num1		
10		
p (child_t)		
num1	**num2**	
20	3.5E+00	
p (grandchild_t)		
num1	**num2**	**name**
30	7.8E+00	gundrun

The DataBlade API indicates that a query returned a jagged row as follows:

- The **mi_value( )** or **mi_value_by_name( )** function returns a value status of MI_ROW_VALUE.
- The contents of the **MI_DATUM** structure that holds the retrieved column is a pointer to a row structure.

The format of the columns depends on whether the control mode for the query data is text or binary representation, as the following table shows.

Control Mode	Contents of Elements within Row Structure
Text representation	Null-terminated strings
Binary representation	Internal formats of column values

For a list of the text and binary representations of data types, see Table 8-3 on page 8-8.

To retrieve jagged rows:

1. Use the **mi_get_row_desc( )** function to get a row descriptor for each row structure that **mi_value( )** or **mi_value_by_name( )** obtains.
2. Use the **mi_column_count( )** function with the row descriptor to get a column count for each row that **mi_next_row( )** retrieves.
3. Retrieve the individual components of the row within an inner column-value loop.

Obtaining Collection Values

For a collection, the value that the **mi_value( )** and **mi_value_by_name( )** functions return depends on whether the control mode for the query data is text or binary representation, as the following table shows.

Return Value	Control Mode	Contents of Value Buffer
MI_NORMAL_VALUE	Text representation	Null-terminated string that contains the text representation of the collection
MI_COLLECTION_VALUE	Binary representation	A pointer to a collection structure (**MI_COLLECTION**)

In a DataBlade API module, a collection can be created in either of the following ways:

- A column has a collection type (SET, LIST, or MULTISET) as its data type.
- An item being selected is a collection subquery, which represents a collection.

A Collection in Text Representation: When the control mode of the query data is text representation, the **mi_value( )** or **mi_value_by_name( )** function returns a value status of MI_NORMAL_VALUE for a collection column. The value buffer contains the text representation of the column.

For example, suppose that a query selects the **set_col** column, which is defined as Figure 8-13 shows.

```
CREATE TABLE table1
(
....
set_col SET(INTEGER NOT NULL),
...
)
```

Figure 8-13. A Sample Collection Column

If the **set_col** column contains a SET collection with the values of 3, 5, and 7, the value buffer contains the following string after **mi_value( )** or **mi_value_by_name( )** executes:

```
"SET{3          ,5          ,7          }"
```

For a description of collection text representation, see "Collection Text Representation" on page 5-2.

A Collection in Binary Representation: When the control mode of the query data is in binary representation, the **mi_value( )** or **mi_value_by_name( )** function returns a value status of MI_COLLECTION_VALUE for a collection column. The value buffer contains a pointer to the collection structure for the collection. You can extract the individual elements from the collection structure with the DataBlade API collection functions, as follows:

- The **mi_collection_open( )** function opens the collection that the collection structure describes.
- The **mi_collection_fetch( )** functions fetches the elements from the collection. The element that **mi_collection_fetch( )** obtains is in its binary representation.
- The **mi_collection_close( )** function closes the collection that the collection structure describes.

For more information on the use of the DataBlade API collection functions, see "Collections" on page 5-2.

For the collection column that Figure 8-13 on page 8-53 defines, the following code fragment handles the MI_COLLECTION_VALUE value status that **mi_value( )** or **mi_value_by_name( )** returns for a collection column in binary representation:

```
switch( mi_value(row, i, &colval, &collen) )
   {
   ...
   case MI_COLLECTION_VALUE:
      if ( (colldesc = mi_collection_open(conn,
           (MI_COLLECTION *)colval) != NULL )
         {
         while ( mi_collection_fetch(conn, colldesc,
              MI_CURSOR_NEXT, 0, (MI_DATUM *)&elmtval,
              &elmtlen) != MI_END_OF_DATA )
```

```
                                     {
                                     int_val = (mi_integer)elmtval;
                                     DPRINTF("trc_class", 11,
                                        ("Element value=%d\n", int_val));
                                     }
                              }
                     break;
```

Example: The get_data( ) Function

The **get_data( )** function retrieves data from a query that **mi_exec( )** sends to the database server. This example makes the following assumptions:

- The query data is in text representation because the original call to **mi_exec( )** in the **send_statement( )** function specifies the MI_QUERY_NORMAL control flag (see the example on page 8-10).

- All the rows are of the same type and therefore share the same row descriptor (that is, no jagged rows).

The code for the **get_data( )** function follows:

```
/*
 * FUNCTION: get_data( )
 * PURPOSE: Gets rows that a query returns.
 *
 * CALLED BY: get_results( ) (See page 8-39.)
 */
#include "mi.h"

mi_integer get_data( MI_CONNECTION *conn )
 {
    MI_ROW          *row = NULL;
    MI_ROW_DESC     *rowdesc;
    mi_integer      error;
    mi_integer      numcols;
    mi_integer      i;
    mi_string       *colname;
    mi_integer      collen;
    mi_string       *colval;
    mi_integer      is_nested_row;

    /* Get the row descriptor for the current statement */
    rowdesc = mi_get_row_desc_without_row(conn);

  /* Get the number of columns in the row */

     numcols = mi_column_count(rowdesc);

  /* Obtain the column names from the row desriptor */
    i = 0;
    while( i < numcols )
       {
       colname = mi_column_name(rowdesc, i);
       DPRINTF("trc_class", 11, ("  %s\t", colname));

        i++;
       }
    DPRINTF("trc_class", 11,("\n\n"));

  /* For each retrieved row: */
    while ( NULL != (row = mi_next_row(conn, &error)) )
       {
       /* For each column */
       for ( i = 0; i < numcols; i++ )
          {
          is_nested_row = MI_FALSE;
```

```
             /* Initialize column value and length */
            colval = NULL;
            collen = 0;

             /* Put the column value in colval */
            switch( mi_value(row, i, &colval, &collen) )
                {
              case MI_ERROR:
                  mi_db_error_raise(conn, MI_EXCEPTION,
               "\nCannot get column value (mi_value)\n" );

               case MI_NULL_VALUE:
                  colval = "NULL";
                  break;

               case MI_NORMAL_VALUE:
                  break;

               case MI_ROW_VALUE:
                  is_nested_row = MI_TRUE;
                  get_rowtype_data((MI_ROW *)colval);
                  break;

               default:
                  mi_db_error_raise(conn, MI_EXCEPTION,
                     "\nUnknown value (mi_value)\n" );
                  return( -1 );
                }     /* end switch */

          if ( is_nested_row )
             {
             /* process row type */
             }
          else
             {
             /* Print the column value */
             DPRINTF("trc_class", 11, ("  %s\t", colval));
             }

           } /* end for */

     DPRINTF("trc_class", 11, ("\n"));
    } /* end while */

  if ( MI_ERROR == error )
    {
    DPRINTF("trc_class", 11, ("\nReached last row\n"));
    }

  DPRINTF("trc_class", 11, ("\n"));

  return(1);
}
```

The **get_data( )** function calls **mi_get_row_desc_without_row( )** to obtain the row descriptor and **mi_column_count( )** to obtain the number of columns. It then calls **mi_column_name( )** in a **for** loop to obtain and print the names of the columns in the row descriptor.

Server Only

The **get_data( )** function assumes it is called from within a C UDR. The function uses the DPRINTF statement, which is part of the DataBlade API tracing feature and is available *only* to C UDRs. The first DPRINTF statement sends the name of each retrieved column to a trace file when the trace level is 11 or higher. Another

DPRINTF statement sends the column value to the trace file. For more information on tracing, see "Using Tracing" on page 12-28.

─────────────── **End of Server Only** ───────────────

─────────────── **Client Only** ───────────────

For the **get_data( )** function to execute in a client LIBMI application, it would need to replace the DPRINTF statement with a client-side output function such as **printf( )** or **fprintf( )**. The following code fragment uses the **fprintf( )** function to display the names of retrieved columns:

```
while( i < numcols )
  {
    colname = mi_column_name(rowdesc, i);
    fprintf(stderr, "%s\t", colname);

    i++;
  }
fprintf(stderr, "\n\n");
```

All occurrences of DPRINTF would need to be replaced by appropriate client-side output functions.

─────────────── **End of Client Only** ───────────────

In the outer loop, **mi_next_row( )** obtains every row, and in the inner loop, **mi_value( )** obtains every value in the row. The pointer returned in the value buffer is not valid after the next call to **mi_value( )**. If the data were needed for later use, you would need to copy the data in the value buffer into a previously defined variable.

The **get_data( )** function retrieves column data that is in text representation. The return values of the **mi_value( )** function handle the text representations as follows:

- For the MI_NORMAL_VALUE return value, **get_data( )** breaks out of the **switch** statement.

  It does not need to perform any special handling based on column data type because values for all data types have the same data type: they are all null-terminated strings. Therefore, the **colval** variable that is passed into **mi_value( )** is declared as a pointer to an **mi_string**. After **mi_value( )** completes, **colval** points to the null-terminated string for the column value.

- For the MI_NULL_VALUE return value, **get_data( )** assigns the string "NULL" as the column value.

  The **mi_value( )** function does not assign a null-terminated string to a column value when the column contains the SQL NULL value. Therefore, **get_data( )** explicitly sets the column value to hold the null-terminated string: "NULL".

- For the MI_ROW_VALUE return value, **get_data( )** calls the **get_row_data( )** function to handle the row structure.

  The **get_row_data( )** function obtains values from the row structure in their text representation. For more information, see "Obtaining Row Values" on page 8-50.

- The **get_data( )** function does *not* handle the MI_COLLECTION_VALUE return status.

  The **mi_value( )** function returns the MI_NORMAL_VALUE value status for a collection column when the query is in text representation. For sample code that

handles a collection column when query data is in binary mode, see "A Collection in Binary Representation" on page 8-53.

Completing Execution

The DataBlade API provides the following functions to complete execution of the current statement.

DataBlade API Function	Statement Completion
mi_query_finish( )	Finishes processing any remaining rows and releases implicitly allocated resources for the current statement
mi_query_interrupt( )	Releases implicitly allocated resources for the current statement

After each of these functions executes, the next iteration of the **mi_get_result( )** function returns a status of MI_NO_MORE_RESULTS.

Finishing Execution

The **mi_query_finish( )** function completes execution of the current statement. The function performs the following steps:

- Processes any pending results that are not already processed with calls to **mi_next_row( )**
- Releases the implicit resources for the current statement

The **mi_query_finish( )** function does not affect prepared statements or calls to DataBlade API file-access functions. To determine whether the current statement has completed execution, use the **mi_command_is_finished( )** function.

Processing Remaining Rows

The **mi_exec( )** function opens an implicit cursor to hold the resulting rows of a query. For such queries, the **mi_query_finish( )** function ensures that the database server processes the results of a statement. The **mi_query_finish( )** function processes all the remaining rows in the cursor of the current statement and throws them away.

If the current statement failed, **mi_query_finish( )** returns MI_ERROR. In this case, **mi_query_finish( )** guarantees that the database server is ready for the next statement (unless the database server has dropped the connection). All callbacks are properly invoked during **mi_query_finish( )** processing.

Releasing Statement Resources

The **mi_query_finish( )** function releases implicitly allocated resources associated with the current statement. The following table summarizes the implicitly allocated resources for different query executions.

DataBlade API Function That Allocated Statement Resource	Resources That mi_query_finish( ) Releases
mi_exec( )	• Close any implicit cursor (for queries). • Release the implicit statement descriptor associated with the current statement. • Release any other resources associated with the current statement.

DataBlade API Function That Allocated Statement Resource	Resources That mi_query_finish( ) Releases
mi_exec_prepared_statement( ), **mi_open_prepared_statement( )**	None Use **mi_close_statement( )** and **mi_drop_prepared_statement( )**. For more information, see "Releasing Prepared-Statement Resources" on page 8-31.
mi_next_row( )	Release the row structure for the current row.
mi_get_row_desc_without_row( )	Release the row descriptor for the current row.

The **mi_exec( )** function creates an implicit statement descriptor and opens an implicit cursor for the SQL statement it executes. These structures have as their scope from the time they are allocated with **mi_exec( )** until whichever of the following events occurs first:

- The **mi_query_finish( )** function finishes execution of the current statement.
- The **mi_query_interrupt( )** function interrupts execution of the current statement.
- The **mi_close( )** function closes the connection.

Server Only

- The SQL statement that invoked the C UDR ends.

End of Server Only

To conserve resources, use the **mi_query_finish( )** function to explicitly close the implicit cursor and free the implicit statement descriptor once your DataBlade API module no longer needs access to the current statement. The **mi_query_finish( )** function is the destructor function for the implicit cursor and its associated implicit statement descriptor.

The **mi_query_finish( )** and **mi_query_interrupt( )** functions also free the implicit row structure and row descriptor that hold each row as it is fetched from a cursor. A general rule of DataBlade API programming is that you do not explicitly free a data type structure that you have not explicitly allocated. This rule applies to the row structure and row descriptor of the current statement, in particular:

- Do *not* explicitly free the row structure for the current statement (which the **mi_next_row( )** function returns).
- Do *not* explicitly free the row descriptor for the current statement (which the **mi_get_row_desc_without_row( )** function returns).

These data type structures are freed when the connection closes. For more information, see "Closing a Connection" on page 7-18.

Interrupting Execution

The **mi_query_interrupt( )** function interrupts execution of the current statement on a connection. It releases resources that an **mi_exec( )** function implicitly allocates without processing any remaining rows in a query. To release resources, **mi_query_interrupt( )** has the same behavior as **mi_query_finish( )**. For more information, see "Releasing Statement Resources" on page 8-57.

Inserting Data into the Database

To insert a row of data into a database, you must execute the INSERT statement.

To send an INSERT statement to the database server for execution:

1. Assemble the statement string.
2. Send the INSERT statement with either **mi_exec( )** or **mi_exec_prepared_statement( )**.
3. Process the results of the completed statement.

Assembling an Insert String

Assemble a statement string for the INSERT statement you want to execute. If you know the values you want to insert into the columns, you can create a basic SQL statement; that is, one that you can execute with **mi_exec( )**. If you do not know the column values, use input parameters in the statement string in place of the column values in the VALUES clause. You must prepare any parameterized INSERT statement with a call to **mi_prepare( )**. Figure 8-4 on page 8-12 shows a statement string for an INSERT that contains input parameters.

Sending the Insert Statement

The choice of DataBlade API statement-execution function for an INSERT statement depends on whether the statement string was prepared with **mi_prepare( )**, as follows:

- If the statement string is *not* prepared, use **mi_exec( )** to send the INSERT to the database server.
- If the statement string was prepared, use **mi_exec_prepared_statement( )** to send the INSERT to the database server.

Processing Insert Results

After the database server executes an INSERT statement, the **mi_get_result( )** function returns a MI_DML statement status. You can obtain the following information about the statement:

- The **mi_result_row_count( )** function returns the number of rows that were inserted.
- The value of the SERIAL or SERIAL8 column in the row just inserted.

The SERIAL and SERIAL8 data types allow you to have an integer column for which the database server automatically increments the value with each insert. You can obtain the newly inserted serial value for the most recent INSERT statement with the following DataBlade API functions.

Serial Data Type	DataBlade API Function
SERIAL	**mi_last_serial( )**
SERIAL8	**mi_last_serial8( )**

Using Save Sets

Save sets provide a mechanism for a DataBlade API module to access several rows simultaneously. When a DataBlade API module retrieves rows from a cursor in an **mi_next_row( )** loop, only one row is current at a time. Each iteration of **mi_next_row( )** overwrites the row from the previous iteration. If your DataBlade API module needs to perform comparisons or other types of processing on more

than one row, you can save the rows in a save set. The DataBlade API maintains a save set as a FIFO (first-in, first-out) queue.

The DataBlade API provides the save-set structure, MI_SAVE_SET, to hold the rows of a save set. The following table summarizes the memory operations for a save-set structure.

Memory Duration	Memory Operation	Function Name
PER_STMT_EXEC	Constructor	**mi_save_set_create( )**
	Destructor	**mi_save_set_destroy( )**

Table 8-8 lists the functions that the DataBlade API provides for use with a save set.

Table 8-8. Save-Set Functions of the DataBlade API

Save-Set Operation	DataBlade API Function
Determine the number of rows in a save set	**mi_save_set_count( )**
Create a new save set	**mi_save_set_create( )**
Delete a row from a save set	**mi_save_set_delete( )**
Free resources associated with a save set	**mi_save_set_destroy( )**
Get first row from a save set	**mi_save_set_get_first( )**
Get last row from a save set	**mi_save_set_get_last( )**
Get next row from a save set	**mi_save_set_get_next( )**
Get previous row from a save set	**mi_save_set_get_previous( )**
Insert a new row into the save set	**mi_save_set_insert( )**
Determine if a specified row is a member of a save set	**mi_save_set_member( )**

Creating a Save Set

You create a save set with **mi_save_set_create( )**, which returns a pointer to a save-set structure (MI_SAVE_SET). The **mi_save_set_create( )** function is a constructor for the save-set structure. You pass this save-set structure to other DataBlade API save-set functions so they can access the save set.

The save set is associated with a specified connection. Therefore, you must pass a connection descriptor into **mi_save_set_create( )**.

Inserting Rows into a Save Set

The **mi_save_set_insert( )** function inserts a row into a save set.

To insert a new row into a save set:

1. Obtain a row structure for the row you want to insert into the save set.

   This row structure is usually the row that the **mi_next_row( )** function retrieves from a cursor of a query.

2. Pass a pointer to this row structure to **mi_save_set_insert( )**.

   Because a save set is a FIFO structure, **mi_save_set_insert( )** appends the new row to the end of the save set.

If the insert is successful, the **mi_save_set_insert( )** function returns a pointer to the row structure it just inserted.

Building a Save Set

To build a save set, a DataBlade API module can create a save set and fetch rows into it from the **mi_next_row( )** loop. The rows inserted into the save set are copies of the rows in the database, so modifications to the database after a row is inserted into a save set are not reflected in the save set. In effect, the save set stores stale rows.

To build a save set:

1. Create the save set with **mi_save_set_create( )**.
2. Execute the query to select rows from the database (for example, with **mi_exec( )**).
3. When **mi_get_result( )** returns MI_ROWS, initiate an **mi_next_row( )** loop to get the rows.
4. Inside the **mi_next_row( )** loop, for each row that you want to save in the save set, invoke **mi_save_set_insert( )**.

The user function **build_saveset( )** creates a save set and inserts rows into it. It is called when **mi_get_result( )** returns MI_ROWS and the application wants to store the rows temporarily in a save set. Another user function, **get_saveset_data( )**, is called to access and manipulate the data in the save set.

```
/*
 * Example of how to build a save set.
 */
#include <mi.h>

mi_integer build_saveset( MI_CONNECTION *conn)
   {
   MI_SAVE_SET  *save_set;
   MI_ROW       *row;
   MI_ROW       *row_in_saveset;

   mi_integer   error;

   save_set = mi_save_set_create(conn);

   if ( NULL == save_set )
      {
      DPRINTF("trc_class", 11,
         ("Could not create save set\n"));
      return (-1);
      }

/* Insert each row into the save set */
   while( NULL != (row = mi_next_row(conn, &error)) )
      {

      row_in_saveset = mi_save_set_insert(save_set, row);

      if( NULL == row_in_saveset )
         {
         mi_db_error_raise(conn, MI_MESSAGE,
            "Could not insert into save set\n");
         return (-1);
         }
      } /* end while */

/* Check reason for mi_next_row( ) completion */
```

```
                    if ( error == MI_ERROR )
                    {
                     mi_db_error_raise(conn, MI_MESSAGE,
                       "Could not get next row\n" );
                     return(-1);
                    }

         /* Print out message to trace file */
             DPRINTF("trc_class", 11,
                 ("%d rows inserted in save set\n",
                 mi_save_set_count(save_set)));

             get_saveset_data(save_set);

             error =  mi_save_set_destroy(save_set);
             if( MI_ERROR == error )
                 {
                 mi_db_error_raise(conn, MI_MESSAGE,
                     "Could not destroy save set\n" );
                 return (-1);
                 }

             return(0);
         }
```

Once the **build_saveset( )** function successfully completes, the **get_saveset_data( )**
function can traverse the save set as a FIFO queue. The **mi_save_set_get_first( )**
function retrieves the first row of the save set, which is the most recently added
row. The DataBlade API module can scan forward through the save set with
mi_save_set_get_next( ) and then backward with **mi_save_set_get_previous( )**. All
of these routines return a pointer to the row structure (MI_ROW) for the current
row in the save set.

The following function, **get_saveset_data( )**, traverses the save set:

```
/*
 * Get Save-Set Data Example
 */
#include "mi.h"

mi_integer get_saveset_data( MI_SAVE_SET *save_set)
    {
    MI_ROW          *row;
    MI_ROW_DESC     *rowdesc;
    mi_integer      error;
    mi_integer      numcols;
    mi_integer      i;
    char            *colname;
    mi_integer      collen;
    char            *colval;

/* Get the first row from the save set */
    row = mi_save_set_get_first(save_set, &error);
    if (error == MI_ERROR)
        {
        DPRINTF("trc_class", 11,
            ("Could not get first row from save set\n"));
        return(-1);
        }

/* Get the description of the row */
    rowdesc = mi_get_row_desc(row);

/* Get the number of columns in the row */
    numcols = mi_column_count(rowdesc);
```

```
/* Print the column names */
   for ( i = 0; i < numcols; i++ )
      {
      colname = mi_column_name(rowdesc, i);
      DPRINTF("trc_class", 11, ("%s\t", colname));
      } /* end for */

   DPRINTF("trc_class", 11, ("\n\n"));

/* For each column */
   for ( i = 0; i < numcols; i++ )
      {
      switch( mi_value(row, i, &colval, &collen) )
         {
         case MI_ERROR:
            DPRINTF("trc_class", 11,
               ("\nCannot get value\n"));
            return(-1);

         case MI_NULL_VALUE:
            colval = "NULL";
            break;

         case MI_NORMAL_VALUE:
            break;

         default:
            DPRINTF("trc_class", 11,
               ("\nUnknown value\n"));
            return(-1);

         } /* end switch */

      DPRINTF("trc_class", 11, ("%s\t", colval));
      } /* end for */

/* For each row */
   while ( (row = mi_save_set_get_next(save_set, &error))
         != NULL )
      {
      if ( error == MI_ERROR )
         {
         DPRINTF("trc_class", 11,
            ("\nCould not get next row"));
         return (-1);
         }

/* For each column */
      for ( i = 0; i < numcols; i++ )
         {
         switch( mi_value(row, i, &colval, &collen) )
            {
            case MI_ERROR:
               DPRINTF("trc_class", 11,
                  ("\nCannot get value\n"));
               return(-1);

            case MI_NULL_VALUE:
               colval = "NULL";
               break;

            case MI_NORMAL_VALUE:
               break;

            default:
               DPRINTF("trc_class", 11,
                  ("\nUnknown value\n"));
```

```
                    break;

              } /* end switch */

          DPRINTF("trc_class", 11, ("%s\t", colval));

          } /* end for */

      DPRINTF("trc_class", 11, ("\n"));

      } /* end while */

  DPRINTF("trc_class", 11, ("\n"));
  return(1);
}
```

When a row is obtained from the save set, its values are extracted using an
mi_value( ) loop, as demonstrated in "Example: The get_data( ) Function" on
page 8-54.

Freeing a Save Set

A save-set structure has a memory duration of PER_STMT_EXEC. Therefore, a
save-set structure remains active until one of the following events occurs:

- The **mi_save_set_destroy( )** function frees the save-set structure.
- The end of the current SQL statement is reached.
- The **mi_close( )** function closes the current connection.

To conserve resources, use the **mi_save_set_destroy( )** function to explicitly
deallocate the save set once your DataBlade API module no longer needs it. The
mi_save_set_destroy( ) function is the destructor function for a save-set structure.
It frees the save-set structure and any resources that are associated with it.

Chapter 9. Executing User-Defined Routines

In This Chapter

This chapter covers the following topics about how to call a user-defined routine (UDR):

- How to access routine-state information from within a UDR
- How to use the DataBlade API Fastpath interface to execute a registered UDR
- How to use UDRs invoked in the trigger action statements to obtain information about the trigger and triggering table information and how to enable UDRs to recognize the High-Availability Data Replication (HDR) server status

Accessing MI_FPARAM Routine-State Information

When the routine manager calls a UDR, it passes the routine-state information as an extra argument, called the *function-parameter structure*, to the routine. This function-parameter structure, **MI_FPARAM**, holds the routine-state information for the C UDR with which it is associated.

This **MI_FPARAM** structure that the routine manager passes lasts for the duration of an SQL command (subquery execution). The following table summarizes the memory operations for an **MI_FPARAM** structure.

Memory Duration	Memory Operation	Function Name
PER_COMMAND	Constructor	Routine manager (when it invokes a UDR)
		mi_fparam_allocate( ), **mi_fparam_copy( )**
	Destructor	Routine manager (when it exits a UDR)
		mi_fparam_free( )

Most UDRs do *not* need to access this routine-state information. For such routines, you do not have to include an **MI_FPARAM** structure as a parameter in the C declaration. Your UDR needs to declare an **MI_FPARAM** parameter only if it needs to perform one of the following tasks.

Task	More Information
Obtain information about each routine argument, such as data type and whether it is NULL	"Checking Routine Arguments" on page 9-3
Obtain or set information about each return value, such as data type and whether it is NULL	"Accessing Return-Value Information" on page 9-6
Maintain user-state information between invocations of the routine for the duration of a single SQL statement	"Saving a User State" on page 9-8
Obtain information about the routine itself, such as the routine identifier and iterator information	"Obtaining Other Routine Information" on page 9-12

Tip: When you declare an **MI_FPARAM** parameter, this declaration must be the *last* parameter in the C declaration of your UDR. For more information, see "MI_FPARAM Argument" on page 13-4.

The UDR can then use the DataBlade API accessor functions that Table 9-1 on page 9-3, Table 9-2 on page 9-6, and Table 9-4 on page 9-12 list to access values in the **MI_FPARAM** structure.

Important: The **MI_FPARAM** structure is an opaque C structure to DataBlade API modules. Do not access its internal fields directly. The internal structure of **MI_FPARAM** may change in future releases. Therefore, to create portable code, always use the accessor functions for this structure to obtain and store values.

A UDR can also allocate an **MI_FPARAM** structure for a UDR that it invokes with the Fastpath interface. For more information, see "Using a User-Allocated MI_FPARAM Structure" on page 9-36.

Checking Routine Arguments

The user state of a C UDR provides the following information about routine arguments:

- Data type information about any arguments
- Boolean value to indicate whether an argument is NULL

Table 9-1 lists the DataBlade API accessor functions that obtain and set information about routine arguments in an **MI_FPARAM** structure.

Table 9-1. Argument Information in an MI_FPARAM Structure

Argument Information	DataBlade API Accessor Function
The *number* of arguments for the UDR with which the **MI_FPARAM** structure is associated	**mi_fp_nargs( )** **mi_fp_setnargs( )**
The *type identifier* of each argument that the **MI_FPARAM** structure contains	**mi_fp_argtype( )** **mi_fp_setargtype( )**
The *length* of each argument that the **MI_FPARAM** structure contains	**mi_fp_arglen( )** **mi_fp_setarglen( )**
The *precision* (total number of digits) of each argument that the **MI_FPARAM** structure contains	**mi_fp_argprec( )** **mi_fp_setargprec( )**
The *scale* of each argument that the **MI_FPARAM** structure contains	**mi_fp_argscale( )** **mi_fp_setargscale( )**
Whether each argument that the **MI_FPARAM** structure contains is an SQL NULL value	**mi_fp_argisnull( )** **mi_fp_setargisnull( )**

Determining the Data Type of UDR Arguments

With the **MI_FPARAM** structure, you can write UDRs that operate over a type hierarchy, rather than on a single type. At runtime, the routine can examine the **MI_FPARAM** structure to determine what data types were passed to the current invocation of the routine.

The **MI_FPARAM** structure stores the information about each UDR argument in several parallel arrays.

Argument Array	Contents
Argument-type array	Each element is a pointer to a type identifier (MI_TYPEID) that indicates the data type of the argument.
Argument-length array	Each element is the integer length of the data type for each argument.
Argument-scale array	Each element is the integer scale in the fixed-point argument. The default value of the scale elements is zero (0). Therefore, any arguments that do not have a fixed-point data type have a scale value of zero (0).
Argument-precision array	Each element is the integer precision in the fixed-point or floating-point argument. The default value of the precision elements is zero (0). Therefore, any arguments that have neither fixed-point nor floating-point data types have a precision value of zero (0).

Argument Array	Contents
Parameter-null array	Each element is either MI_FALSE or MI_TRUE: • MI_FALSE indicates that the argument is *not* an SQL NULL value. • MI_TRUE indicates that the argument is an SQL NULL value. For more information, see "Handling NULL Arguments with MI_FPARAM" on page 9-5.

Use the appropriate **MI_FPARAM** accessor function in Table 9-1 on page 9-3 to access the desired argument array.

All the argument arrays in the **MI_FPARAM** structure have zero-based indexes. To access information for the *n*th argument, provide an index value of *n*-1 to the appropriate accessor function, as Table 9-1 on page 9-3 shows. Figure 9-1 shows how the information at index position 1 of these arrays holds the argument information for the second argument of the UDR.
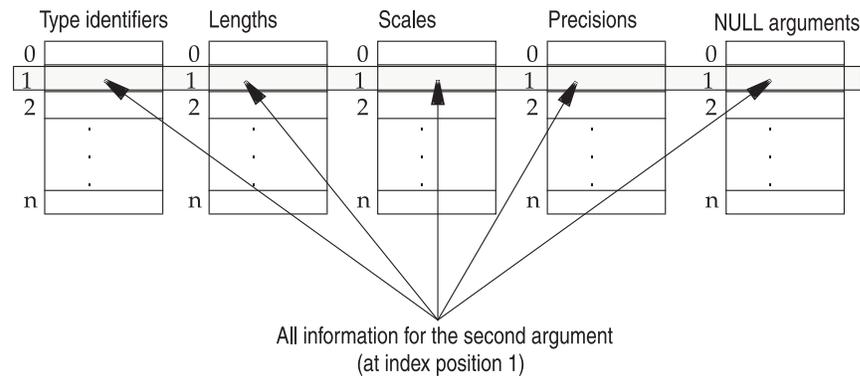
Figure 9-1. Argument Arrays in the MI_FPARAM Structure

The following calls to the **mi_fp_argtype( )** and **mi_fp_arglen( )** functions obtain the type identifier (**arg_type**) and length (**arg_len**) for the second argument from an **MI_FPARAM** structure that **fparam_ptr** identifies:

```
mi_integer my_func(arg0, arg1, arg2, fparam_ptr)
   MI_DATUM arg0;
   MI_DATUM arg1;
   MI_DATUM arg2;
   MI_FPARAM *fparam_ptr;
{
   MI_TYPEID *arg_type;
   mi_integer arg_len;
   ...
   arg_type = mi_fp_argtype(fparam_ptr, 1);
   arg_len = mi_fp_arglen(fparam_ptr, 1);
```

To obtain the number of arguments passed to the UDR (which is also the number of elements in the argument arrays), use the **mi_fp_nargs( )** function. For the argument arrays of the **MI_FPARAM** structure in the preceding code fragment, **mi_fp_nargs( )** would return a value of 3. The **mi_fp_setnargs( )** function stores the number of routine arguments in the **MI_FPARAM** structure.

Tip: For more information on type identifiers and lengths, see "Type Identifiers" on page 2-2. For more information on the scale and precision of fixed-point and floating-point data types, see Chapter 3, "Using Numeric Data Types," on page 3-1.

Handling NULL Arguments with MI_FPARAM

By default, C UDRs do *not* handle SQL NULL values. A UDR is *not* executed if any of its arguments is NULL; the routine automatically returns a NULL value. If you want your UDR to be invoked when it receives NULL values as arguments, take the following steps:

1. Use the following DataBlade API functions to programmatically handle SQL NULL values within the C UDR:
   - Use the **mi_fp_argisnull( )** function to determine whether an argument is NULL.
   - Use the **mi_fp_setargisnull( )** function to set an argument to NULL.

2. Register the UDR that checks for and handles NULL values with the HANDLESNULLS routine modifier in the CREATE FUNCTION or CREATE PROCEDURE statement.

   For more information on how to register a UDR, see "Registering a C UDR" on page 12-14.

The **mi_fp_argisnull( )** function obtains an **mi_boolean** value from an element in the null-argument array of the **MI_FPARAM** structure. If **mi_fp_argisnull( )** returns MI_TRUE, your UDR can take the appropriate action, such as supplying a default value or exiting gracefully from the routine. The code in Figure 9-2 implements the **add_one( )** function that returns a NULL value if the argument is NULL.

```
mi_integer add_one(i, fParam)
   mi_integer i;
   MI_FPARAM *fParam;
{
   /* determine if the first argument is NULL */
   if ( mi_fp_argisnull(fParam, 0) == MI_TRUE )
      {
      mi_db_error_raise(NULL, MI_MESSAGE,
         "Addition to a NULL value is undefined.\n");

      /* return an SQL NULL value */
      mi_fp_setreturnisnull(fParam, 0, MI_TRUE);

      /* the argument to this "return" statement is ignored by the
       * database server because the previous call to the
       * mi_fp_setreturnisnull( ) function has set the return value
       * to NULL
       */
      return 0;
      }
   else
      return(i+1);
}
```

Figure 9-2. The add_one( ) User-Defined Routine

The following CREATE FUNCTION statement registers a function named **add_one( )** in the database:

```
CREATE FUNCTION add_one(i INTEGER) RETURNS INTEGER
   WITH (HANDLESNULLS)
   EXTERNAL NAME '/usr/lib/db_funcs/add.so(add_one)'
   LANGUAGE C;
```

This CREATE FUNCTION statement omits the **MI_FPARAM** parameter of the **add_one( )** user-defined function from the definition of the SQL **add_one( )** UDR.

Accessing Return-Value Information

The **MI_FPARAM** structure of a C user-defined function provides the following information about function return values:

- Data type information about any return values
- Boolean value to indicate whether a return value is NULL

Important: Because a user-defined function is written in the C language, it can only return a single value. However, this single value can be a structure (such as a row descriptor) that contains multiple values. For information on how to return multiple values, see "Returning Multiple Values" on page 13-14.

Table 9-2 lists the DataBlade API accessor functions that obtain and set information about function return values in an **MI_FPARAM** structure. (Only user-defined functions return values; user-defined procedures do not.)

Table 9-2. Return-Value Information in the MI_FPARAM Structure

Return-Value Information	DataBlade API Accessor Function
The *number* of return values for the C UDR with which the **MI_FPARAM** structure is associated	**mi_fp_nrets( )** **mi_fp_setnrets( )**
The *type identifier* of each return value that the **MI_FPARAM** structure contains	**mi_fp_rettype( )** **mi_fp_setrettype( )**
The *length* of each return value that the **MI_FPARAM** structure contains	**mi_fp_retlen( )** **mi_fp_setretlen( )**
The *precision* (total number of digits) of each return value that the **MI_FPARAM** structure contains	**mi_fp_retprec( )** **mi_fp_setretprec( )**
The *scale* (number of digits to the right of the decimal point) of each fixed-point and floating-point return value that the **MI_FPARAM** structure contains	**mi_fp_retscale( )** **mi_fp_setretscale( )**
Whether each return value that the **MI_FPARAM** structure contains *is NULL*	**mi_fp_returnisnull( )** **mi_fp_setreturnisnull( )**

Determining the Data Type of UDR Return Values

The database server sets the return-value data type of the user-defined function. Most user-defined functions might need to check the return-value data type but they do not need to set it.

The routine manager uses the return-value information to determine how to bind the return value to a return variable or an SQL value. You need to access return-value information only if your UDR needs to perform one of the following tasks:

- Override the expected return type (for type hierarchies)

  You can set this return-value data type in the **MI_FPARAM** structure
- Set the actual length, precision, or scale of the return value
- Return an SQL NULL value

  See "Returning a NULL Value" on page 9-8.

- Check the return value of a UDR that you are going to execute with the Fastpath interface and for which you have created a user-allocated **MI_FPARAM** structure

  See "Using a User-Allocated MI_FPARAM Structure" on page 9-36.

If your UDR does *not* need to perform these tasks, it does not need to modify return-value information in the **MI_FPARAM** structure.

The **MI_FPARAM** structure uses several parallel arrays to store the following information about each return value.

Return-Value Array	Contents
Return-type array	Each element is a pointer to a type identifier (MI_TYPEID) that indicates the data type of the return value.
Return-length array	Each element is the integer length of the data type for each return value.
Return-scale array	Each element is the integer scale in the fixed-point return value.
Return-precision array	Each element is the integer precision of the fixed-point or floating-point return value.
Return-null array	Each element has either of the following values: - MI_FALSE: The return value is *not* an SQL NULL value. - MI_TRUE: The return value *is* an SQL NULL value. For more information, see "Returning a NULL Value" on page 9-8.

Use the appropriate **MI_FPARAM** accessor function in Table 9-2 on page 9-6 to access the desired return-value array.

All of the return-value arrays in the **MI_FPARAM** structure have zero-based indexes. To access information for the *n*th return value, provide an index value of *n*-1 to the appropriate accessor function in Table 9-2 on page 9-6. Figure 9-3 shows how the information at index position 0 of these arrays holds the return-value information for the first (and only) return value of a user-defined function.
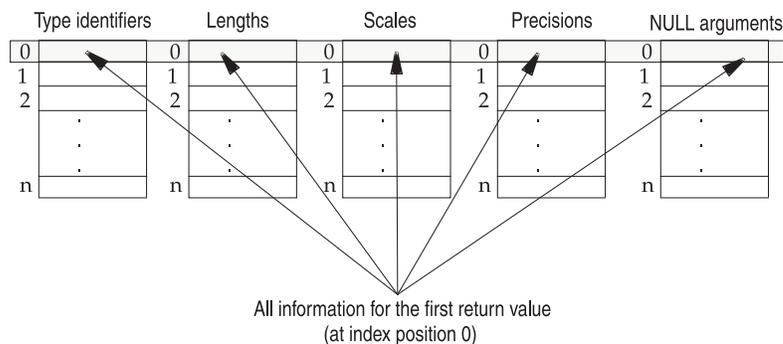
Figure 9-3. Return-Value Arrays in the MI_FPARAM Structure

The following calls to the **mi_fp_rettype( )** and **mi_fp_retlen( )** functions obtain the type identifier (**ret_type**) and length (**ret_len**) for the first (and only) return value from an **MI_FPARAM** structure that **fparam_ptr** identifies:

```
MI_FPARAM *fparam_ptr;
MI_TYPEID *ret_type;
mi_integer ret_len;
...
ret_type = mi_fp_rettype(fparam_ptr, 0);
ret_len = mi_fp_retlen(fparam_ptr, 0);
```

To obtain the number of return values of the user-defined function, use the **mi_fp_nrets( )** function. However, the number of return values is *always* 1 for a C user-defined function. The **mi_fp_setnrets( )** function stores the number of return values in the **MI_FPARAM** structure.

Returning a NULL Value

To return most values from a user-defined function, you use a C **return** statement. (For more information, see "Returning a Value" on page 13-12.) To return the SQL NULL value, however, you must access the **MI_FPARAM** structure of the UDR.

The DataBlade API provides the following functions to support the return of an SQL NULL value from a C user-defined function:

- To indicate that your user-defined function returns a NULL value, use the **mi_fp_setreturnisnull( )** function to store the value of MI_TRUE at the appropriate index position in the null-return array of the **MI_FPARAM** structure.
- The **mi_fp_returnisnull( )** function accesses the **MI_FPARAM** structure to determine whether a return value is NULL.

The **mi_fp_setreturnisnull( )** function sets an **mi_boolean** value to indicate whether the return value is NULL. The code in Figure 9-2 on page 9-5 implements the **add_one( )** function that uses the **mi_fp_setreturnisnull( )** function to return a NULL value when **add_one( )** receives a NULL argument.

Warning: Do not return a NULL-valued pointer from a UDR. If you need to have the UDR return an SQL NULL value, always use **mi_fp_setreturnisnull( )**. Otherwise, serious memory corruption might occur.

Saving a User State

The routine manager provides information about arguments and return values of a UDR in the **MI_FPARAM** structure that is associated with a UDR. In addition, you can store the address of private-state information, called *user-state information*, in a special field of the **MI_FPARAM** structure.

The database server passes the same **MI_FPARAM** structure to every invocation of the UDR within the same routine sequence. When your user-state information is part of the **MI_FPARAM** structure, your UDR can access this information across all the invocations within the same routine sequence. Your routine can use this private area of the **MI_FPARAM** structure to cache information that preserves its own state.

Tip: For more information about user state and the routine sequence, see "Creating the Routine Sequence" on page 12-22.

The **MI_FPARAM** structure can hold a *user-state pointer* that points to this private state information. The user-state pointer references a thread-private place holder that allows a UDR to associate a user-defined state information with a routine sequence. Table 9-3 shows that the DataBlade API provides the following accessor

functions to access the user state of a UDR.

Table 9-3. User-State Information in the MI_FPARAM Structure

User-State Information	DataBlade API Accessor Function
Obtain the user-state pointer from the **MI_FPARAM** structure of a UDR.	**mi_fp_funcstate( )**
Set the user-state pointer in the **MI_FPARAM** structure of a UDR.	**mi_fp_setfuncstate( )**

User-state information is useful for a UDR in the following cases:

- To save information between invocations of an iterator function

  For more information, see "Writing an Iterator Function" on page 15-3.

- To replace static or global variables in C function

  Use of the **MI_FPARAM** structure to hold state information enables a UDR to access global information without the use of static or global variables. It is *never* safe to use static and global variables that are updated because the updated value is not visible if the thread migrates to another virtual processor (VP) and concurrent activity is not interleaved.

Warning: Avoid the use of static and global variables in a UDR. If a UDR uses variables with these scopes, it is an ill-behaved routine. You must execute an ill-behaved UDR in a separate virtual-processor class, called a user-defined *VP*. For more information, see "Using Virtual Processors" on page 13-16.

To save user-state information in the first invocation of a UDR:

1. Use the **mi_fp_funcstate( )** function to retrieve the *user-state pointer* from the **MI_FPARAM** structure.

   Once the UDR has the user-state pointer, it can obtain state information from the private storage area on subsequent invocations.

2. Check for a NULL-valued user-state pointer.

   On the first invocation of your UDR, the user-state pointer is a NULL-valued pointer. If the user-state pointer is a NULL-valued pointer, allocate a private user-defined buffer or structure for the user-state information.

   When you allocate memory for the user-state information, you must protect this memory so that it is not reclaimed while it is still in use. Define a memory duration of PER_COMMAND for this memory with a DataBlade API memory-allocation function such as **mi_dalloc( )** or **mi_switch_mem_duration( )**. For more information, see "Choosing the Memory Duration" on page 14-4.

3. Put the private data in the user-defined buffer or structure to initialize the user state.

4. If the UDR has just allocated the private user-state buffer, use the **mi_fp_setfuncstate( )** function to store the address of this user-defined buffer or structure as a user-state pointer in the **MI_FPARAM** structure.

   You save the user-state pointer in the **MI_FPARAM** structure so that later UDR invocations of the routine sequence can access the routine-state information.

To obtain user-state information in subsequent invocations of the UDR:

1. Use the **mi_fp_funcstate( )** function to retrieve the *user-state pointer* from the **MI_FPARAM** structure.
2. If the user-state pointer is not a NULL-valued pointer, cast the pointer to the data type of your user-state information.

   Once the user-state pointer points to the correct data type, you can access the user-state information.

The **MI_FPARAM** structure is associated with the routine sequence. Therefore, for a UDR in a query that executes in parallel, each thread has its own routine sequence and therefore its own **MI_FPARAM** structure. The first invocation of the routine by each thread would have to perform any initialization. Only UDRs that are declared as *parallelizable* can be executed in parallel queries. The database server always executes an SQL statement that contains a nonparallelizable UDR serially.

Tip: By default, the CREATE FUNCTION statement registers a UDR as non-parallelizable. To declare a user-defined function as parallelizable, specify the PARALLELIZABLE routine modifier in the CREATE FUNCTION or CREATE PROCEDURE statement. For more information, see "Creating Parallelizable UDRs" on page 15-61.

The **MI_FPARAM** structure has a memory duration of PER_COMMAND. The database server reinitializes the user-state information that **mi_fp_funcstate( )** references to NULL at the end of the SQL command (for example, at the end of the subquery execution for each outer row from an outer query).

The code example in Figure 9-4 implements the **rowcount( )** function. This function uses the **MI_FPARAM** structure to hold a count of the number of rows in a query.

```
/* The rowcount( ) function maintains the row count with a variable that
 * is stored as user-state information in the MI_FPARAM structure
 */
mi_integer rowcount (fparam_ptr)
   MI_FPARAM *fparam_ptr;
{
   mi_integer *count = NULL;

   /* obtain the current user-state pointer from the MI_FPARAM structure */
   count = (mi_integer *)mi_fp_funcstate(fparam_ptr);

   /* if the user-state pointer is NULL, this is the first
    * invocation of the function
    */
   if ( count == NULL )
      {
      /* allocate memory for the user-state information */
      count = (mi_integer *)mi_dalloc(sizeof(mi_integer), PER_COMMAND);

      /* save user-state pointer in the MI_FPARAM structure */
      mi_fp_setfuncstate(fparam_ptr, (void *)count);

      /* initialize the row counter */
      *count = 0;
      }

   /* increment the row counter */
   (*count)++;
   return (*count);
}
```

Figure 9-4. Using the MI_FPARAM Structure to Hold Private-State Information

The **rowcount( )** function uses the **mi_fp_funcstate( )** function to obtain the user-state pointer from the **MI_FPARAM** structure. If this pointer is NULL, **rowcount( )** allocates memory for the **count** variable and uses the **mi_fp_setfuncstate( )** function to store this pointer as the user-state pointer in the **MI_FPARAM** structure. It uses the **mi_dalloc( )** function to allocate this memory with a duration of PER_COMMAND so that the database server does not deallocate the memory after the first invocation of the function.

Tip: The **rowcount( )** function in Figure 9-4 shows how to use the **MI_FPARAM** structure to hold private user-state information. This method removes the need for global or static variables, which can make a C UDR ill-behaved. Figure 13-8 on page 13-24 shows the **bad_rowcount( )** function, which incorrectly implements a row counter with a static variable.

For the **rowcount( )** function to be used in an SQL statement, it must be registered in the database. The following CREATE FUNCTION statement registers the **rowcount( )** function for use in SQL statements:

```
CREATE FUNCTION rowcount( ) RETURNS INTEGER
   EXTERNAL NAME '/usr/lib/db_funcs/count.so(rowcount)'
   LANGUAGE C;
```

The CREATE FUNCTION statement must omit the **MI_FPARAM** argument; therefore the registered **rowcount( )** function has no arguments. Suppose that the following query uses the **rowcount( )** function:

```
SELECT rowcount( ) from employee;
```

The query calls the **rowcount( )** function for each row in the **employee** table. Because the **rowcount( )** function uses the **MI_FPARAM** structure to hold its state

information (the **count** variable), each query has its own private **count** variable. Separate queries do not interfere with one another as they might with **static** and global variables.

Tip: You could also implement the **rowcount( )** function as a user-defined aggregate function. User-defined aggregates do not use the **MI_FPARAM** structure to hold state information. For more information, see "Writing an Aggregate Function" on page 15-11.

Obtaining Other Routine Information

The **MI_FPARAM** structure of a C UDR provides additional information about a UDR. Table 9-4 lists the DataBlade API accessor functions that obtain and set other routine information of a UDR.

Table 9-4. Other Routine Information in the MI_FPARAM Structure

Routine Information	DataBlade API Accessor Function
The *name* of the UDR	**mi_fp_funcname( )**
The *iterator status* for this iteration of an iterator function Values are SET_INIT, SET_RETONE, and SET_END.	**mi_fp_request( )**
The *iterator-completion flag*, which indicates whether an iterator function has finished returning rows of data	**mi_fp_setisdone( )**
The *identifier* of the UDR with which the **MI_FPARAM** structure is associated	**mi_fp_getfuncid( )**, **mi_fp_setfuncid( )**
The **MI_FPARAM** structure of the UDR A UDR needs to obtain the address of its **MI_FPARAM** structure only in special cases. For more information, see the description of the accessor function.	**mi_fparam_get_current( )**
The column identifier associated with the UDR A UDR needs to access its column identifier only in special cases. For more information, see the description of the accessor function.	**mi_fp_getcolid( )** **mi_fp_setcolid( )**
The row structure associated with the UDR A UDR needs to access its row structure only in special cases. For more information, see the description of the accessor function.	**mi_fp_getrow( )** **mi_fp_setrow( )**

For more information about the use of the iterator-completion flag and iterator status, see "Writing an Iterator Function" on page 15-3. For information about the use of routine identifiers, see "Routine Resolution" on page 12-19.

Calling UDRs Within a DataBlade API Module

Within a DataBlade API module, you can use either of the following methods to call a UDR, as long as you know the name of the UDR you want to call:

- Execute an SQL statement that invokes a registered UDR

────────────────────────── **Server Only** ──────────────────────────

- Call directly any UDR that resides in the same shared-object file

---
End of Server Only

Invoking a UDR Through an SQL Statement

You can call any registered UDR in an SQL statement. When your UDR is called in an SQL statement, the database server parses the statement and produces a query plan. It then automatically performs any routine resolution necessary and loads the shared-object file in which that UDR resides into shared memory (if it is not already loaded) when it parses and compiles the SQL statement. For more information on how the database server executes a UDR in an SQL statement, see "Executing a UDR" on page 12-18.

Within the DataBlade API, you can execute SQL statements with the **mi_exec( )** function and execute prepared SQL statements with the **mi_exec_prepared_statement( )** function. For example, the following call to **mi_exec( )** sends the EXECUTE FUNCTION statement to the database server to execute the **myfunc( )** user-defined function:

```
mi_exec(conn, "EXECUTE FUNCTION myfunc(5,5);",
    MI_QUERY_BINARY);
```

For more information on the use of **mi_exec( )** and **mi_exec_prepared_statement( )**, see Chapter 7, "Handling Connections," on page 7-1.

Calling a UDR Directly (Server)

From within a C UDR, you can directly call another C function when the following conditions are met:

- At compile time, you know the name of the C function that you want to call.
- The C function resides in the same shared-object file as the calling UDR.

This C function can be a registered UDR. In Figure 9-5, assume that the **func2( )** and **func3( )** functions were registered as user-defined functions with the CREATE FUNCTION statement. The **func3( )** user-defined function can directly call the **func2( )** UDR because **func3( )** and **func2( )** reside in the same shared-object file, **source1.so**.
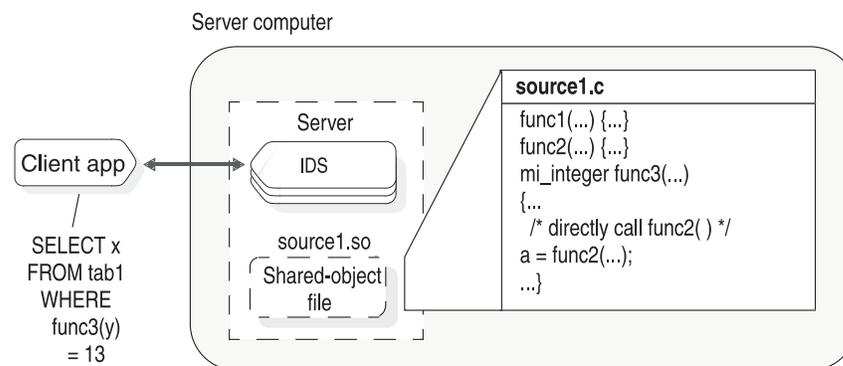
Figure 9-5. Calling a UDR Directly from Another UDR

If the UDR that you want to call is an overloaded routine, the database server executes the version of the UDR that resides in the *same* shared-object file. This UDR gets neither parameter casting nor a default **MI_FPARAM** structure. If no

version of this UDR exists in the same shared-object file, you receive a runtime error. To execute UDRs in other shared-object files, use the Fastpath interface.

Named Parameters and UDRs

Named parameters cannot be used to invoke UDRs that overload data types in their routine signatures. Named parameters are valid in resolving non-unique routine names only if the signatures have different numbers of parameters:

```
func( x::integer, y );     -- VALID if only these 2 routines
func( x::integer, y, z ); -- have the same 'func' identifier

func( x::integer, y );     -- NOT VALID if both routines have
func( x::float, y ;        -- same identifier and 2 parameters
```

For both ordinal and named parameters, the routine with the fewest parameters is executed if two or more UDR signatures have multiple numbers of defaults:

```
func( x, y default 1 )
func( x, y default 1, z default 2 )
```

If two registered UDRs that are both called `func` have the signatures shown above, then the statement `EXECUTE func(100)` invokes `func(100,1)`.

You cannot supply a subset of default values using named parameters unless they are in the positional order of the routine signature. That is, you cannot skip a few arguments and rely on the database server to supply their default values.

For example, given the signature:

```
func( x, y default 1, z default 2 )
```

you can execute:

```
func( x=1, y=3 )
```

but you cannot execute:

```
func( x=1, z=3 )
```

Calling UDRs with the Fastpath Interface

The *DataBlade API Fastpath interface* allows DataBlade API modules to directly invoke a UDR that was registered in the database. This interface bypasses the overhead associated with invoking a UDR through an SQL statement. This interface bypasses the query optimizer and executor (which are needed for an SQL statement). You can use this interface to execute any SQL routine.

Important: You cannot use the Fastpath interface to execute iterator functions or SPL functions with the WITH RESUME keywords in their RETURN statement. For more information on iterator functions, see "Writing an Iterator Function" on page 15-3.

The Fastpath interface is useful for calling a UDR in the following situations:

- You do not know the location of the UDR you want to call.
- At compile time, you do not know the name of the UDR you want to call.
- You need to promote or cast arguments of the UDR you want to call.

- The UDR you want to call is an overloaded routine and you need to obtain the version for particular argument data types.

- The UDR you want to call resides in a *different* shared-object file or DataBlade.

The Fastpath interface looks up a UDR or cast function in the system catalog tables to obtain information about the routine and then executes it. It passes any return values from the routine to the caller in internal (binary) format. With the Fastpath interface, a DataBlade API module can call a *foreign* UDR.

For a C UDR, a *foreign* UDR is a UDR that does not reside in the same shared-object file as the UDR that calls it. One UDR can only call another UDR directly when that called UDR resides within the *same* shared-object file or DataBlade module as the calling UDR. For example, in Figure 9-5 on page 9-13, the **func3( )** user-defined function can directly call **func2( )** because both of these functions reside in the **source1.so** shared-object file.

However, there is no portable way for the C code in one shared-object file to directly call a function in another shared-object file. Different operating systems provide different degrees of support for this type of calling. In addition, if the foreign UDR is part of a DataBlade module, your UDR has no way of knowing which DataBlade modules might be installed at a particular customer site.

To call a foreign UDR, a C UDR must use the DataBlade API Fastpath interface. Figure 9-6 shows how a UDR that is one shared-object file, **source1.so**, can call a foreign UDR, **funcB( )**. Even though the **funcB( )** routine is defined in the **source2.so** shared-object file, **func3( )** can invoke it through the Fastpath interface.
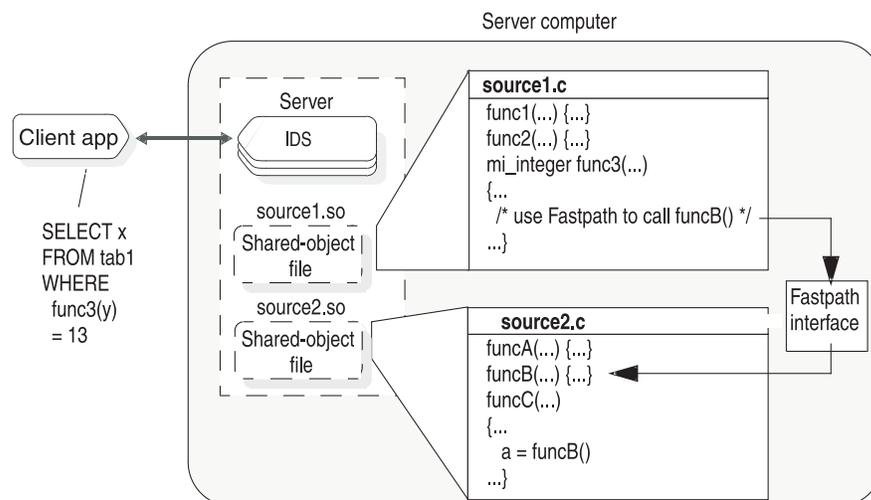
Figure 9-6. Using Fastpath to Access a Routine in Another Shared-Object File

In Figure 9-6, the Fastpath interface loads the **source2.so** shared-object file, which contains the **funcB( )** routine, into memory. For Fastpath to be able to invoke **funcB( )**, the **funcB( )** routine must already have been registered in the database.

The call to **funcB( )** within **funcC( )** does *not* require use of the Fastpath interface because these two functions reside in the same shared-object file, **source2.so**.

The Fastpath interface allows a DataBlade developer to extend a DataBlade module that someone else provides. This developer can define new UDRs on data types that some other DataBlade provides.

────────────── **End of Server Only** ──────────────

────────────── **Client Only** ──────────────

For a client LIBMI application, a *foreign* UDR is any UDR that is registered in the database that is currently open. Client LIBMI programs can use the Fastpath interface to directly invoke registered UDRs.

────────────── **End of Client Only** ──────────────

You can execute foreign UDRs with an SQL statement, such as EXECUTE FUNCTION. (For more information, see "Calling UDRs Within a DataBlade API Module" on page 9-12.) However, Fastpath is usually a quicker method for the execution of a UDR because it bypasses query processing.

Important: For the Fastpath interface to execute a UDR, the UDR must be registered in the database with a CREATE FUNCTION or CREATE PROCEDURE statement. When you create a DataBlade, register internal UDRs that might be of general use. In this way, you can cleanly protect private interfaces and support public ones.

The Fastpath interface provides the following DataBlade API functions to look up a registered UDR, execute it, and free resources.

DataBlade API Function	Purpose	More Information
Look up a UDR and obtain a function descriptor for it:		page 9-17
mi_cast_get( )	Looks up a cast function that casts between two data types (specified by type identifiers) and returns its function descriptor	
mi_func_desc_by_typeid( )	Looks up a UDR by its routine identifier and returns its function descriptor	
mi_routine_get( )	Looks up a UDR by its routine signature (specified as a character string) and returns its function descriptor	
mi_routine_get_by_typeid( )	Looks up a UDR by its routine signature (specified as separate arguments) and returns its function descriptor	
mi_td_cast_get( )	Looks up a cast function that casts between two data types (specified by type descriptors) and returns its function descriptor	
Obtain information from a function descriptor:		page 9-23
mi_fparam_get( )	Returns a pointer to the **MI_FPARAM** structure that is associated with the function descriptor	
mi_func_handlesnulls( )	Determines whether the UDR that is associated with the function descriptor can handle NULL arguments	

DataBlade API Function	Purpose	More Information
mi_func_isvariant( )	Determines whether the user-defined function that is associated with the function descriptor is a variant function	
mi_func_negator( )	Determines whether the user-defined function that is associated with the function descriptor has a negator function	
mi_routine_id_get( )	Returns the identifier for the routine that is associated with the function descriptor	
Execute the UDR through its function descriptor:		page 9-27
mi_routine_exec( )	Executes a UDR that is associated with a specified function descriptor	
Use a user-allocated **MI_FPARAM** structure for the UDR:		page 9-36
mi_fparam_allocate( )	Allocates an **MI_FPARAM** structure	
mi_fparam_copy( )	Creates a copy of an existing **MI_FPARAM** structure	
mi_fparam_free( )	Deallocates a user-allocated **MI_FPARAM** structure	
mi_fp_usr_fparam( )	Determines whether a specified **MI_FPARAM** has been allocated by the database server or the user	
Free resources that the function descriptor uses:		page 9-38
mi_routine_end( )	Releases resources that are associated with the function descriptor	

The following sections describe each of these tasks in detail.

Obtaining a Function Descriptor

A *function descriptor*, **MI_FUNC_DESC**, contains static information about a UDR that is to be invoked with the Fastpath interface. It is basically a structured version of the row in the **sysprocedures** system catalog table that describes the UDR. The function descriptor also identifies the routine sequence for the associated UDR. (For more information on the routine sequence, see "Creating the Routine Sequence" on page 12-22.)

The following table summarizes the memory operations for a function descriptor.

Memory Duration	Memory Operation	Function Name
PER_COMMAND	Constructor	**mi_cast_get( )**, **mi_func_desc_by_typeid( )**, **mi_routine_get( )**, **mi_routine_get_by_typeid( )**, **mi_td_cast_get( )**
	Destructor	**mi_routine_end( )**

Tip: Function descriptors are stored with the connection descriptor. Because a connection descriptor has a PER_COMMAND duration, so too does a function descriptor. However, it is possible to obtain a session-duration connection descriptor and, consequently, session-duration function descriptors. For more information, see "Obtaining a Session-Duration Connection Descriptor" on page 7-13.

A calling DataBlade API module uses a function descriptor as a handle to identify the UDR it needs to invoke with the Fastpath interface. To obtain a function descriptor, call one of the Fastpath look-up functions in Table 9-5.

Table 9-5. Fastpath Look-Up Functions

Type of UDRs	Fastpath Look-Up Function
Looking up general UDRs	**mi_func_desc_by_typeid( )**, **mi_routine_get( )**, **mi_routine_get_by_typeid( )**
Looking up cast functions	**mi_cast_get( )**, **mi_td_cast_get( )**

Looking Up UDRs

To look up a UDR, use one of the following Fastpath look-up functions.

DataBlade API Function	How It Looks Up a UDR
mi_routine_get( )	Looks up a UDR using a routine signature that is passed as a character string
mi_routine_get_by_typeid( )	Looks up a UDR using a routine signature that is passed as separate arguments
mi_func_desc_by_typeid( )	Looks up a UDR by its routine identifier and returns its function descriptor

─────────────────────────────── Server Only ───────────────────────────────

The **mi_func_desc_by_typeid( )** function is available only within a C UDR. It is *not* valid within a client LIBMI application.

─────────────────────────── End of Server Only ───────────────────────────

To obtain a function descriptor for a UDR, a Fastpath look-up function performs the following steps:

1. Asks the database server to look up the UDR in the **sysprocedures** system catalog

   If the name of the UDR in the routine signature that you specify is *not* unique in the database, the **mi_routine_get( )** and **mi_routine_get_by_typeid( )** functions use the routine signature to perform routine resolution. For more information, see "Routine Resolution" on page 12-19.

   ─────────────────────────────── Server Only ───────────────────────────────
   A routine identifier uniquely identifies a UDR, so the **mi_func_desc_by_typeid( )** function does not need to perform routine resolution. The routine identifier corresponds to the entry for the UDR in **sysprocedures.procid**. A negative routine identifier indicates a built-in function that does not have an entry in **sysprocedures**. The database server looks up information for a built-in function in an internal cache.
   ─────────────────────────── End of Server Only ───────────────────────────

2. Allocates a function descriptor for the routine and save the routine sequence in this descriptor

   You can obtain information about the UDR from this function descriptor. For more information, see "Obtaining Information from a Function Descriptor" on page 9-23. You can also allocate your own **MI_FPARAM** structure to use

instead of this automatically allocated one. For more information, see "Using a User-Allocated MI_FPARAM Structure" on page 9-36.

3. Allocates an **MI_FPARAM** structure for the function descriptor

   You can get a pointer to this structure with the **mi_fparam_get( )** function. For more information, see "Obtaining the MI_FPARAM Structure" on page 9-23.

4. Returns a pointer to the function descriptor that identifies the specified UDR

   Subsequent calls to **mi_routine_exec( )** can use this function descriptor to identify the UDR to execute. For more information, see "Executing the Routine" on page 9-27.

Suppose the following CREATE FUNCTION statements register three user-defined functions named **numeric_func( )** in your database:

```
CREATE FUNCTION numeric_func(INTEGER, INTEGER) RETURNS INTEGER;
CREATE FUNCTION numeric_func(FLOAT, FLOAT) RETURNS FLOAT;
CREATE FUNCTION numeric_func(MONEY, MONEY) RETURNS MONEY;
```

The **numeric_func( )** user-defined function is an overloaded routine. The code fragment in Figure 9-7 uses the **mi_routine_get( )** function to obtain the function descriptor for the version of **numeric_func( )** that handles INTEGER arguments.

```
MI_CONNECTION *conn;
MI_FUNC_DESC *fdesc = NULL;
MI_FPARAM *fparam;
...
fdesc = mi_routine_get(conn, 0,
   "function numeric_func(integer, integer)");
if ( fdesc =! NULL )
   {
   fparam = mi_fparam_get(conn, fdesc);
   if ( mi_fp_nrets(fparam) > 1 )
      /* multiple return values: have SPL routine */
      ...
   else
      /* no matching user-defined routine*/
      ...
   }
```

Figure 9-7. Obtaining a Function Descriptor for the numeric_func( ) Function

The **mi_routine_get( )** function returns a NULL-valued pointer to indicate either no matching routine exists or the routine has multiple return values. Figure 9-7 also shows how to determine which of these conditions a NULL return value indicates. It uses the **mi_fparam_get( )** function to obtain the **MI_FPARAM** structure that is associated with the located **numeric_func( )** function. (The **mi_routine_get( )** function has allocated and initialized this **MI_FPARAM** structure as part of the look-up process.) The code fragment then uses the **mi_fp_nrets( )** accessor function to obtain the number of UDR return values from this **MI_FPARAM** structure. Because C UDRs can only return one value, any UDR that returns more than one value must be an SPL routine.

Use **mi_routine_get( )** when you can create the full signature of the UDR as a literal string. Otherwise, you can use the **mi_routine_get_by_typeids( )** function to build the routine signature. For example, if you have a user-supplied query, you could use **mi_column_typeid( )** to get the type identifier for the column that the query returns. The **mi_routine_get_by_typedesc( )** function is also useful when you need to invoke overloaded UDRs with different parameter data types (and you have parameter type identifiers).

In Figure 9-7, you could replace the call to **mi_routine_get( )** with the following call to the **mi_routine_get_by_typeid( )** function:

```
MI_TYPEID *arg_types[2];
...
arg_type[0] = mi_typestring_to_id(conn, "integer");
arg_type[1] = arg_type[0];
fdesc = mi_routine_get_by_typeid(conn, MI_FUNC,
    "numeric_func", NULL, 2, arg_types);
```

In this call to **mi_routine_get_by_typeid( )**, the **arg_types** array contains pointers to the type identifiers for the two INTEGER parameters.

————————————— **Server Only** —————————————

If you already have a routine identifier for the UDR that you want to execute with Fastpath, use the **mi_func_desc_by_typeid( )** function to obtain the function descriptor of the UDR. In Figure 9-7, you could replace the call to **mi_routine_get( )** with the following call to **mi_func_desc_by_typeid( )**:

```
mi_funcid rout_id;
...
fdesc = mi_func_desc_by_typeid(conn, rout_id);
```

In this call, the **mi_funcid** data type holds the routine identifier of the UDR to look up.

————————————— **End of Server Only** —————————————

————————————— **Client Only** —————————————

When you call **mi_routine_get( )** or **mi_routine_get_by_typeid( )** from a client LIBMI application, the function allocates a local copy (on the client computer) of the function descriptor and **MI_FPARAM** structure. You can use the function descriptor and **MI_FPARAM** accessor functions within a client LIBMI application to access these local copies.

The **mi_func_desc_by_typeid( )** function is *not* valid within a client LIBMI application.

————————————— **End of Client Only** —————————————

Looking Up Cast Functions

A cast function is a user-defined function that converts one data type (the source data type) to a different data type (the target data type).

Tip: For more information on how to register a cast function, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

The way that a cast is called depends on the type of the cast, as the following table shows.

Type of Cast	How It Is Called
Built-in cast	Called by the database server automatically when built-in types need conversion in an SQL statement or a UDR call
Implicit cast	Called by the database server automatically when castable data types are part of an SQL statement

Type of Cast	How It Is Called
Explicit cast	• Called explicitly within an SQL statement with the **::** operator or CAST AS keywords • Called explicitly within a DataBlade API module with the Fastpath interface

To look up a cast function by its source and target data types, use one of the following Fastpath look-up functions.

DataBlade API Function	How It Looks Up a Cast Function
mi_cast_get( )	Looks up a cast function for source and target data types specified as type identifiers
mi_td_cast_get( )	Looks up a cast function for source and type data types specified as type descriptors

To obtain a function descriptor for a cast function, the **mi_cast_get( )** or **mi_td_cast_get( )** function performs the following steps:

1. Asks the database server to look up the cast function in the **syscasts** system catalog

   Once the function locates a **syscasts** entry for the cast function, it obtains routine information for the cast function from the **sysprocedures** system catalog table. These functions also determine the type of cast that the cast function performs: an explicit cast, an implicit cast, or a built-in cast.

2. Allocates a function descriptor for the cast function and save the routine sequence in this descriptor

   You can obtain information about the UDR from this function descriptor. For more information, see "Obtaining Information from a Function Descriptor" on page 9-23.

3. Allocates an **MI_FPARAM** structure for the function descriptor

   You can get a pointer to this structure with the **mi_fparam_get( )** function. For more information, see "Obtaining the MI_FPARAM Structure" on page 9-23. You can also allocate your own **MI_FPARAM** structure to use instead of this automatically allocated one. For more information, see "Using a User-Allocated MI_FPARAM Structure" on page 9-36.

4. Returns a pointer to the function descriptor that identifies the specified cast function

   Subsequent calls to **mi_routine_exec( )** can use this function descriptor to identify the cast function to execute. For more information, see "Executing the Routine" on page 9-27.

Tip: The **mi_cast_get( )** and **mi_td_cast_get( )** functions search for cast functions only in the **syscasts** system catalog table. Therefore, these functions can locate only cast functions that the CREATE CAST statement has registered.

Suppose the following CREATE CAST statements register explicit casts between the DECIMAL and **mytype** data types in your database:

```
CREATE CAST (mytype AS DECIMAL(5,3) WITH mt_to_dec);
CREATE CAST (DECIMAL(5,3) AS mytype WITH dec_to_mt);
```

Figure 9-8 uses the **mi_cast_get( )** function to obtain the function descriptor for a cast function that casts from a DECIMAL data type to the **mytype** data type.

```
MI_CONNECTION *conn;
MI_FUNC_DESC *fdesc2 = NULL;
MI_TYPEID *src_type, *trgt_type;
mi_char cast_status = 0;
mi_boolean need_cast = MI_TRUE;
mi_integer error;
MI_DATUM *ret_val;
mi_decimal dec_val;
...
/* Get type identifiers for source and target types */
src_type = mi_typestring_to_id(conn, "DECIMAL(5,3)");
trgt_type = mi_typestring_to_id(conn, "mytype");

/* Look up cast function based on type identifiers */
fdesc2 = mi_cast_get(conn, src_type, trgt_type, &cast_status);

switch ( cast_status )
    {
    case MI_ERROR_CAST: /* error in function look-up */
       mi_db_error_raise(NULL, MI_EXCEPTION, "mi_cast_get( ) failed");
       break;

    case MI_NO_CAST: /* no cast function exists */
       mi_db_error_raise(NULL, MI_EXCEPTION, "No cast function found");
       break;

    case MI_NOP_CAST: /* do not need a cast */
       need_cast = MI_FALSE;
       break;
    }

if ( need_cast )
    /* Execute the cast function with Fastpath */
    ...
```

Figure 9-8. Obtaining a Cast Function for a DECIMAL-to-mytype Cast

The **mi_cast_get( )** function allocates and initializes a function descriptor, **fdesc2**, for the **dec_to_mt( )** cast function. The **src_type** and **trgt_type** variables are pointers to the type identifiers for the DECIMAL(5,3) and **mytype** data types, respectively.

The **mi_cast_get( )** function returns a NULL-valued pointer to indicate several different conditions. In Figure 9-8, the **cast_status** variable identifies which of these conditions has occurred, as follows.

cast_status Value	Condition
MI_ERROR_CAST	The **mi_cast_get( )** function has not executed successfully.
MI_NO_CAST	No cast function exists between the specified source and target types.
MI_NOP_CAST	No cast function is needed between the specified source and target types.

The **switch** statement handles the possible status **cast_status** values from the **mi_cast_get( )** call.

If you have type descriptors instead of type identifiers for the source and target types, use the **mi_td_cast_get( )** function instead of **mi_cast_get( )**. For example,

the casting process might need information about scale and precision for built-in data types that have this information. A type descriptor stores scale and precision; a type identifier does not.

You could replace the call to **mi_cast_get( )** in Figure 9-8 with the following call to the **mi_td_cast_get( )** function:

```
MI_TYPEID *src_type, *trgt_type;
MI_TYPE_DESC *src_tdesc, *trgt_tdesc;

...
/* Get type descriptors for source and target types */
src_tdesc = mi_type_typedesc(conn, src_type);
trgt_tdesc = mi_type_typedesc(conn, trgt_type);

/* Look up cast function based on type descriptors */
fdesc2 = mi_td_cast_get(conn, src_tdesc, trgt_tdesc,
    &cast_status);
```

The **src_tdesc** and **trgt_tdesc** variables are pointers to type descriptors for the DECIMAL(5,3) and **mytype** data types, respectively. The **mi_type_typedesc( )** function creates type descriptors from the type identifiers that **src_type** and **trgt_type** reference.

Client Only

When you call **mi_cast_get( )** or **mi_td_cast_get( )** from a client LIBMI application, the function allocates a local copy (on the client computer) of the function descriptor and **MI_FPARAM** structure. You can use the function-descriptor and **MI_FPARAM** accessor functions within a client LIBMI application to access these local copies.

End of Client Only

Obtaining Information from a Function Descriptor

You can use the following DataBlade API functions to obtain additional information about the UDR or cast function that is associated with a function descriptor.

DataBlade API Function	Description
mi_fparam_get( )	Returns a pointer to the **MI_FPARAM** structure that is associated with the function descriptor
mi_routine_id_get( )	Returns the identifier for the UDR or cast function
mi_func_commutator( )	Determines if the UDR has a commutator function
mi_func_handlesnulls( )	Determines whether the UDR or cast function handles NULL arguments
mi_func_isvariant( )	Determines if the UDR or cast function is a variant function
mi_func_negator( )	Determines if the UDR has a negator function

Obtaining the MI_FPARAM Structure

By default, the Fastpath look-up functions allocate an **MI_FPARAM** structure and assign a pointer to this structure in the function descriptor. To obtain the **MI_FPARAM** structure that is associated with a function descriptor, use the **mi_fparam_get( )** function. After the call to **mi_fparam_get( )**, you can use the **MI_FPARAM** accessor functions to retrieve information from the **MI_FPARAM**

structure, such as argument information (Table 9-1 on page 9-3) and return-value information (Table 9-2 on page 9-6). For information about these accessor functions and the information that they can retrieve from an **MI_FPARAM** structure, see "Accessing MI_FPARAM Routine-State Information" on page 9-2.

Figure 9-7 on page 9-19 shows the use of the **mi_fparam_get( )** function to obtain the **MI_FPARAM** structure that is associated with the **numeric_func( )** user-defined function. This code fragment uses the **mi_fp_nrets( )** accessor function to obtain the number of return values for **numeric_func( )** from the **MI_FPARAM** structure.

Tip: You can allocate your own **MI_FPARAM** structure for a UDR that you execute with the Fastpath interface. For more information, see "Using a User-Allocated MI_FPARAM Structure" on page 9-36.

Obtaining a Routine Identifier

To obtain the identifier for a UDR or cast function that a function descriptor describes, use the **mi_routine_id_get( )** function. A routine identifier is a unique integer that identifies a UDR within the **sysprocedures** system catalog. This routine identifier is stored in the **procid** column of **sysprocedures**.

The following code fragment obtains the identifier for the **numeric_func( )** function that accepts INTEGER arguments:

```
MI_CONNECTION *conn;
MI_FUNC_DESC *fdesc;
mi_integer rout_id;
...
fdesc = mi_routine_get(conn, 0,
   "function numeric_func(integer, integer)");
rout_id = mi_routine_id_get(conn, fdesc);
```

─── **Server Only** ─────────────────────────

If your UDR is executing many other UDRs and it needs to keep several function descriptors for subsequent execution, it can use the routine identifiers to distinguish the different function descriptors.

─── **End of Server Only** ─────────────────

Determining If a UDR Handles NULL Arguments

Before you execute a UDR with the Fastpath interface, you can determine whether this routine handles SQL NULL values as arguments. If the current argument values are SQL NULL values and the UDR does not handle NULL values, you do not need to call the actual UDR. By default, a C UDR does *not* handle NULL values. When the routine manager receives NULL arguments for such a UDR, it does not even invoke the UDR. It just returns a NULL value.

To determine whether the UDR or cast function that a function descriptor describes can handle SQL NULL values as arguments, use the **mi_func_handlesnulls( )** function. This function determines whether the UDR was registered with the HANDLESNULLS routine modifier of the CREATE FUNCTION or CREATE PROCEDURE statement (stored in the **handlesnulls** column of the **sysprocedures** system catalog table).

The **mi_func_handlesnulls( )** function indicates whether a UDR can handle SQL NULL values, as follows.

mi_func_handlesnulls( ) Return Value	Meaning
1	The routine that the function descriptor describes has been registered with the HANDLESNULLS routine modifier.
2	The routine that the function descriptor describes has *not* been registered with the HANDLESNULLS routine modifier.

The code fragment in Figure 9-9 determines whether to invoke the
numeric_func( ) function based on whether it handles NULL arguments.

```
MI_CONNECTION *conn;
MI_FUNC_DESC *fdesc;
MI_FPARAM *fdesc_fparam;
MI_DATUM ret_val;
mi_integer error;
...
fdesc = mi_routine_get(conn, 0,
   "function numeric_func(integer, integer)");

/* Determine whether to execute the UDR */
if ( mi_func_handlesnulls(fdesc) == 1 )
   {
   fdesc_fparam = mi_fparam_get(conn, fdesc);
   mi_fp_setargisnull(fdesc_fparam, 0, MI_TRUE);
   ret_val = mi_routine_exec(conn, fdesc, &error, 0, 2);
   }
else
   /* have numeric_func( ) return zero if it has NULL args */
   ret_val = 0;
```

Figure 9-9. Handling Fastpath Execution of a UDR with NULL Arguments

If **numeric_func( )** handles NULL arguments, the **mi_func_handlesnulls( )**
function returns 1 and the code fragment invokes **numeric_func( )** with arguments
of NULL and 2. The code fragment uses the **mi_fparam_get( )** and
mi_fp_setargisnull( ) functions to set the first argument to an SQL NULL value.

If **numeric_func( )** does *not* handle NULL arguments, the code fragment does not
invoke **numeric_func( )**; instead, it sets the return value of **numeric_func( )** to
zero (0).

Checking for a Variant Function

Before you execute a UDR with the Fastpath interface, you might need to
determine whether this routine is variant. By default, a UDR *is* a variant function.
A *variant function* has one of the following characteristics:

- It returns different values when it is invoked with the same arguments.
- It has variant side effects that access some database or variable state.

A nonvariant function *always* returns the same value when it receives the same
arguments and it has none of the above variant side effects. Therefore, nonvariant
functions cannot contain SQL statements or access external files.

To determine whether a UDR is variant, pass its function descriptor to the
mi_func_isvariant( ) function. This function determines whether the user-defined
function was registered with the VARIANT or NOT VARIAnt routine modifier of
the CREATE FUNCTION or CREATE PROCEDURE statement. If the UDR was

registered with neither the VARIANT nor NOT VARIANT modifier, the **variant** column of **sysprocedures** indicates that the UDR is variant.

The **mi_func_isvariant( )** function indicates whether a UDR is variant, as follows.

mi_func_isvariant( ) Return Value	Meaning
1	The routine that the function descriptor describes is variant.
2	The routine that the function descriptor describes is *not* variant.

For more information about variant and nonvariant functions, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Checking for a Negator Function

Before you execute a Boolean user-defined function with the Fastpath interface, you might want to determine whether this function has a negator function. A *negator function* evaluates the Boolean NOT condition for its associated Boolean user-defined function. In many cases, a negator can be more efficient to execute than the actual Boolean user-defined function.

To determine whether a user-defined function has a negator function, pass its function descriptor to the **mi_func_negator( )** function. This function determines whether the user-defined function associated with this function descriptor was registered with the NEGATOR routine modifier of the CREATE FUNCTION statement. If so, **mi_func_negator( )** returns the name of the negator function (from the **negator** column of the **sysprocedures** system catalog table). If the negator is more efficient, you can use this function name to obtain a function descriptor for the negator function with a Fastpath function such as **mi_routine_get( )**.

For more information about negator functions, see "Creating Negator Functions" on page 15-60.

Checking for a Commutator Function

Before you execute a user-defined function with the Fastpath interface, you might want to determine whether this function has a commutator function. If a user-defined function has either of the following characteristics, it is a *commutator* of another user-defined function:

- A user-defined function takes the same arguments as another user-defined function, but in opposite order.
- A user-defined function returns the same result as another user-defined function.

In many cases, a commutator function can be more efficient to execute than the actual user-defined function.

To determine whether a user-defined function has a commutator function, pass its function descriptor to the **mi_func_commutator( )** function. This function determines whether the user-defined function associated with this function descriptor was registered with the COMMUTATOR routine modifier of the CREATE FUNCTION statement. If so, **mi_func_commutator( )** returns the name of the commutator function (from the **commutator** column of the **sysprocedures** system catalog table). If the commutator is more efficient, you can use this function name to obtain a function descriptor for the commutator function with a Fastpath function such as **mi_routine_get( )**.

For more information about commutator functions, see "Creating Commutator Functions" on page 15-60.

Executing the Routine

The **mi_routine_exec( )** function can execute any UDR that is registered in the open database. Therefore, any UDR that you can use in an SQL statement, you can directly execute with **mi_routine_exec( )**. Once you obtain a function descriptor for a registered UDR or cast function, the **mi_routine_exec( )** function sends it to the routine manager for execution. You can use the function descriptor in repeated calls to **mi_routine_exec( )**. This executed routine runs in the virtual processor (VP) that was defined for it. This VP is not necessarily the VP in which the calling UDR runs.

Important: You cannot use the Fastpath interface to execute iterator functions or SPL functions with the WITH RESUME keywords in their RETURN statement. However, to simulate iterator functionality, you can call the same UDR repeatedly, passing it the same **MI_FPARAM** structure. Each invocation of the UDR can return one value. For more information on iterator functions, see "Writing an Iterator Function" on page 15-3.

The **mi_routine_exec( )** function takes the following steps:

1. Passes the argument values in its argument list to the UDR
2. Returns any return value from the user-defined function to the calling module

Passing in Argument Values

When you call **mi_routine_exec( )**, you provide argument values for the UDR that Fastpath is to execute. Keep the following points in mind when you create the argument list for **mi_routine_exec( )**:

- If an argument has a default value, you do not need to include its argument value in this argument list.
- You must pass in the arguments as **MI_DATUM** values.

  Therefore, you must pass in the arguments with the appropriate passing mechanism for the data type. Most data types are to be passed by reference to the UDR. For a list of data types that can be passed by value, see Table 2-5 on page 2-33. For examples of how to pass arguments to a UDR through **mi_routine_exec( )**, see "Sample mi_routine_exec( ) Calls" on page 9-28.
- If the UDR can handle NULL arguments, pass a NULL-valued pointer as an argument to **mi_routine_end( )**, as Figure 9-9 on page 9-25 shows.

  If you call a UDR that does *not* handle NULLs with a NULL argument value, the routine might return incorrect values.
- If you allocate your own **MI_FPARAM** structure, pass in a pointer to this structure as the *last* argument in the argument list.

  For more information, see "Using a User-Allocated MI_FPARAM Structure" on page 9-36.

The **mi_routine_exec( )** function dispatches the UDR through the routine manager. Therefore, each UDR gets a call to **mi_call( )** because the routine manager checks for sufficient space before it executes the UDR.

Receiving the Return Value

When the **mi_routine_exec( )** function executes a user-defined function, it returns as an **MI_DATUM** value the return value of the user-defined function that it has

executed. This **MI_DATUM** value contains a value appropriate for the passing mechanism for the data type. Most data types are to be passed by reference from the UDR. For a list of data types that can be passed by value, see Table 2-5 on page 2-33.

You can then use C casting to convert this **MI_DATUM** value to the appropriate data type. You can obtain information about the return type (such as its data type) from the **MI_FPARAM** structure.

If the user-defined function returned an SQL NULL value, **mi_routine_exec( )** returns a NULL-valued pointer and sets the *status* argument to MI_OK.

For examples of how to receive a UDR return value from **mi_routine_exec( )**, see "Sample mi_routine_exec( ) Calls" on page 9-28. For more information on **MI_DATUM** values, see "The MI_DATUM Data Type" on page 2-32.

Sample mi_routine_exec( ) Calls

The code fragment in Figure 9-10 uses the **mi_routine_exec( )** function to execute the **numeric_func( )** routine that has INTEGER parameters.

```
MI_CONNECTION *conn;
MI_FUNC_DESC *fdesc;
MI_DATUM ret_val;
mi_integer int_ret;
mi_integer error;
...
/* fdesc obtains from code in Figure 9-7 on page 9-19 */
ret_val = mi_routine_exec(conn, fdesc, &error, 1, 2);
if ( ret_val == NULL )
    {
    if ( error == MI_OK )
       /* numeric_func( ) returned an SQL NULL value */
       ...
    else /* error in mi_routine_exec( ) */
       {
       mi_db_error_raise(NULL, MI_EXCEPTION,
          "mi_routine_exec( ) failed");
       return MI_ERROR;
       }
    }
else /* cast MI_DATUM in ret_val to INTEGER */
    {
    int_ret = (mi_integer) MI_DATUM;
    ...
```

Figure 9-10. Executing the numeric_func( ) Function with INTEGER Arguments

In Figure 9-10, the **mi_routine_exec( )** function uses the **fdesc** function descriptor that the **mi_routine_get( )** function obtained for the **numeric_func( )** function that accepts INTEGER arguments and returns an INTEGER value (Figure 9-7 on page 9-19). The last two arguments to **mi_routine_exec( )** are the integer argument values for **numeric_func( )**. The **ret_val** variable is an **MI_DATUM** value for the **mi_integer** data type that the **numeric_func( )** function returns.

Figure 9-10 also tests the return status of **mi_routine_exec( )** to check for the return value from **numeric_func( )**. If the return value (**ret_val**) is a NULL-valued pointer, the code then determines which of the following results this NULL value indicates:

- The **numeric_func( )** function has returned an SQL NULL value: **error** is MI_OK.
- The **mi_routine_exec( )** function failed: **error** is *not* MI_OK.

Finally, the code fragment in Figure 9-10 casts the **MI_DATUM** value that **mi_routine_exec( )** has returned to an **mi_integer** value. The **MI_DATUM** structure contains the actual return value because the routine manager can pass integer values by value (they can fit into an **MI_DATUM** structure).

Suppose the call to **mi_routine_get( )** had been calling the version of **numeric_func( )** that accepted a FLOAT argument and returned a FLOAT value, as follows:

```
fdesc = mi_routine_get(conn, 0,
   "function numeric_func(float)");
```

The call to **mi_routine_exec( )** to execute this version of **numeric_func( )** would require that the argument and the return value be passed by reference, because the FLOAT data type cannot be stored directly in an **MI_DATUM** structure.

Figure 9-11 shows a sample call to **mi_routine_exec( )** to execute the **numeric_func**(FLOAT) user-defined function. The argument list to **mi_routine_exec( )** passes the FLOAT value by reference and the returned FLOAT value is returned by reference.

```
MI_CONNECTION *conn;
MI_FUNC_DESC *fdesc;
MI_DATUM ret_val;
mi_double_precision double_arg1, double_arg2, double_ret;
mi_integer error;
...
fdesc = mi_routine_get(conn, 0, "function numeric_func(float)");
ret_val = mi_routine_exec(conn, fdesc, &error, &double_arg1,
      &double_arg2);
if ( ret_val == NULL )
    {
    if ( error == MI_OK )
       /* numeric_func( ) returned an SQL NULL value */
       ...
    else /* error in mi_routine_exec( ) */
       {
       mi_db_error_raise(NULL, MI_EXCEPTION,
          "mi_routine_exec( ) failed");
       return MI_ERROR;
       }
    }
else /* cast MI_DATUM in ret_val to FLOAT */
    {
    double_ret = (mi_double_precision *)MI_DATUM;
    ...
```

Figure 9-11. Executing the numeric_func( ) Function with INTEGER Arguments

The following call to **mi_routine_exec( )** executes the DECIMAL-to-**mytype** cast function for which Figure 9-8 on page 9-22 obtained the function descriptor, **fdesc2**:

```
ret_val = mi_routine_exec(conn, fdesc2, &error, dec_val);
mytype_val = (mytype *)ret_val;
```

The **dec_val** argument is an **mi_decimal** variable that contains the DECIMAL source data type to cast to the **mytype** target data type with the **dec_to_mt( )** cast function. The **ret_val** variable is an **MI_DATUM** value of the **mytype** data type that contains the casted DECIMAL value.

Executing a Built-in Cast Function

The execution of a built-in cast (system or MI_SYSTEM_CAST) function is different from the execution of a user-defined cast function. A user-defined cast function takes only one argument, the data value to be converted, but a built-in cast function takes three arguments:

- The source value to be cast.
- The length or the maxlength of the target type. If this parameter is NULL, then the casting function obtains the length from the return type information in **FPARAM**. It is best to either pass this value by getting the length or the maxlength value from the target type or to use the **mi_fp_setretlen(fparam,0,length)** function.
- Target type precision value. For the DATETIME, INTERVAL, DECIMAL, or MONEY data types, the total precision value also includes the scale value. It is recommended that you set this target type precision value as NULL, and then use the **mi_fp_setretprec( )** and the **mi_fp_setretscale( )** functions to set the return value's precision and scale.

The **FPARAM** information for the value argument comes from the source type for the cast. The length and precision fields are both integers so their precision is always 0. Below is an example of a system cast.

```
                        if (cast_status == MI_SYSTEM_CAST)
              {
                        mi_integer      precision
=
mi_type_precision(tdTarget);
                        mi_integer         length =
mi_type_maxlength(tdTarget);
              mi_integer scale = mi_type_scale(tdTarget);
                                   fd_fparam =
mi_fparam_get(conn,fd);
                  ....
                  .....
              mi_setretlen(fd_fparam,0,length);
              mi_setretprec(fd_fparam,0,precision);
              mi_setretscale(fd_fparam,0,scale);
            retValue = mi_routine_exec(conn, fd, &error,
 value,
NULL,NULL);
```

Reusing a Function Descriptor

Looking up a UDR and creating its function descriptor can be an expensive operation. If you have multiple UDR invocations calling the same UDR through Fastpath, you can cache the function descriptor to make it available for reuse within either of the following scopes:

- Within the current SQL command

  If the same UDR executes many times within a single SQL command, you can cache this function descriptor as part of the **MI_FPARAM** structure of the UDR.

- Within the session

  If the same UDR executes many times within a session, you can cache the function descriptor in PER_SESSION named memory.

When you reuse the function descriptor, you save the overhead of looking up the UDR and allocating a function descriptor each time you reexecute the same UDR.

Function Descriptors Within an SQL Command: When you pass a public connection descriptor to one of the Fastpath look-up functions (see Table 9-5 on page 9-18), the look-up function allocates the function descriptor with a PER_COMMAND memory duration. Therefore, the function descriptor remains allocated for the duration of the SQL command that invokes the UDR.

To cache the function descriptor, save its address in the **MI_FPARAM** structure of the UDR. The **MI_FPARAM** structure has a PER_COMMAND duration. Therefore, storing the function-descriptor address in the user state of **MI_FPARAM** guarantees that all UDR invocations within the SQL command can access it. You can also cache the connection description in **MI_FPARAM**. When the SQL command completes, the function descriptor, the connection descriptor, and the **MI_FPARAM** structure are deallocated.

The following code fragment for the **non_optimal( )** UDR opens a connection and obtains the function descriptor for the user-defined function **equal( )** *for each invocation* of **non_optimal( )**:

```
mi_integer non_optimal(arg1, arg2, fparam)
   mi_integer arg1, arg2;
   MI_FPARAM *fparam;
{
   MI_CONNECTION *conn=NULL;
   MI_FUNC_DESC *func_desc=NULL;
   MI_DATUM func_result;
   mi_integer func_error;
   mi_string *func_sig="equal(int, int)";

/* Open a connection */
if ( (conn = mi_open(NULL, NULL, NULL))
     == (MI_CONNECTION *)NULL )
   {
   mi_db_error_raise(NULL, MI_EXCEPTION,
     "mi_open( ) call failed");
   return MI_ERROR;
   }

/* Get the function descriptor for equal( ) */
if ( (func_desc = mi_routine_get(conn, 0, func_sig))
     == (MI_FUNC_DESC *)NULL )
   {
   mi_db_error_raise(NULL, MI_EXCEPTION,
     "mi_routine_get( ) call failed");
   return MI_ERROR;
   }

/* Execute the equal( ) user-defined function */
func_result = mi_routine_exec(conn, func_desc, &func_error,
   arg1, arg2);
if ( func_error == MI_ERROR )
   {
   mi_db_error_raise(NULL, MI_EXCEPTION,
     "mi_routine_exec( ) call failed");
   return MI_ERROR;
   }
...
```

When you cache the function descriptor in the user state of **MI_FPARAM**, you do not have to repeatedly call **mi_routine_get( )** for each invocation of the **equal( )** function. Instead, you can call **mi_routine_get( )** *only* in the first invocation of

equal( ). The following code fragment opens a connection and gets the function descriptor in the *first* invocation of the user-defined function **equal( )** only. It caches these descriptors into the **user_state** structure, whose address it stores in the user-state pointer of **MI_FPARAM**, for subsequent invocations of **equal( )**:

```
typedef struct user_state
   {
   MI_CONNECTION *conn;
   MI_FUNC_DESC *func_desc;
   };

mi_integer optimal(arg1, arg2, fparam)
   mi_integer arg1, arg2;
   MI_FPARAM *fparam;
{
   MI_CONNECTION *local_conn = NULL;
   MI_FUNC_DESC *local_fdesc = NULL;
   mi_string *func_sig="equal(int, int)";
   user_state *func_state;

   /* Obtain the connection descriptor from MI_FPARAM */
   func_state = (user_state *)mi_fp_funcstate(fparam);
   if ( func_state == NULL ) /* first time UDR is called */
      {
      /* Allocate a user_state structure */
      func_state =
         (user_state *)mi_dalloc(sizeof(user_state),
         PER_COMMAND);

      /* Obtain the connection descriptor */
      if ( (local_conn = mi_open(NULL, NULL, NULL))
           == (MI_CONNECTION *)NULL )
         {
         mi_db_error_raise(NULL, MI_EXCEPTION,
            "mi_open( ) call failed");
         return MI_ERROR;
         }

      /* Obtain the function descriptor for equal( ) */
      if ( (local_fdesc =
           mi_routine_get(local_conn, 0, func_sig))
           == (MI_FUNC_DESC *)NULL )
         {
         mi_db_error_raise(NULL, MI_EXCEPTION,
            "mi_routine_get( ) call failed");
         return MI_ERROR;
         }

      /* Cache the connection descriptor and function
       * descriptor in MI_FPARAM
       */
      func_state->conn = local_conn;
      func_state->func_desc = local_fdesc;

      /* Save the user state in MI_FPARAM */
      mi_fp_setfuncstate(fparam, (void *)func_state);
      }

   /* Execute the equal( ) user-defined function */
   func_result = mi_routine_exec(func_state->conn,
      func_state->func_desc, &func_error, arg1, arg2);
   if ( func_error == MI_ERROR )
      {
      mi_db_error_raise(NULL, MI_EXCEPTION,
```

```
        "mi_routine_exec( ) call failed");
    return MI_ERROR;
    }
...
```

Function Descriptors Within a Session (Server): When you pass a session-duration connection descriptor to one of the Fastpath look-up functions (see Table 9-5 on page 9-18), the look-up function allocates a function descriptor with a PER_SESSION memory duration, called a *session-duration function descriptor*. The session-duration function descriptor remains allocated until the session ends. In this case, all UDRs within the session can access the cached function descriptor.

Warning: The session-duration connection descriptor and session-duration function descriptor are advanced features of the DataBlade API. They can adversely affect your UDR if you use them incorrectly. In addition, session-duration function descriptors require named memory to store the pointers to function descriptors. Without named memory, UDRs cannot share these pointers. Named memory is also an advanced feature of the DataBlade API. Use a session-duration function descriptor only when a regular function descriptor cannot perform the task you need done.

The following table summarizes the memory operations for a session-duration function descriptor in a C UDR.

Memory Duration	Memory Operation	Function Name
PER_SESSION	Constructor	**mi_cast_get( )**, **mi_func_desc_by_typeid( )**, **mi_routine_get( )**, **mi_routine_get_by_typeid( )**, **mi_td_cast_get( )**
		When passed a session-duration connection descriptor instead of a public connection descriptor
	Destructor	**mi_routine_end( )**
		When the session ends

Caching a Session-Duration Function Descriptor: To cache the function descriptor, save its address in PER_SESSION named memory. This location guarantees that all UDRs within the session can access the function descriptor. A UDR can create a session-duration function descriptor and cache it as follows:

• Obtain a session-duration connection descriptor with the **mi_get_session_connection( )** function.
• Allocate named memory with a PER_SESSION memory duration to hold the address of the session-duration function descriptor.

  The UDR must store the pointers to these function descriptors in a named-memory block so that they can be accessed across different UDRs. Use the **mi_named_alloc( )** or **mi_named_zalloc( )** function to allocate the PER_SESSION memory. Other UDRs might also be using the same session-duration connection. Therefore, you might need to handle concurrency issues on the named memory.

• Allocate a session-duration function descriptor by passing the session-duration connection descriptor to one of the Fastpath look-up functions (see Table 9-5 on page 9-18).

When one of these look-up functions receives a session-duration connection descriptor (instead of a public connection descriptor), it allocates a session-duration function descriptor.

The following code fragment uses the **mi_routine_get( )** function to obtain a session-duration function descriptor for the **func1( )** UDR:

```
MI_CONNECTION *sess_conn;
MI_FUNC_DESC **fdesc;
mi_integer status;

/* Obtain a session-duration connection descriptor */
sess_conn = mi_get_session_connection( );

/* Allocate a PER_SESSION named-memory block named
 * 'funcptr_blk'. Assign address of this block to fdesc
 * pointer.
 */
if ( (status = mi_named_alloc((sizeof)(MI_FUNC_DESC *),
      "funcptr_blk", PER_SESSION, (void **)&fdesc))
      != MI_OK )
   {
   /* Unable to allocate named-memory block. Handle error */
   }

/* Obtain the session-duration function descriptor for
 * func1( ). Store function descriptor in named-memory block.
 */
if ( (*fdesc = mi_routine_get(sess_conn, 0,
      "function func1(int, char)") == (MI_FUNC_DESC *)NULL)
   {
   /* Unable to obtain function descriptor for func1( ) UDR.
    * Handle error.
    */
   }
```

The preceding code fragment uses the **mi_get_session_connection( )** function to obtain the session-duration connection descriptor, **sess_conn**. It then passes **sess_conn** to the **mi_routine_get( )** function to obtain a session-duration function descriptor for **func1( )**. The address of this session-duration function descriptor is stored in a PER_SESSION named-memory block named **funcptr_blk**. All UDRs that need to access the **func1( )** function descriptor can obtain it from **funcptr_blk**. For more information, see "Reusing the Session-Duration Function Descriptor" on page 9-34.

Important: Do not store the address of a session-duration function descriptor in the **MI_FPARAM** structure of the UDR. Neither should you allocate PER_SESSION memory with **mi_dalloc( )** and store the address of this memory in **MI_FPARAM**. Both these methods cause the address of the session-duration function descriptor to be lost because the **MI_FPARAM** structure gets freed when the UDR instance completes. However, you can optimize the named-memory look-up by caching the address of the named-memory block in **MI_FPARAM**. This method requires only one call to **mi_named_get( )** for each instance of the UDR. The first UDR invocation that needs the information must allocate the named-memory block and populate the named memory.

Reusing the Session-Duration Function Descriptor: To reuse the session-duration function descriptor in another UDR, you:

• Get the address of the PER_SESSION named-memory block that contains the function descriptor.

- Extract the address of the session-duration function descriptor from the named-memory block.
- Pass the session-duration function descriptor to the **mi_routine_exec( )** function to execute the associated UDR.

The following code fragment shows how other UDRs can access the **fdesc** function descriptor until the session ends:

```
MI_CONNECTION *sess_conn;
MI_FUNC_DESC **fdesc;
mi_integer status, error;
mi_integer i = 55;
mi_char ch = 'c';
MI_DATUM *value;

/* Obtain a public and session-duration connection
 * descriptor
 */
conn = mi_open(NULL, NULL, NULL);
sess_conn = mi_get_session_connection( );

/* Obtain the address of the 'funcptr_blk' named-memory block,
 * which contains the session-duration function descriptor
 */
if ( (status = mi_named_get("funcptr_blk", PER_SESSION,
     (void **)&fdesc)) != MI_OK )
   {
   /* Handle error */
   ...
   }

/* Execute the UDR associated with the 'fdesc'
 * session-duration function descriptor
 */
value = mi_routine_exec(conn, *fdesc, &error, i, c);
```

Once you obtain the session-duration function descriptor, the **mi_routine_exec( )** function can execute the associated UDR. You can specify either a public connection descriptor or a session-duration connection descriptor to **mi_routine_exec( )**. If the UDR that the function descriptor references is dropped while the session-duration function descriptor is still in use, the database server generates an error when you try to execute the routine.

Deallocating a Session-Duration Function Descriptor: The session-duration function descriptor has a PER_SESSION memory duration. The function descriptor remains active until either of the following events occurs:

- The **mi_routine_end( )** function explicitly releases the session-duration function descriptor.

  To explicitly free a session-duration function descriptor, you must free both the function descriptor with **mi_routine_end( )** and the associated PER_SESSION named memory with **mi_named_free( )**, as the following code fragment shows:

  ```
  /* Free resources for the session-duration function
   * descriptor
   */
  if ( mi_routine_end(sess_conn, *fdesc) != MI_OK )
     {
     /* Handle error */
     ...
     }
  mi_named_free("funcptr_blk", PER_SESSION);
  ```
- The client application ends the session.

The database server automatically frees memory for the session-duration function descriptor and its PER_SESSION named memory at the end of the session.

Using a User-Allocated MI_FPARAM Structure

The Fastpath look-up functions (in Table 9-5 on page 9-18) automatically allocate an **MI_FPARAM** structure and save a pointer to this structure in the function descriptor that they allocate. However, there are some cases in which you might want to allocate your own **MI_FPARAM** structure for the UDR that Fastpath executes.

The following DataBlade API functions support use of a user-allocated **MI_FPARAM** structure.

DataBlade API Function	Description
mi_fparam_allocate( )	Allocates a new **MI_FPARAM** structure
mi_fparam_copy( )	Copies an existing **MI_FPARAM** structure into a new **MI_FPARAM** structure
mi_fparam_free( )	Frees a user-allocated **MI_FPARAM** structure
mi_fp_usr_fparam( )	Determines whether a specified **MI_FPARAM** structure was allocated by the database server or by a UDR

Creating a User-Allocated MI_FPARAM Structure

The following DataBlade API functions create a user-allocated **MI_FPARAM** structure and return a pointer to this newly allocated structure:

- The **mi_fparam_allocate( )** function allocates an **MI_FPARAM** structure and returns a pointer to this newly allocated structure.

  This user-allocated **MI_FPARAM** structure holds the number of arguments that you specify to **mi_fparam_allocate( )**.

- The **mi_fparam_copy( )** function copies an existing **MI_FPARAM** structure to a new **MI_FPARAM** structure that the function allocates.

  The user-allocated **MI_FPARAM** structure holds the same number of arguments as the **MI_FPARAM** structure that **mi_fparam_copy( )** copied.

Both these functions are constructor functions for an **MI_FPARAM** structure. They allocate the user-allocated **MI_FPARAM** structure in the *current* memory duration. By default, the current memory duration is PER_ROUTINE. For calling a UDR with Fastpath, the PER_ROUTINE memory duration refers to the duration of the calling UDR, not the UDR that you call with Fastpath.

If you have changed the current memory duration with the **mi_switch_mem_duration( )** function, **mi_fparam_allocate( )** or **mi_fparam_copy( )** uses the current memory duration that **mi_switch_mem_duration( )** has specified for the **MI_FPARAM** structure that it allocates.

If the current memory duration is not acceptable for your use of the **MI_FPARAM** structure, call **mi_switch_mem_duration( )** with the desired memory duration *before* the call to **mi_fparam_allocate( )** or **mi_fparam_copy( )**. Keep in mind that when you call **mi_switch_mem_duration( )**, you change the current memory duration for all subsequent memory allocations, including those made by **mi_alloc( )**.

Using a User-Allocated MI_FPARAM Structure (Server)

For C UDRs, one of the primary uses of a user-allocated **MI_FPARAM** structure is for data type control for generic routines. If you are calling a *generic* UDR, one that handles many possible data types, you can set the arguments in the **MI_FPARAM** structure to the specific data type. Possible uses for generic UDRs include over collection types or over type hierarchies. You can set the **MI_FPARAM** structure to accept the correct parameter data types for a particular invocation of the routine. For example, you could pass an **employee_t** data type into a UDR that was defined with the supertype **person_t**.

Suppose you have a type hierarchy with **person_t** as the supertype and **employee_t** as a subtype of person. The **person_udr( )** UDR might be called with either the **person_t** or **employee_t** data type in the first two arguments. Suppose **person_udr( )** needs to use Fastpath to execute another UDR, named **person_udr2( )**, to handle some additional task. The following code fragment shows how the second and third arguments from **person_udr( )** are passed to **person_udr2( )**:

```
person_udr(person1, person2, data, fparam)
   pers_type *person1;
   pers_type *person2;
   mi_lvarchar *data;
   MI_FPARAM *fparam;
{
   MI_TYPEID *pers2_typeid;
   MI_FUNC_DESC *my_funcdesc;
   MI_FPARAM *my_fparam;
   mi_integer *myint_error;

   ...

   pers2_typeid = mi_fp_argtype(fparam, 1);
   my_funcdesc = mi_routine_get(conn, 0,
      "myfunc(person_t, lvarchar)");
   my_fparam = mi_fparam_get(conn, my_funcdesc);
   mi_fp_setargtype(my_fparam, 0, pers2_typeid);
   mi_routine_exec(conn, my_funcdesc, &myint_error, person2,
      data, my_fparam);
...
}
```

In the preceding code fragment, the first argument of **person_udr2( )** is set to have the same type as the second argument of **person_udr( )**, based on its **MI_FPARAM** structure. In this implementation, **person_udr2( )** needs the actual data type of the argument, not the supertype.

Tip: Another possible use of a user-allocated **MI_FPARAM** structure is in conjunction with per-session function descriptors. Per-session function descriptors are an advanced feature of the DataBlade API. For more information, see "Obtaining a Session-Duration Connection Descriptor" on page 7-13.

Passing a User-Allocated MI_FPARAM Structure

To pass the user-allocated **MI_FPARAM** structure to the Fastpath interface, specify it as the last argument of the argument list that you provide to **mi_routine_exec( )**. The following call to **mi_routine_exec( )** executes the **numeric_func( )** UDR (see Figure 9-7 on page 9-19) and specifies a user-allocated **MI_FPARAM** structure:

```
my_fparam = mi_fparam_allocate(2);
...
ret_val = mi_routine_exec(conn, fdesc, &error, 1, 2,
   my_fparam);
```

Freeing a User-Allocated MI_FPARAM

The database server automatically deallocates memory for the **MI_FPARAM** structures that it allocates for the function descriptor of a UDR. However, the database server does *not* deallocate any **MI_FPARAM** structure that you allocate. A user-defined **MI_FPARAM** structure has a memory duration of PER_COMMAND.

To conserve resources, use the **mi_fparam_free( )** function to deallocate explicitly the user-defined **MI_FPARAM** structure once your DataBlade API module no longer needs it. The **mi_fparam_free( )** function is the destructor function for a user-defined **MI_FPARAM** structure. It frees the **MI_FPARAM** structure and any resources that are associated with it.

Releasing Routine Resources

A function descriptor for a UDR has a memory duration of PER_COMMAND. Therefore, a function descriptor remains active until one of the following events occurs:

- The **mi_routine_end( )** function frees the function descriptor.
- The end of the current SQL command is reached.
- The **mi_close( )** function closes the current connection.

To conserve resources, use the **mi_routine_end( )** function to explicitly deallocate the function descriptor once your DataBlade API module no longer needs it. The **mi_routine_end( )** function is the destructor function for a function descriptor. It frees the function descriptor and any resources that are associated with it.

The following call to **mi_routine_end( )** explicitly releases the resources for the **fdesc** function descriptor that the code in Figure 9-7 on page 9-19 and Figure 9-10 on page 9-28 uses:

```
if ( mi_routine_end(conn, fdesc) != MI_OK )
   /* handle error */
   ...
```

Important: It is recommended that you explicitly deallocate function descriptors with **mi_routine_end( )** once you no longer need them. Otherwise, these function descriptors remain until the end of the associated SQL command.

──────────────── **Server Only** ────────────────

If your UDR accesses any session-duration function descriptors, these descriptors have a memory duration of PER_SESSION. Therefore, they remain active until the end of the session. You can explicitly deallocate them with **mi_routine_end( )**.

Warning: Session-duration function descriptors are an advanced feature of the DataBlade API. Do not use session-duration function descriptors unless regular function descriptors (PER_COMMAND) cannot perform the task you need done. For more information, see "Obtaining a Session-Duration Connection Descriptor" on page 7-13.

──────────────── **End of Server Only** ────────────────

Obtaining Trigger Execution Information and HDR Database Server Status

Trigger Information

You can create user-defined routines that are invoked in trigger action statements to obtain information about the triggers, triggering tables, views, statements, and the values of rows involved in the trigger actions. To create such a reporting routine, use the DataBlade APIs described below. Using these DataBlade APIs, you can write a general purpose user-defined routine that you can use to audit any table and any trigger event. You can use **mi_trigger functions** in the triggered action list of the FOR EACH ROW clause.

DataBlade API Function	Purpose
mi_trigger_event( )	Returns the value derived from an OR operation from the following events. These values are defined in the **milib.h** file. • MI_TRIGGER_NOT_IN_EVENT • MI_TRIGGER_INSERT_EVENT • MI_TRIGGER_DELETE_EVENT • MI_TRIGGER_UPDATE_EVENT • MI_TRIGGER_SELECT_EVENT • MI_TRIGGER_BEFORE_EVENT • MI_TRIGGER_AFTER_EVENT • MI_TRIGGER_FOREACH_EVENT • MI_TRIGGER_INSTEAD_EVENT • MI_TRIGGER_REMOTE_EVENT Each bit set in the returned value indicates the type of trigger currently executing. These definitions are in the public header file. The returned value is combined with these values to determine the current event. If the UDR is not currently executing, it returns MI_TRIGGER_NOT_IN_EVENT.
mi_trigger_get_new_row( )	Returns the new row being inserted or the updated value of the row. It returns NULL when called in other trigger action statements.
mi_trigger_get_old_row( )	Returns the row that was deleted or the value of the row before it was updated. It returns all the columns in the requested row, not just those columns into which data was explicitly inserted. Columns with default data are also returned. It returns NULL when called in other trigger action statements.
mi_trigger_level( )	Returns the nesting level of the current trigger. The values returned begin at 1 and increment by 1 for each nesting level, with a maximum of 61 levels.
mi_trigger_name( )	Returns the name of the currently executing trigger in the format *ownername.triggername*.
mi_trigger_tabname( )	Returns the triggering table or view name.

For the syntax of these functions and for more information about their purposes, see the *IBM Informix DataBlade API Function Reference*.

HDR Status Information

To enable UDRs to recognize a High-Availability Data Replication (HDR) configuration, you can use the following function.

mi_hdr_status()

The **mi_hdr_status()** function checks if the current database server is executing in an HDR environment and returns the following information:

- The server type, such as primary server, SD secondary server, RS secondary server, or HDR secondary server
- The HDR status
- The ability to update data on secondary servers

For a full description and the syntax of this function, see the *IBM Informix DataBlade API Function Reference*.

Chapter 10. Handling Exceptions and Events

In This Chapter

This chapter covers the following topics, which describe events and explain how to handle them in DataBlade API modules:

- DataBlade API event types
- Event-handling mechanisms
- Callback functions
- Database server exceptions
- State-transition events
- Client LIBMI errors

DataBlade API Event Types

For a DataBlade API module, an *event* is a communication from IBM Informix software that indicates the existence of some predefined condition, usually the occurrence of an exception (warning or error). The DataBlade API represents an event as one of the enumerated values of the **MI_EVENT_TYPE** data type. Table 10-1 shows the **MI_EVENT_TYPE** values that the DataBlade API supports.

Table 10-1. DataBlade API Event Types

Event Type	Occurrence	Where Event Occurs	
		C UDR	**Client LIBMI Application**
MI_Exception	Raised when the database server generates an exception (a warning or an error)	Yes	Yes
MI_EVENT_SAVEPOINT	Raised after the cursor flushes within an explicit transaction	Yes	No
MI_EVENT_COMMIT_ABORT	Raised when the database server reaches the end of a transaction in which work was done	Yes	No
MI_EVENT_END_XACT	Raised when the database server reaches the end of a transaction, or if a hold cursor is involved, raised only after the hold cursor is closed It is preferable to register the MI_EVENT_COMMIT_ABORT callback.	Yes	No
MI_EVENT_END_STMT	Raised when the database server completes the execution of the current SQL statement, or for statements associated with a cursor, raised when the cursor is closed	Yes	No
MI_EVENT_POST_XACT	Raised just after the database commits or rolls back a transaction if work was done in the transaction or if an MI_EVENT_END_XACT event was raised	Yes	No
MI_EVENT_END_SESSION	Raised when the database server reaches the end of the current session	Yes	No
MI_Xact_State_Change	Raised when the database server starts or ends a transaction, whether the transaction contains one or multiple SQL statements	No	Yes
MI_Client_Library_Error	Raised when the client LIBMI library encounters an error	No	Yes

Important: MI_All_Events is a deprecated event type and is not listed with the **MI_EVENT_TYPE** values. Using the MI_All_Events event type is not recommended.

The **milib.h** header file defines the **MI_EVENT_TYPE** data type and its event types. The DataBlade API event types can be grouped as follows.

Event Type	Event Group	More Information
MI_Exception	Database server exceptions	"Database Server Exceptions" on page 10-20

Event Type	Event Group	More Information
MI_EVENT_SAVEPOINT	State-transition events	"State-Transition Events" on page 10-49
MI_EVENT_COMMIT_ABORT		
MI_EVENT_POST_XACT		
MI_EVENT_END_STMT		
MI_EVENT_END_XACT		
MI_EVENT_END_SESSION		
MI_Xact_State_Change		
MI_Client_Library_Error	Client LIBMI errors	"Client LIBMI Errors" on page 10-55

Event-Handling Mechanisms

An *event-handling mechanism* provides a way for one DataBlade API module to inform another module (or another part of the same module) that an event has occurred during execution of a function. An event-handling mechanism has two parts:

- A function that *throws* an event

  A function in a DataBlade API module might encounter a condition that it cannot handle. Events represent many common conditions (see Table 10-1 on page 10-2). When a module encounters one of these conditions, it can throw the associated event to indicate that some event handling needs to be performed.

- A function that *catches* an event

  A special function, called a *callback function*, is invoked when its associated event occurs. The callback function can perform the appropriate actions to handle or recover from this event.

This division of event-handling responsibility enables you to put common event-handling code for a particular condition in a single location, in a callback function. Any DataBlade API module that requires the associated event handling can then *register* this callback function.

When an event occurs, the DataBlade API performs event handling based on whether it finds a registered callback that can catch (or handle) the event, as follows:

- If a registered callback exists for the event, the DataBlade API invokes this callback.
- If no registered callback exists for the event, the DataBlade API takes the appropriate default behavior.

Invoking a Callback

A callback function (or just *callback*) is a function that you write to handle a particular event. The DataBlade API provides the following functions to handle invocation of a callback function.

Callback Task	DataBlade API Function
Register a callback	**mi_register_callback( )**, **mi_unregister_callback( )**

Callback Task	DataBlade API Function
Enable a callback	**mi_enable_callback( )**
Disable a callback	**mi_disable_callback( )**
Retrieve a pointer to the callback function	**mi_retrieve_callback( )**

For the DataBlade API to invoke a callback when the associated event occurs, the following conditions must be met in the DataBlade API module:

- The module must register the callback for the event on the current connection.

  The **mi_register_callback( )** function registers a callback for a particular event. The DataBlade API module that requires the event handling must register the callback.

- The registered callback must be enabled.

  It is possible to disable a registered callback to suspend its invocation. The **mi_enable_callback( )** function enables a previously disabled callback.

In addition, a module can save a registered callback and restore it at a later time.

Registering a Callback

For a callback to execute when its associated event occurs, you must *register* it with the **mi_register_callback( )** function. When you register a callback, you take the following actions:

- Associate the callback function with the event it is to catch.
- Provide arguments for the callback parameters.

The call to the **mi_register_callback( )** function must occur *before* the statement for which you want the exception handling. If you register more than one callback to handle a particular event, all registered callbacks execute when the event occurs. The order in which these callbacks execute is not predetermined.

Tip: You do not need to register a callback function in the database with the CREATE FUNCTION statement. You need to register the callback only with the **mi_register_callback( )** function.

The **mi_register_callback( )** function requires the following arguments.

Argument Type	Description	More Information
MI_CONNECTION *	A pointer to the connection on which the callback is to be registered. It might be NULL if the connection is undefined, as is the case with some client LIBMI errors and state-transition events.	"Connection Descriptor" on page 10-6
MI_EVENT_TYPE	The event type that the callback handles	"Types of Callbacks" on page 10-5
MI_CALLBACK_FUNC	A pointer to the callback function to invoke when the specified event occurs	"Callback-Function Pointer" on page 10-7

Argument Type	Description	More Information
void *	A pointer to the user data, which is passed to the callback function when the specified event occurs. It can be used to pass additional information to and from the callback.	"Returning Error Information to the Caller" on page 10-32 "Managing Memory Allocations" on page 10-52
MI_CALLBACK_HANDLE *	Must be NULL	"Callback Handle" on page 10-7

These arguments initialize many of the parameters of the callback function. For more information, see Figure 10-2 on page 10-16.

When **mi_register_callback( )** registers the callback, the function returns a callback handle for the callback. For more information, see "Callback Handle" on page 10-7.

If a callback is *not* registered when its event occurs, the DataBlade API takes the default behavior. For more information, see "Using Default Behavior" on page 10-11.

By default, a callback remains registered until the end of the connection. For more information, see "Registration Duration" on page 10-7.

Types of Callbacks: The second argument of the **mi_register_callback( )** function is the event type. The DataBlade API supports a type of callback for each event type. The following table lists the types of callbacks that the DataBlade API supports and the events that invoke them.

Callback Type	Event Type	More Information
Exception callback	MI_Exception	"Database Server Exceptions" on page 10-20
State-transition callbacks:		"State-Transition Events" on page 10-49
Savepoint callback	MI_EVENT_SAVEPOINT	
Commit-abort callback	MI_EVENT_COMMIT_ABORT	
Post-transaction callback	MI_EVENT_POST_XACT	
End-of-statement callback	MI_EVENT_END_STMT	
End-of-transaction callback	MI_EVENT_END_XACT	
End-of-session callback	MI_EVENT_END_SESSION	
State-change callback	MI_Xact_State_Change	
Client LIBMI callback	MI_Client_Library_Error	"Client LIBMI Errors" on page 10-55

For a general introduction on how to write a callback function, see "Callback Functions" on page 10-12.

Connection Descriptor: The first argument of the **mi_register_callback( )** function is a connection descriptor. This connection descriptor can be either a NULL-valued pointer or a pointer to a valid connection. The valid value depends on whether the calling module is a C user-defined routine (UDR) or a client LIBMI application.

Server Only

For a UDR, the connection descriptor must be a NULL-valued pointer when you register callbacks for the following state-transition events:

- MI_EVENT_SAVEPOINT
- MI_EVENT_COMMIT_ABORT
- MI_EVENT_POST_XACT
- MI_EVENT_END_STMT
- MI_EVENT_END_XACT
- MI_EVENT_END_SESSION

For example, the following call to **mi_register_callback( )** specifies a connection descriptor of NULL to register the **endxact_callback( )** end-of-transaction callback, (which "State Transitions in a C UDR (Server)" on page 10-51 defines):

```
cback_hndl = mi_register_callback(NULL, MI_EVENT_END_XACT,
    endxact_callback, NULL, NULL);
```

The **mi_register_callback( )** function requires a valid connection descriptor for the MI_Exception event type. For example, the following code fragment registers the **handle_errs( )** callback:

```
conn = mi_open(NULL, NULL, NULL);

if ( mi_register_callback(conn, MI_Exception, handle_errs,
     NULL, NULL) == NULL )
   /* handle error */
```

In the preceding code fragment, the **mi_register_callback( )** function specifies a valid connection descriptor, which the **mi_open( )** function has initialized.

For the MI_Exception event, you can also provide a NULL-valued pointer as a connection descriptor. In this case, the DataBlade API looks for callbacks registered by the function that called the C UDR. If no callback exists for this calling function, the DataBlade API continues up the calling hierarchy looking for registered callbacks until it reaches the client application (which initially invoked the UDR). This hierarchy of callbacks takes advantage of the calling hierarchy.

For further information on how to register an exception callback, see "Exceptions in a C UDR (Server)" on page 10-25.

End of Server Only

Client Only

For a client LIBMI application, you must provide a valid connection descriptor to **mi_register_callback( )** to register callbacks for the following event types:

- MI_Exception
- MI_Xact_State_Change
- MI_Client_Library_Error

If the connection descriptor is not valid, the **mi_register_callback( )** function raises an exception.

Callback-Function Pointer: The third argument of the **mi_register_callback( )** function is a *callback-function pointer*. The DataBlade API stores a pointer to the location of a callback function in the **MI_CALLBACK_FUNC** data type. In the **mi_register_callback( )** function, you can specify the callback function in either of the following ways:

- The name of the function

    When you specify the function name as the third argument, **mi_register_callback( )** creates a callback-function pointer from the function name.

    The sample callback registrations in "Connection Descriptor" on page 10-6 pass the name of the callback as the third argument of **mi_register_callback( )**. Both of these registrations are the first time that the callback is registered within the module.

- A pointer to an **MI_CALLBACK_FUNC** variable

    If **mi_register_callback( )** has already initialized a callback-function pointer, you can specify this pointer as the third argument of **mi_register_callback( )**.

    In the code fragment in "Retrieving a Callback Function" on page 10-8, the second registration of the **initial_cback( )** function passes a pointer to an **MI_CALLBACK_FUNC** variable (**&initial_cbptr**) as the third argument of **mi_register_callback( )**.

Callback Handle: The **mi_register_callback( )** function returns a *callback handle*, which accesses a registered callback within a DataBlade API module. A callback handle has the **MI_CALLBACK_HANDLE** data type. Use a callback handle to identify a callback for the following tasks.

Callback Task	DataBlade API Function
Enable a callback	**mi_enable_callback( )**
Disable a callback	**mi_disable_callback( )**
Retrieve a pointer to a callback function	**mi_retrieve_callback( )**
Unregister a callback	**mi_unregister_callback( )**

Tip: The last argument of the **mi_register_callback( )** function is also a callback handle, but this argument is reserved for future use and must currently be a NULL-valued pointer.

Registration Duration: The registration of a callback survives until one of the following conditions is met:

- The connection on which the callback is registered closes (either the UDR exits or the **mi_close( )** function executes).
- The DataBlade API calls the callback, which happens for state-transition callbacks when one of the following events occurs:
    - MI_EVENT_SAVEPOINT
    - MI_EVENT_COMMIT_ABORT
    - MI_EVENT_POST_XACT
    - MI_EVENT_END_STMT

- MI_EVENT_END_XACT
- MI_EVENT_END_SESSION

- You explicitly unregister the callback with the **mi_unregister_callback( )** function.

Enabling and Disabling a Callback

A callback must be *enabled* for the database server to invoke it. The **mi_register_callback( )** function automatically enables the callback that it registers. You can explicitly disable a registered callback with the **mi_disable_callback( )** function. When you disable a callback, you suspend its invocation. You can later explicitly reenable it with the **mi_enable_callback( )** function.

Tip: If you want to reuse a callback, it is usually less resource intensive to disable and reenable the callback than to unregister and reregister it.

Both the **mi_enable_callback( )** and **mi_disable_callback( )** functions take the following arguments:

- A connection descriptor

  The connection descriptor must have the same value that the **mi_register_callback( )** statement used when it registered the callback. For more information, see "Connection Descriptor" on page 10-6.

- An **MI_EVENT_TYPE** value

  This argument identifies the event type that the callback handles. For more information, see Table 10-1 on page 10-2.

- A callback handle for a registered callback

  The **mi_register_callback( )** function returns a callback handle when it successfully registers a callback. This handle identifies the callback to the **mi_enable_callback( )** or **mi_disable_callback( )** function. For more information, see "Callback Handle" on page 10-7.

Retrieving a Callback Function

The **mi_retrieve_callback( )** function returns a callback-function pointer (**MI_CALLBACK_FUNC**) when you pass in a callback handle (**MI_CALLBACK_HANDLE**). This function is useful when a DataBlade API module needs to change temporarily the callback that is registered for a particular event.

To change a registered callback temporarily:

1. Register the initial callback with **mi_register_callback( )**.

   The **mi_register_callback( )** function returns a callback handle for the callback that it receives as its third argument.

2. Perform tasks that require the event handling of the initial callback.

3. Obtain the callback-function pointer for the initial callback with **mi_retrieve_callback( )**.

   Pass the callback handle for the initial callback as an argument to the **mi_retrieve_callback( )** function. The function returns a callback-function pointer, which saves the location of the registered initial callback.

   The initial callback should be unregistered.

4. Register the temporary callback with **mi_register_callback( )**.

   This call to **mi_register_callback( )** overwrites the previous callback that was registered for the event and returns a callback handle for the temporary callback.

5. Perform the tasks that require the event handling of the temporary callback.
6. Restore the initial callback with **mi_register_callback( )**.

    Pass the saved callback-function pointer (step 3) of the initial callback as the third argument of **mi_register_callback( )**. The function returns a new callback handle for the initial callback.

A temporary callback is useful when C UDRs are nested and some inner function wants to trap an event type in its own way instead of in the way that the outer function provides.

For example, suppose the **func1( )** function specifies the **func1_callback( )** function to handle *event_type* events and then calls the **func2( )** function, as follows:

```
func1(...)
{
   MI_CALLBACK_HANDLE *cback_hndl;
   ...
   /* Set callback for event_type event type in func1( ) */
   cback_hndl = mi_register_callback(conn, event_type,
      func1_callback, ...)

   /* do some stuff */
   ...
   /* call func2( ), which "inherits" func1( ) callback */
   func2( )
   ...
}

func2( )
{
   MI_CALLBACK_HANDLE *cback_hndl;
   MI_CALLBACK_FUNC *old_callback;

   /* Save func1( ) callback in 'old_callback' */
   if ( mi_retrieve_callback(conn, event_type, cback_hndl,
         old_callback, NULL) == MI_ERROR )
      /* handle error */

   /* Set up func2( ) callback */
   mi_unregister_callback(conn, event_type, cback_hndl);
   mi_register_callback(conn, event_type, func2_callback, ...);

   /* do some other stuff */
   ...

   /* restore func1( ) callback */
   cback_hndl = mi_register_callback(conn, event_type,
      old_callback, ...)
   ...
}
```

By default, the database server uses the **func1_callback( )** callback to handle any *event_type* events that occur during execution of **func2( )**. For the **func2( )** routine to trap an *event_type* event in its own way, the routine must save the **func1_callback( )** callback and then register its own callback for the *event_type* event type.

In the preceding code, the **func2( )** function performs the following tasks:

1. Saves the **func1_callback( )** from the **func1( )** function

The **func2( )** function passes in the **func1_callback( )** callback handle (*cback_hndl*) to the **mi_retrieve_callback( )** function, which puts the **MI_CALLBACK_FUNC** handle for **func1_callback( )** into the **old_callback** argument.

2. Registers its own callback, **func2_callback( )**

   The **mi_register_callback( )** function registers the **func2_callback( )** function for the *event_type* event type.

3. Performs its own work

   Any *event_type* event that occurs during this work causes the database server to invoke the **func2_callback( )** function.

4. Restores the callback of the **func1( )** function

   The **func2( )** function uses **mi_register_callback( )** again, this time to restore the **func1_callback( )** as the callback for the *event_type* event type.

```
┌──────────────────────── Server Only ────────────────────────┐
```

For example, suppose the UDR **func1( )** specifies the **initial_cback( )** function to handle the MI_Exception event type, but the UDR requires the **tmp_cback( )** callback for MI_Exception events during a portion of its execution. The following code fragment shows the use of **mi_retrieve_callback( )** and **mi_register_callback( )** to save the **initial_cback( )** callback, to use the **tmp_cback( )** callback temporarily, and then to restore **initial_cback( )**:

```
func1( )
{
  MI_CONNECTION *conn;
  MI_CALLBACK_HANDLE *initial_hndl, *tmp_hndl;
  MI_CALLBACK_FUNC initial_cbptr;

  conn = mi_open(NULL, NULL, NULL);

  /* Register the initial callback (register #1). */
  initial_hndl = mi_register(conn, MI_Exception,
     initial_cback, NULL, NULL);

  /* Do tasks that require initial_cback( ) as callback. */
  ...
  /* Retrieve the current callback-function pointer on
   * this connection into initial_cbptr. Pass in the
   * callback handle of initial_cback( ).
   */
  mi_retrieve_callback(conn, MI_Exception, initial_hndl,
     &initial_cbptr, NULL);
  /* Register the temporary callback (register #2).
   * Callback handle for initial callback is overwritten.
   */
  tmp_hndl = mi_register_callback(conn, MI_Exception,
     tmp_cback, NULL, NULL);

  /* Do tasks that require tmp_cback( ) as callback. */
  ...
  /* Restore initial callback (register #3) */
  initial_hndl = mi_register(conn, MI_Exception,
     &initial_cbptr, NULL, NULL);

  /* Continue with tasks that require initial_cback( ) as
   * callback.
   */
  ...
}
```

Using Default Behavior

If no callback is registered for a particular event, the DataBlade API uses its default behavior when this event occurs. The default event handling depends on whether the event occurs in a C UDR or in a client LIBMI application.

Default Behavior in a C UDR (Server)

If an exception occurs during the execution of the UDR and the UDR does *not* register any callback to handle this event, the DataBlade API takes one of the following default actions.

Exception Type	Default Behavior
MI_Exception	An unhandled MI_MESSAGE exception does *not* halt execution of the current statement. The DataBlade API passes the warning to the client application, and processing continues at the next statement of the UDR.
	An unhandled MI_EXCEPTION exception aborts execution of the current statement in the UDR. The DataBlade API returns control to the calling module.
MI_Xact_State_Change	When received in a UDR, an MI_Xact_State_Change exception is treated the same as an MI_EVENT_END_XACT event.

If a UDR does not register a callback for the MI_Exception event whose exception level is MI_EXCEPTION (a runtime error), the DataBlade API aborts the UDR and returns control to the calling module, which might have been either of the following modules:

- A client application that called the UDR in an SQL statement
- Another UDR that called the UDR that encountered the runtime error

The calling module might have a registered callback (or some other method) to handle the exception. To prevent database runtime errors from aborting a UDR, use the **mi_register_callback( )** function to register callbacks in the UDR. For more information, see "Exceptions in a C UDR (Server)" on page 10-25.

Important: Programming errors do not cause execution of callbacks. If a UDR contains a serious programming error (such as a segmentation fault), execution jumps out of the routine and back to the routine manager. The routine manager attempts to make an entry in the database server message log file (**online.log** by default).

Default Behavior in Client LIBMI Applications

If an exception occurs during the execution of a client LIBMI application and the application does *not* register any callback to handle the exception, the client LIBMI takes one of the following default actions.

Exception Type	Default Behavior
MI_Exception	The client LIBMI executes the system-default callback. The **mi_default_callback( )** function implements this system-default callback.
	An unhandled MI_MESSAGE does *not* halt execution of the current statement. The DataBlade API passes the warning to the client LIBMI application, and processing continues at the next statement of this client application.
	An unhandled MI_EXCEPTION aborts execution of the current statement in the client LIBMI application. The DataBlade API passes the error to the client LIBMI application and returns control to this client application.
MI_Xact_State_Change	An unhandled MI_Xact_State_Change does *not* halt execution of the current statement. Processing continues at the next statement of the client LIBMI application.
MI_Client_Library_Error	The client LIBMI executes the system-default callback. The **mi_default_callback( )** function implements this system-default callback.

──────────────────── **UNIX/Linux Only** ────────────────────

On UNIX or Linux, the system-default callback causes the client LIBMI application to send an error or warning message to **stderr** in response to an unhandled exception.

──────────────── **End of UNIX/Linux Only** ────────────────

──────────────────── **Windows Only** ────────────────────

On Windows, the system-default callback causes the client LIBMI to display an error or warning message in a Windows message box in response to an unhandled exception.

──────────────── **End of Windows Only** ────────────────

To prevent the execution of the system-default callback, use the **mi_register_callback( )** function to register callbacks in the client LIBMI application. For more information, see "Exceptions in a Client LIBMI Application (Client)" on page 10-31 and "Client LIBMI Errors" on page 10-55.

Callback Functions

To catch or handle an event, you create a C function called a *callback function*. In your DataBlade API module, you can register callback functions to handle recovery from events. The DataBlade API invokes a registered (and enabled) callback when the event associated with the callback occurs.

This section describes how to declare a callback function and how to write the body of a callback function. For more information on how to register callbacks, see "Registering a Callback" on page 10-4.

Declaring a Callback Function

To declare a callback function, you provide the following information:

--- **Windows 2000 Only** ---

- The MI_CALLBACK_STATUS return type

--- **End of Windows 2000 Only** ---

- The optional MI_PROC_CALLBACK modifier
- Parameter declarations

Figure 10-1 shows the declaration of a callback function named **myhandler( )** for use in a UDR.

Callback Declaration

```
MI_CALLBACK_STATUS MI_PROC_CALLBACK
myhandler(event_type, conn, event_data, user_data)
```

Figure 10-1. A Sample Callback Declaration

Return Value of a Callback Function

When a callback function completes execution, it returns any return value that it might have to the DataBlade API, which invoked it. The data type of the callback return value depends on whether a UDR or a client LIBMI application triggered the callback.

--- **Server Only** ---

When a UDR causes a callback function to be invoked, the DataBlade API expects the callback-function return value to be one of the enumerated values of the **MI_CALLBACK_STATUS** data type. The **MI_CALLBACK_STATUS** values indicate how to continue handling the event once the callback completes.
Table 10-2 shows the valid values for the **MI_CALLBACK_STATUS** return type.

Table 10-2. MI_CALLBACK_STATUS Return-Status Values

Return-Status Values	Description
MI_CB_EXC_HANDLED	Only an exception callback can return this status value. When the callback completes, the DataBlade API returns control to the first statement after the statement that raised the exception event. This return status indicates that the callback has successfully handled the event and the DataBlade API does not need to continue with event handling. Therefore, the DataBlade API does not abort the statement that invoked the callback.

Table 10-2. MI_CALLBACK_STATUS Return-Status Values (continued)

Return-Status Values	Description
MI_CB_CONTINUE	This is the only status value that a callback other than an exception callback can return. If a nonexception callback returns this value, the database server continues processing after the callback completes.
	When an exception callback completes, the DataBlade API continues to look for registered callbacks that handle the event:
	• Callbacks registered for the same event (on the same connection) and at the same level in the calling sequence
	• Callbacks registered for the same event (on the same connection) in a higher level of the calling sequence
	When an exception callback returns this value to the highest-level function in a calling sequence and no other registered callback exists, the DataBlade API aborts the UDR and any current transaction.

The **milib.h** header file defines MI_CALLBACK_STATUS and its return-status values.

The end-of-transaction callback on page 10-51 shows use of the MI_CB_CONTINUE status. For information on the use of these return codes in exception callbacks, see "Determining How to Handle the Exception" on page 10-29.

——————————— **End of Server Only** ———————————

——————————— **Client Only** ———————————

When a client LIBMI application causes a callback to be invoked, the DataBlade API does not expect the callback to return a status value. The client LIBMI ignores any return value from a callback that a client LIBMI application registers. Therefore, any such callbacks can return **void**.

In effect, the client LIBMI always assumes a MI_CB_EXC_HANDLED return status from a callback. The client LIBMI returns control to the first statement after the one that threw the event. The client LIBMI application must include code that decides how to proceed based on the failure.

If a callback returns MI_CB_CONTINUE, the client LIBMI ignores the return code because this return value does not have a meaning within a client application. Within a C UDR, you can pass an exception up to a higher level in the calling sequence because the routine executes in the context of the database server. However, a client LIBMI application does *not* execute in the context of the database server. Therefore, it cannot assume this general exception-handling mechanism.

For an example of a callback that a client LIBMI application registers, see the **clntexcpt_callback( )** function in "Returning Error Information to the Caller" on page 10-32.

——————————— **End of Client Only** ———————————

MI_PROC_CALLBACK Modifier (Windows)

The MI_PROC_CALLBACK modifier on a callback definition is required for callbacks that execute with Windows applications. For all other operating systems, this modifier is optional. To make callbacks portable between operating systems, include the MI_PROC_CALLBACK modifier in your callback declaration.

The MI_PROC_CALLBACK modifier follows the callback return type and precedes the callback name. Figure 10-1 on page 10-13 shows the location of the MI_PROC_CALLBACK modifier in the declaration of the **myhandler( )** callback.1fs

Callback-Function Parameters

A callback function takes the following parameters.

Argument Type	Description	Initialized by mi_register_callback( )?
MI_EVENT_TYPE	The event type that triggers the callback	Yes
MI_CONNECTION *	A pointer to the connection on which the event occurred If the connection is undefined, it might be NULL, as is the case with some client-library errors and state-transition events.	Yes
void *	A pointer to an event-type structure that holds event information For example, if the event is an MI_Exception event, the DataBlade API passes in an **MI_ERROR_DESC** structure, which holds the exception level and the message text.	No The DataBlade API sets this pointer to the associated event structure when it invokes the callback.
void *	A pointer to any user data, which you can use to pass any additional information to and from the callback	Yes

When you register a callback with the **mi_register_callback( )** function, you provide arguments for most parameters of the callback. Figure 10-2 shows how a call to **mi_register_callback( )** provides the arguments that initialize the parameters of the **myhandler( )** callback.

Callback Declaration

```
MI_CALLBACK_STATUS MI_PROC_CALLBACK
myhandler(event_type, conn, event_data, user_data)
```

```
mi_register_callback(conn, MI_Exception, myhandler, NULL, NULL);
```

Callback Registration
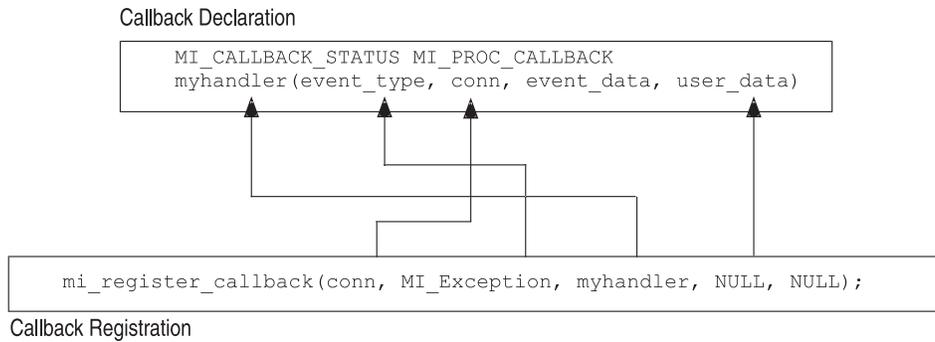
Figure 10-2. Initializing a Callback

The only callback parameter that the **mi_register_callback( )** call does *not* initialize is the event-type structure (the *event_data* parameter in Figure 10-2). For more information about event-type structures, see "Event Information" on page 10-17.

Writing a Callback Function

Within the body of a callback function, you provide the code that handles a particular event or events. Only certain tasks are valid within a callback. When a callback function is invoked for an event, the DataBlade API passes information about the event to the callback.

Restrictions on Content

A callback can call a DataBlade API function to perform its task. Callbacks often clean up resources with such functions as **mi_free( )**, **mi_close( )**, and **mi_lo_spec_free( )**. The MI_EXCEPTION, MI_END_SESSION, and MI_EVENT_POST_XACT callbacks cannot perform the following tasks:

- Execute SQL statements
- Raise database server exceptions
- Register other callbacks

Server Only

The following types of callbacks are not subject to the same restrictions as other callbacks:

- Commit-abort callback
- End-of-statement callback
- End-of-transaction callback
- Savepoint callback

Specifically, these callbacks can raise an exception and they can register their own exception callbacks. If an end-of-transaction or end-of-statement callback issues a call to a DataBlade API function that generates an exception, the action taken depends on whether the callback has registered its own exception callback, as follows:

- If the callback has *not* registered any exception callback, any failure of a DataBlade API function results in the return of the MI_ERROR or NULL failure code from the DataBlade API function.

  The callback must check for possible failure and take any necessary exception-handling tasks.

- If the callback has registered an exception callback, control is thrown to the exception callback.

  For information on how an end-of-event callback can handle exceptions, see "State Transitions in a C UDR (Server)" on page 10-51.
- An MI_EVENT_POST_XACT callback cannot raise an error because the transaction has already been committed or rolled back.

--------------------- End of Server Only ---------------------

Event Information

When a callback function is invoked for a particular event, the DataBlade API passes an *event-type structure* as the third parameter of this function. This event-type structure contains information about the event that triggered the callback. The DataBlade API stores event information in one of the following structures based on the event type:

- Exceptions and errors are stored in *error descriptors*.
- State transitions are stored in *transition descriptors*.

The following table shows the event types and the corresponding event-type structures that describe them.

Event-Type Structure	Event Type
Error descriptor (**MI_ERROR_DESC**)	MI_Exception MI_Client_Library_Error
Transition descriptor (**MI_TRANSITION_DESC**)	MI_EVENT_SAVEPOINT MI_EVENT_COMMIT_ABORT MI_EVENT_POST_XACT MI_EVENT_END_STMT MI_EVENT_END_XACT MI_EVENT_END_SESSION MI_Xact_State_Change

The **milib.h** header file defines the **MI_ERROR_DESC** and **MI_TRANSITION_DESC** structures.

Using an Error Descriptor: The DataBlade API stores information about exceptions and errors in an *error descriptor*. An error descriptor is an **MI_ERROR_DESC** structure. It holds information for the MI_Exception and MI_Client_Library_Error event types. The following table summarizes the memory operations for an error descriptor.

Memory Duration	Memory Operation	Function Name
Current memory duration	Constructor	**mi_error_desc_copy( )**
	Destructor	**mi_error_desc_destroy( )**

When an MI_Exception or MI_Client_Library_Error event occurs, the DataBlade API invokes the appropriate callback. To this callback, the DataBlade API passes an initialized error descriptor as the third callback argument. The error descriptor contains information about the MI_Exception or MI_Client_Library_Error event. Within the callback, use the accessor functions in Table 10-3 on page 10-18 to obtain the error information from the error descriptor.

Accessing an Error Descriptor: The error descriptor is an opaque structure. You must use the DataBlade API functions in Table 10-3 to access information within it.

Table 10-3. Accessor Functions for an Error Descriptor

Error-Descriptor Information	Description	DataBlade API Function
Error or warning message	The text of the error or warning message	**mi_errmsg( )**
Exception or error level	An MI_Exception event has an exception level that indicates the type of exception that has occurred: a warning (MI_WARNING) or a runtime error (MI_EXCEPTION) An MI_Client_Library_Error event has an error level to indicate the type of error that has occurred. For a list of possible client LIBMI error levels, see "Client LIBMI Errors" on page 10-55.	**mi_error_level( )**
SQLSTATE value	A five-character status value that is compliant with ANSI and X/Open standards. For more information, see "SQLSTATE Status Value" on page 10-22.	**mi_error_sql_state( )**
SQLCODE value	An Informix-specific status value that contains an integer value. For more information, see "SQLCODE Status Value" on page 10-24.	**mi_error_sqlcode( )**

Each of the DataBlade API functions in Table 10-3 requires that you pass in a pointer to a valid error descriptor.

Important: The **MI_ERROR_DESC** structure is an opaque structure to DataBlade API modules. Do not access its internal fields directly. The internal structure of **MI_ERROR_DESC** may change in future releases. Therefore, to create portable code, always use the accessor functions in Table 10-3 to obtain values from this structure.

For a sample callback that obtains information from an error descriptor, see the **excpt_callback2( )** function in "Associating with a Callback" on page 10-33.

Creating a Copy of an Error Descriptor: The DataBlade API passes the error descriptor as an argument to the callback. Therefore, the DataBlade API allocates memory for the error descriptor when it invokes the callback and deallocates this memory when the callback exits. To preserve the error information for the calling routine, you can create a user copy of the error descriptor within the callback.

The following DataBlade API functions facilitate an error-descriptor copy.

DataBlade API Function	Description
mi_error_desc_copy( )	Allocates memory for a user copy of a specified error descriptor and returns a pointer to this user copy
mi_error_desc_is_copy( )	Determines whether the specified error descriptor is a user copy

DataBlade API Function	Description
mi_error_desc_destroy( )	Frees memory for a specified user copy of an error descriptor (which was allocated with **mi_error_desc_copy( )**)

Using a Transition Descriptor: The *transition descriptor,* MI_TRANSITION_DESC, stores information about a transition in the processing state of the database server. It holds information for all state-transition events:

Server Only

- MI_EVENT_SAVEPOINT
- MI_EVENT_COMMIT_ABORT
- MI_EVENT_POST_XACT
- MI_EVENT_END_STMT
- MI_EVENT_END_XACT
- MI_EVENT_END_SESSION

End of Server Only

- MI_Xact_State_Change

The **milib.h** header file defines the MI_TRANSITION_DESC structure.

When a state transition event occurs, the DataBlade API invokes the appropriate callback. To this callback, the DataBlade API passes an initialized transition descriptor as the third callback argument. The transition descriptor contains the transition type that initiated the state-transition event. To obtain the transition type from the descriptor, use the **mi_transition_type( )** function. This function returns a value of type **MI_TRANSITION_TYPE** to indicate the transition type of the event that occurred. For a list of valid **MI_TRANSITION_TYPE** values, see "Understanding State-Transition Events" on page 10-49.

Important: The **MI_TRANSITION_DESC** structure is an opaque structure to DataBlade API modules. Do not access its internal fields directly. The internal structure of **MI_TRANSITION_DESC** may change in future releases. Therefore, to create portable code, always use the **mi_transition_type( )** accessor function to obtain the transition type from this structure.

The following code fragment uses the **mi_transition_type( )** function to determine which action to take when it receives a state-transition event:

```
MI_TRANSITION_DESC *event_data;
MI_TRANSITION_TYPE trans_type;
mi_string s[30];
...
trans_type = mi_transition_type(event_data);
switch ( trans_type )
   {
   case MI_BEGIN: /* client LIBMI apps only */
      s = "Transaction started.";
      break;
   case MI_NORMAL_END:
      s = "Successful event";
      break;
   case MI_ABORT_END:
      s = "Event failed and rolled back,";
```

```
        break;
    default:
       s = "Unknown transition type";
       break;
    }
fprintf(stderr, "%s\n", s);
```

Database Server Exceptions

When the database server or a UDR raises a database server exception, the
DataBlade API invokes any callbacks that are registered for the exception. This
section provides information about exception handling in DataBlade API modules:

- An explanation of database server exceptions
- How to handle a database server exception in a UDR and in a client LIBMI
  application
- How to return error information to the calling code
- How to handle errors that generate multiple exceptions
- How to explicitly raise a database server exception

Understanding Database Server Exceptions

A *database server exception* is an unexpected condition that occurs within the
database server. A database server exception can occur in any of the following
tasks:

- Within an SQL statement

  The **mi_exec( )** and **mi_exec_prepared_statement( )** functions execute SQL
  statements. An exception can occur when the database server executes an SQL
  statement.

- Within a UDR

  The **mi_routine_exec( )** function executes UDRs through the Fastpath interface.
  An exception can occur when this UDR executes.

- By the DataBlade API function, **mi_db_error_raise( )**

  The **mi_db_error_raise( )** function can explicitly raise a database server
  exception within a DataBlade API module. For more information, see "Raising
  an Exception" on page 10-40.

- Within the execution of some other DataBlade API function

  Other functions in the DataBlade API might raise exceptions when they execute.

When the database server encounters a database server exception, it raises the
MI_Exception event.

Warnings and Errors

The MI_Exception event indicates which of the following status conditions has
caused the database server exception:

- A *warning* is a condition that does *not* prevent successful execution of an SQL
  statement; however, the effect of the statement is limited and the statement
  might not produce the expected results.

- A *runtime error* (or failure) indicates that the SQL statement or DataBlade API
  function did not execute successfully and it made no change to the database.

  Runtime errors can occur at the following levels:

  – Hardware errors include controller failure, bad sector on disk, and so on.

  – Kernel errors include file-table overflow, insufficient semaphores, and so on.

- Access-method errors include duplicated index keys, SQL null inserted into non-null columns, and so on.
- Parser errors include invalid syntax, unknown objects, invalid statements, and so on.
- DataBlade API library errors are usually caused by invalid arguments.

The following list describes the most common DataBlade API library errors:
- The DataBlade API might not have been initialized.
- One of the DataBlade API initialization functions must be the first DataBlade API call in the module. For more information, see "Initializing the DataBlade API" on page 7-17.
- A function that passes a connection descriptor (**MI_CONNECTION**) passes an invalid connection, one that was closed or dropped.
- A function that passes a row descriptor (MI_ROW_DESC) can pass an invalid row descriptor.
- A function that passes a row structure (MI_ROW) passes an invalid row.
- A function that passes a column name passes a column name that does not exist in the object or objects being accessed.
- A function that passes a column number passes a column number that is out of range (greater or less than the number of columns in the object).
- A function that passes an event type (**MI_EVENT_TYPE**) passes a nonexistent event type.
- A function that passes a save set (**MI_SAVE_SET**) passes an invalid save set.
- A function that passes a buffer passes a null buffer or a buffer that is too small.
- A function that passes a pointer passes an invalid pointer.
- If the **pointer_checks_enabled** field of the parameter information structure is set, a pointer might not be within the process heap space.

Potential exceptions other than these types of common invalid arguments are mentioned in the **Return Values** section of the individual function descriptions in the *IBM Informix DataBlade API Function Reference*.

An error descriptor for an MI_Exception event indicates the status condition of the event with one of the following *exception levels*.

Status Condition	Exception Level	Description
Warning	MI_MESSAGE	Raised when the database server generates a warning or an informational message. The database server passes a warning back to the client application; it is up to the client to display the warning message.
Runtime error (failure)	MI_EXCEPTION	Raised when the database server generates a runtime error.

The **mi_error_level( )** function returns the exception level from an error descriptor for the MI_Exception event.

Status Variables
To identify the particular cause of an exception, the database server sets the following status variables:

- The **SQLSTATE** status variable holds a five-character code that is compliant with ANSI and X/Open standards.
- The **SQLCODE** status variable holds an integer value that is Informix specific.

SQLSTATE Status Value: **SQLSTATE** is a five-character string that the database server sets after it executes each DataBlade API function. The value of **SQLSTATE** indicates the status of the function execution.

--- American National Standards Institute ---

The **SQLSTATE** status variable is compliant with ANSI and X/Open standards.

--- End of American National Standards Institute ---

This five-character code consists of a two-character *class code* and a three-character *subclass* code. In Figure 10-3, `"IX"` is the class code and `"000"` is the subclass code. The **SQLSTATE** value `"IX000"` indicates that an Informix-specific error has occurred.
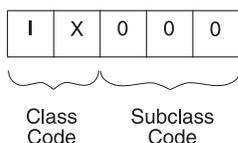
Figure 10-3. The Structure of the SQLSTATE Code with the Value "IX000"

SQLSTATE can contain *only* digits and capital letters. The class code is unique but the subclass code is not. The meaning of the subclass code depends on the associated class code. The initial character of the class code indicates the source of the exception, as Table 10-4 summarizes.

Table 10-4. Initial SQLSTATE Class-Code Values

Initial Class-Code Value	Source of Exception Code	Notes
0 to 4 A to H	X/Open and ANSI/ISO	The associated subclass codes also begin in the range 0 to 4 or A to H.
5 to 9	Defined by the implementation	Subclass codes are also defined by the implementation.
I to Z	Dynamic Server, a DataBlade module, a C UDR, a client LIBMI application	Any of the Informix-specific error messages (those that the X/Open or ANSI/ISO reserved range does *not* support) have an initial class-code value of ″I″ (**SQLSTATE** value of ″IX000″).
		If a UDR returns an error message that this routine has defined, the initial class-code value is ″U″ (**SQLSTATE** value of ″U0001″).
		Other **SQLSTATE** class-code values can be defined by the implementation.

After the database server executes a DataBlade API function, it sets **SQLSTATE** to indicate one of the status conditions, as Table 10-5 shows.

Table 10-5. Status Conditions in SQLSTATE

Status Condition	SQLSTATE Value	
	Class Code	Subclass Code
Success	″00″	″000″
Success, but no rows found	″02″	″000″
Success, but warnings generated	″01″	For ANSI and X/Open warnings: "000" to "006" For Informix-specific warnings: "I01" to "I11" For literal warnings that DataBlade API modules raise: "U01" For custom warnings that DataBlade API modules define: other subclass values, as defined in the **syserrors** system catalog table
Failure, runtime error generated	For ANSI and X/Open errors: > ″02″ For Informix-specific errors: ″IX″ For literal errors that DataBlade API modules raise: ″U0″ For custom errors that DataBlade API modules define: other class codes, as defined in the **syserrors** system catalog table	Error-specific value

For a list of reserved ANSI and X/Open values for **SQLSTATE**, see the description of the GET DIAGNOSTICS statement in the *IBM Informix Guide to SQL: Syntax*. For more information on DataBlade API literal exceptions (″U0001″ and ″01U01″), see "Passing Literal Messages" on page 10-42. For more information on DataBlade API custom exceptions, see "Raising Custom Messages" on page 10-43.

Identifying Warnings with SQLSTATE: When the database server executes a DataBlade API function successfully but encounters a warning condition, it takes the following actions:

- Sets the class code of **SQLSTATE** to "01"
- Sets the subclass code of **SQLSTATE** to a unique value that indicates the cause of the warning (see Table 10-5 on page 10-23)
- Throws an MI_Exception event with an MI_MESSAGE exception level

Identifying Runtime Errors with SQLSTATE: When an SQL statement results in a runtime error, the database server takes the following actions:

- Sets the class code of **SQLSTATE** to a value greater than "02" (see Table 10-5 on page 10-23)
- Sets the subclass code of **SQLSTATE** to a unique value that indicates the cause of the error
- Raises an MI_Exception event with an MI_EXCEPTION exception level

The actual class and subclass codes of **SQLSTATE** identify the particular error. For Informix-specific errors (**SQLSTATE** is "IX000"), you can also check the value of the **SQLCODE** variable to identify the error.

Tip: The database server sets **SQLSTATE** to "02000" (class = "02") when a SELECT or FETCH statement encounters NOT FOUND (or END OF DATA). However, the NOT FOUND condition does not cause a database server exception. Therefore, you do not use **SQLSTATE** to detect this condition from within a callback of a DataBlade API module. Instead, your DataBlade API module can check for the MI_NO_MORE_RESULTS return code from the **mi_get_result( )** function. For more information, see "Retrieving Query Data" on page 8-39.

SQLCODE Status Value: Each **SQLSTATE** value also has an associated Informix-specific status code. The database server saves this Informix-specific status code in the **SQLCODE** status variable. The **SQLCODE** variable is an integer that indicates whether the SQL statement succeeded or failed.

When the database server executes an SQL statement, the database server automatically updates the **SQLCODE** variable. After an SQL statement executes, the **SQLCODE** variable can indicate one of the status conditions that Table 10-6 shows.

Table 10-6. Status Conditions In SQLCODE

Status Condition	SQLCODE Value
Success	0
Success, but no rows found	100
Success, but warnings generated	*not available directly from* SQLCODE
Failure, runtime error generated	< 0

Identifying Warnings with SQLCODE: When the database server executes an SQL statement successfully but encounters a warning condition, it takes the following actions:
- Sets the **SQLSTATE** variable to a five-character warning value
- Sets the **SQLCODE** variable to zero (success)
- Raises the MI_Exception event with the MI_MESSAGE exception level

To identify warnings, examine the value of **SQLSTATE**. The **SQLCODE** value does *not* indicate the cause of a warning. For more information, see "Identifying Warnings with SQLSTATE" on page 10-23.

Identifying Runtime Errors with SQLCODE: When an SQL statement results in a runtime error, the database server takes the following actions:
- Sets **SQLCODE** to a negative value
- Raises an MI_Exception event with an MI_EXCEPTION exception level

The actual number in **SQLCODE** identifies the particular Informix runtime error. The **finderr** or **Error Messages** utility lists error messages and describes corrective actions.

Tip: The database server sets **SQLCODE** to 100 when a SELECT or FETCH statement encounters NOT FOUND (or END OF DATA). However, the NOT FOUND condition does not cause a database server exception. Therefore, you

do not use **SQLCODE** to detect this condition from within a callback of a DataBlade API module. Instead, your DataBlade API module can check for the MI_NO_MORE_RESULTS return code from the **mi_get_result( )** function. For more information, see "Retrieving Query Data" on page 8-39.

Providing Exception Handling

By default, the DataBlade API aborts the current statement when the statement generates a database runtime error and continues execution when the statement generates a database warning. (For more information, see "Using Default Behavior" on page 10-11.)

To override the default exception handling, you must take the following actions:

1. Write a callback function that handles the MI_Exception event.

   To handle an MI_Exception event, you can write either of the following types of callback functions:

   • Exception callback, which executes only when the MI_Exception event occurs

   • All-events callback, which executes when many events occur and can include handling for the MI_Exception event

   Within a callback, the DataBlade API function **mi_error_level( )** returns the exception level for the database server exception. You can also use **mi_error_sql_state( )**, **mi_error_sqlcode( )**, and **mi_errmsg( )** to get more details about the database server exception from its error descriptor. For more information, see "Accessing an Error Descriptor" on page 10-18.

2. Register the callback that handles the MI_Exception event in the DataBlade API module that needs the exception handling.

   Use the **mi_register_callback( )** function to register callback functions. After you register a callback that handles the MI_Exception event, the DataBlade API invokes this callback instead of performing its default exception handling for the event.

Important: Exception callbacks are subject to some restrictions on what tasks they can perform. For more information, see "Writing a Callback Function" on page 10-16.

A database server exception triggers an exception callback only if the DataBlade API module has registered (and enabled) a callback that handles the MI_Exception event. The way that your DataBlade API module handles a database server exception depends on whether the DataBlade API module is a UDR or a client LIBMI application.

Exceptions in a C UDR (Server)

If a C UDR has *not* registered an exception callback on the current connection, the DataBlade API takes a default action based on the exception level of the MI_Exception event. For more information, see "Default Behavior in a C UDR (Server)" on page 10-11.

For example, in Figure 10-4, the **return** statement never executes when a runtime error occurs in the SQL statement that **mi_exec( )** executes.

```
mi_integer
no_exception_handling(flag)
   mi_integer flag;
{
   MI_CONNECTION *conn;

   conn = mi_open(NULL, NULL, NULL);
   mi_exec(conn, "bad SQL statement", MI_QUERY_NORMAL);

/* Not reached; this function aborts on an exception. */
...
   return 1;
}
```

Figure 10-4. A C UDR with Default Exception Handling

When an exception with an MI_EXCEPTION exception level occurs, the DataBlade API aborts the **mi_exec( )** call and the **no_exception_handling( )** routine. The database server returns control to the calling module.

To provide event handling for database server exceptions within a UDR, perform the following tasks:
- Determine if a callback can handle the runtime error.
- In the UDR, register a callback that handles the MI_Exception event.
- In the callback function, return a value that determines how to continue the exception handling once the callback completes.

Handling Errors from DataBlade API Functions: Function descriptions in the *IBM Informix DataBlade API Function Reference* contain a section titled "Return Values." This section lists the possible return values for the associated DataBlade API function. However, whether the calling code actually receives a return value depends on whether the DataBlade API function throws an MI_Exception event when it encounters a runtime error. The DataBlade API functions can be divided into the following subsets based on their response to a database server exception:
- Functions that throw an MI_Exception event
- Functions that do *not* throw an MI_Exception event but provide either MI_ERROR or a NULL-valued pointer when a database server exception occurs
- Functions that can raise an error when a database server exception occurs

Functions That Throw MI_Exception: Most DataBlade API functions throw an MI_Exception event when they encounter a database server exception. For these functions, you can register an exception callback to gain control after a database server exception occurs. Whether the calling code receives a return value from the DataBlade API function depends on how the registered callback handles the MI_Exception event.

Tip: Even if you expect a DataBlade API function to throw an error, the exception handling might possibly ignore it. Therefore, it is recommended that you always check the return value of each DataBlade API function for possible failure.

Functions That Return MI_ERROR or NULL-Valued Pointer: The DataBlade API functions that do *not* throw an MI_Exception event when they encounter a database server exception include the following functions:

- DataBlade API file-access functions: **mi_file_allocate( )**, **mi_file_close( )**, **mi_file_errno( )**, **mi_file_open( )**, **mi_file_read( )**, **mi_file_seek( )**, **mi_file_sync( )**, **mi_file_tell( )**, **mi_file_to_file( )**, **mi_file_unlink( )**, and **mi_file_write( )**
- Memory-allocation functions: **mi_alloc( )**, **mi_dalloc( )**, **mi_realloc( )**, and **mi_zalloc( )**

When one of the preceding DataBlade API functions encounters an exception, the function does *not* cause any callbacks registered for the MI_Exception event to be invoked. Instead, these functions return one of the following values to the calling code to indicate failure:

- MI_ERROR, if the function returns an integer value
- NULL-valued pointer, if the function returns a pointer

The calling code must check the return value of the DataBlade API function and take the appropriate actions. Uncorrected error conditions might lead to worse failures later in processing. For conditions that cannot be corrected, the calling code can provide an informational message to notify the user about what has occurred. The calling code can use the **mi_db_error_raise( )** function to perform the following tasks:

- Explicitly raise an MI_Exception event
- Send a message to the client application

Registering an Exception Callback: When the database server or a UDR raises a database server exception, the database server invokes any callbacks that handle the MI_Exception event and that the UDR has registered (and enabled) on the current connection. Use the **mi_register_callback( )** function to register such a callback. For general information about **mi_register_callback( )**, see "Registering a Callback" on page 10-4.

The code fragment in Figure 10-5 contains the same **mi_exec( )** call as Figure 10-4 on page 10-26. However, this UDR, **has_exception_handling( )**, registers the **excpt_callback( )** function as an exception callback.

```
#include <mi.h>

mi_integer
has_exception_handling(flag)
   mi_integer flag;
{
   static MI_CALLBACK_STATUS MI_PROC_CALLBACK
      excpt_callback( );

   MI_CONNECTION     *conn = NULL;
   MI_CALLBACK_HANDLE *cback_hndl;

   /* Obtain the connection descriptor */
   conn = mi_open(NULL, NULL, NULL);

   /* Register the 'excpt_callback' function as an
   ** exception callback */
   cback_hndl = mi_register_callback(conn, MI_Exception,
      excpt_callback, NULL, NULL);

   /* Generate a syntax error that excpt_callback( ) will
   ** catch */
   ret = mi_exec(conn, "bad SQL statement",
      MI_QUERY_NORMAL);
   if ( ret == MI_ERROR )
      /* handle exception */
   ...
}
```

Figure 10-5. A C UDR with Exception Handling

When the database server exception occurs in the SQL statement that **mi_exec( )** executes, **mi_exec( )** returns MI_ERROR and the **if** statement handles the exception. For a sample implementation of the **excpt_callback( )** callback function, see Figure 10-7 on page 10-30.

For the **excpt_callback( )** function to be invoked for database exceptions that occur on the current connection, you must specify the connection descriptor of the current connection when you register **excpt_callback( )**. In Figure 10-5, **mi_register_callback( )** passes the **conn** connection descriptor, which the **mi_open( )** call has obtained, when it registers **excpt_callback( )**.

The **mi_open( )** function can be resource intensive. If your UDR is likely to be executed many times in the context of a single SQL statement, you might want to cache the connection descriptor from the initial **mi_open( )** call in an **MI_FPARAM** structure. After you save this descriptor, you can reuse it in subsequent invocations of the UDR.

The C UDR in Figure 10-6, **has_exception_handling2( )**, saves the connection descriptor in its **MI_FPARAM** structure the first time it is called and obtains the saved connection descriptor on subsequent calls.

```
mi_integer
has_exception_handling2(flag, fparam)
   mi_integer flag;
   MI_FPARAM *fparam;
{
   MI_CONNECTION *conn = NULL;
   MI_CALLBACK_HANDLE *cback_hndl;
   DB_ERROR_BUF error;

   /* Obtain the connection descriptor from MI_FPARAM */
   conn = (MI_CONNECTION *)mi_fp_funcstate(fparam);
   if ( conn == NULL ) /* first time routine is called */
      {
      /* Obtain the connection descriptor */
      conn = mi_open(NULL, NULL, NULL);

      /* Register the 'excpt_callback2( ) function as an
      ** exception callback on this connection */
      cback_hndl = mi_register_callback(conn,
         MI_Exception,
         excpt_callback2, (void *)&error, NULL);

      /* Save connection descriptor in MI_FPARAM */
      mi_fp_setfuncstate(fparam, (void *)conn);
      }
   ...
}
```

Figure 10-6. Caching the Connection Descriptor in an Exception Callback

For more information on the **MI_FPARAM** structure and the user state, see
"Saving a User State" on page 9-8.

In the preceding code fragment, the **has_exception_handling2( )** routine registers
the **excpt_callback2( )** function as its exception callback. This callback uses a
user-provided buffer to store event information. As its fourth argument, the
mi_register_callback( ) call passes a user-defined buffer named **error** to the
exception callback. For more information, see "Returning Error Information to the
Caller" on page 10-32.

Determining How to Handle the Exception: The return value of an exception
callback tells the database server how to continue handling a database server
exception once the callback completes. An exception callback for a UDR must
return one of the MI_CALLBACK_STATUS values that Table 10-2 on page 10-13
lists.

Handling an Exception in the Callback: To indicate that the callback function
executes *instead of* the default exception handling, an exception callback function
returns the MI_CB_EXC_HANDLED status. This return status tells the DataBlade
API that the actions of the callback have completely handled the exception.

An exception callback function returns the MI_CB_EXC_HANDLED status to
indicate that the callback has completely handled the exception. That is, the actions
of the callback have provided the exception handling. When the DataBlade API
receives the MI_CB_EXC_HANDLED return status, it does *not* perform its default
exception handling. It assumes that the callback has executed *instead of* the default
exception handling. (For more information, see "Default Behavior in a C UDR
(Server)" on page 10-11.)

When a callback returns MI_CB_EXC_HANDLED, the DataBlade API does not propagate the exception up the calling sequence. Therefore, a client application that has executed an SQL expression that contains a UDR does not receive an error from the execution of the UDR (unless the callback uses a user-provided error buffer). If the SQL expression contains no other exceptions, the client application would have an SQLSTATE value of 00000 (success).

Figure 10-7 shows the **excpt_callback( )** exception callback, which is written to handle the MI_Exception event. It returns MI_CB_EXC_HANDLED to indicate that no further exception handling is required.

```
static MI_CALLBACK_STATUS MI_PROC_CALLBACK
excpt_callback(event_type, conn, event_data, user_data)
   MI_EVENT_TYPE event_type;
   MI_CONNECTION *conn;
   void *event_data;
   void *user_data;
{
   /* claim to have handled the exception */
   return MI_CB_EXC_HANDLED;
}
```

Figure 10-7. A Simple Exception Callback

The **excpt_callback( )** function in Figure 10-7 returns MI_CB_EXC_HANDLED, which prevents the DataBlade API from taking any further exception-handling steps, such as invoking other callbacks that handle MI_Exception or aborting the current statement. This callback executes *instead of* the default exception handling.

For the **has_exception_handling( )** routine (which Figure 10-5 on page 10-28 defines), the DataBlade API takes the following steps when the **mi_exec( )** function executes:

1. Executes the **excpt_callback( )** callback when **mi_exec( )** throws an exception
2. Returns control to the first statement in **has_exception_handling( )** after **mi_exec( )**. As a result, execution of the **has_exception_handling( )** routine returns from the **mi_exec( )** call with a return value of MI_ERROR.

Important: Because **excpt_callback( )** returns MI_CB_EXC_HANDLED, the database server assumes that the MI_Exception event does not require any further handling. However, **excpt_callback( )** does not actually take any exception-handling steps; it contains only a **return** statement to return an MI_CB_EXC_HANDLED status. In an actual DataBlade API module, you would probably want to add code to **excpt_callback( )** that provides exception handling.

Continuing with Exception Handling: To indicate that the callback function executes *in addition to* the default exception handling, an exception callback function returns the MI_CB_CONTINUE return status. This return status tells the DataBlade API that the actions of the callback have *not* completely handled the exception and that the DataBlade API should continue with its default exception handling. (For more information, see "Default Behavior in a C UDR (Server)" on page 10-11.) The actions of the callback provide supplemental exception handling.

If the **excpt_callback( )** function in Figure 10-7 had returned MI_CB_CONTINUE instead of MI_CB_EXC_HANDLED, the database server would handle the exception in the **has_exception_handling( )** routine as follows:

1. Execute the **excpt_callback( )** function when the **mi_exec( )** call throws an exception.
2. Abort the **mi_exec( )** call in **has_exception_handling( )**.
3. Return control back to the calling module that called **has_exception_handling( )**.

If **has_exception_handling( )** was a UDR in an SQL statement, the database server would abort the SQL statement and return control to the client application. The client application would be expected to handle the runtime error for the end user.

However, if **has_exception_handling( )** was called by another C UDR that had registered an exception callback, the database server would have executed this callback and continued with the exception handling as the return status of this callback indicated (MI_CB_CONTINUE or MI_CB_EXC_HANDLED). If this callback also returned MI_CB_CONTINUE, the database server would continue up the calling sequence, looking for a registered callback that handled the MI_Exception event. If the database server reached the top-most level in the calling sequence (the UDR within an SQL statement) without locating an exception callback that returned MI_CB_EXC_HANDLED, the database server would abort the UDR and return control to the client application.

For more information on how to write a callback function for a UDR, see "Callback Functions" on page 10-12.

Tip: The MI_Exception event might overlap with the MI_EVENT_END_STMT and MI_EVENT_END_XACT events because an exception always causes either a statement or a transaction to abort. Design the corresponding callbacks with this relationship in mind.

Exceptions in a Client LIBMI Application (Client)

If the client LIBMI application has *not* registered a callback that handles the MI_Exception event on the current connection, the client LIBMI calls the system-default callback. (For more information, see "Default Behavior in Client LIBMI Applications" on page 10-11.)

To provide event handling for database server exceptions within a client LIBMI application, register a callback that handles the MI_Exception event in the client LIBMI application. The DataBlade API invokes any exception callback that the application has registered (and enabled) on the current connection when either of the following actions occurs:

- A client LIBMI application executes a DataBlade API function that throws an MI_Exception event.
- An exception occurs in a UDR that is invoked from a statement in the client LIBMI application and any exception callbacks that the UDR has registered return the MI_CB_CONTINUE return status.

Function descriptions in the *IBM Informix DataBlade API Function Reference* contain a section titled "Return Values." This section lists the possible return values for the DataBlade API function. In a C UDR, DataBlade API function calls might or might not return a value, depending on whether the DataBlade API function throws an MI_Exception event when it encounters a runtime error. However, DataBlade API function calls in a client LIBMI application *always* indicate failure because client-side callbacks always return to the DataBlade API function that generated the error.

On failure, DataBlade API functions return one of the following values to a client LIBMI application:

- MI_ERROR if the return value is an integer
- NULL if the return value is a pointer.

The client LIBMI application can check for these error values and take any appropriate actions.

The client LIBMI application registers callbacks with the **mi_register_callback( )** function. You must provide a valid connection descriptor to **mi_register_callback( )** for all valid event types. For more information, see "Registering a Callback" on page 10-4.

For example, the following **mi_register_callback( )** call registers the **clntexcpt_callback( )** function to handle MI_Exception events:

```
int main (argc, arcv)
   int argc;
   char *argv;
{
   MI_CONNECTION *client_conn;
   MI_CALLBACK_HANDLE *client_cback;
   mi_integer ret;

/* Open a connection to the database server */
client_conn = mi_open(argv[1], NULL, NULL);

/* Register the exception callback */
client_cback = mi_register_callback(client_conn,
   MI_Exception, (MI_VOID *)clntexcpt_callback, NULL, NULL);
if ( client_cback == NULL )
   /* do something appropriate */
...
ret = mi_exec(client_conn, "bad SQL statement",
   MI_QUERY_NORMAL);
if ( ret == MI_ERROR )
   /* perform error recovery */
...
}
```

For more information about how to write a callback function for a client LIBMI application, see "Callback Functions" on page 10-12.

Returning Error Information to the Caller

The fourth argument of a callback function is a pointer to callback *user data*. The user data is a C variable or structure that contains application-specific information that a callback can use. You pass the user data to a callback when you register the callback with the **mi_register_callback( )** function. The fourth argument of **mi_register_callback( )** provides a pointer to the user data (see Figure 10-2 on page 10-16).

One of the most common uses of user data is a *user-defined error structure*. When a callback handles exceptions, DataBlade API functions return either MI_ERROR or NULL on failure. This information is often not specific enough for the calling code to determine the cause of the error. You can create a user-defined error structure to pass more specific error information back to the calling code, as follows:

1. The calling code defines and allocates a user-defined error structure.
2. The callback function populates the user-defined structure with error information.

Defining a User-Defined Error Structure

The calling code can define a user-defined error structure to hold error information. This user-defined structure can be a single C variable or a structure with several pieces of error information. It can contain as much error information as the calling code requires.

Figure 10-8 shows a sample user-defined error structure named DB_ERROR_BUF.

```
#define MSG_SIZE 256
typedef struct error_buf_
{
   mi_integer error_type;
   mi_integer error_level;
   mi_string sqlstate[6];
   mi_string error_msg[MSG_SIZE];
} DB_ERROR_BUF;
```

Figure 10-8. A Sample User-Defined Error Structure

The DB_ERROR_BUF structure holds the following error information.

Error Field	Description
error_type	The event type for the event
	For exception handling, this event type should always be MI_Exception.
error_level	The exception level (or error level) for the event
	For exception handling, this field holds the exception level: MI_MESSAGE or MI_EXCEPTION.
sqlstate	The value of the **SQLSTATE** variable, which indicates the cause of the exception
error_msg	The text of the error message, up to a limit of MSG_SIZE bytes

The calling code must allocate memory for the user-defined error structure. You can use the DataBlade API memory-allocation functions such as **mi_alloc( )** and **mi_dalloc( )**. When you allocate the user-defined error structure, you must associate a memory duration with this structure that you declare that is appropriate to its usage. For example, if the user-defined error structure is to be associated with a registered callback, you must allocate the structure with a memory duration of PER_STMT_EXEC so that this memory is still allocated when the callback executes.

The following **mi_dalloc( )** call allocates a DB_ERROR_BUF buffer with a PER_STMT_EXEC memory duration:

```
mi_dalloc(sizeof(DB_ERROR_BUF), PER_STMT_EXEC);
```

Implementing the Callback

The calling code can use one of the following ways to make a user-defined error structure available to a callback:

- Associate the user-defined structure with the callback.
- Associate the user-defined structure with the database connection.

Associating with a Callback: To associate a user-defined error structure with the registered callback, specify the address of the structure as the fourth argument of

mi_register_callback( ) function. The call to **mi_register_callback( )** initializes the fourth parameter of the exception callback with a pointer to the user-defined structure. For more information, see Figure 10-2 on page 10-16.

─────── **Server Only** ───────

The following **func1( )** UDR registers a callback named **excpt_callback2( )**, which puts error information in the DB_ERROR_BUF user-defined structure (which Figure 10-8 on page 10-33 defines):

```
void func1(flag)
   mi_integer flag;
{
   MI_CONNECTION *conn;
   MI_CALLBACK_HANDLE *cback_hndl;
   DB_ERROR_BUF error;

   /* Initialize information in the error buffer */
   error.sqlstate[0] = '\0';
   strcpy(error.error_msg,
      "func3: initialized error buffer.");

   /* Obtain connection descriptor */
   conn = mi_open(NULL, NULL, NULL);
   if ( conn == NULL )
      mi_db_error_raise(NULL, MI_EXCEPTION,
         "func1: mi_open( ) call failed!");

   /* Register the exception callback */
   cback_hndl = mi_register_callback(conn, MI_Exception,
         excpt_callback2, (void *)&error), NULL):

   /* Execute SQL statement */
      mi_exec(conn, "bad SQL statement", MI_QUERY_NORMAL);

   /* Execution does not reach this point if the
    * excpt_callback2( ) callback returns MI_CB_CONTINUE.
    */
```

The call to **mi_register_callback( )** specifies the address of the user-defined structure as its fourth argument. This structure is, in turn, passed in as the third argument of the **excpt_callback2( )** callback (see Figure 10-2 on page 10-16). The following code implements the **excpt_callback2( )** callback function:

```
MI_CALLBACK_STATUS
excpt_callback2(event_type, conn, event_info, user_data)
   MI_EVENT_TYPE event_type;
   MI_CONNECTION *conn;
   void *event_info;
   void *user_data; /* user-defined error buffer gets
                     * passed here
                     */
{
   DB_ERROR_BUF *user_info;
   mi_integer state_type;
   mi_string *msg;

   user_info = ((DB_ERROR_BUF *)user_data);
   user_info->error_type = event_type;

   if ( event_type != MI_Exception )
      {
      user_info->sqlstate[0] = '\0';
      sprintf(user_info->error_msg,
         "excpt_callback2 called with wrong event type ",
         "%d", event_type);
```

```
      }
   else /* event_type is MI_Exception */
      {
      mi_error_sql_state((MI_ERROR_DESC *)event_info,
         user_info->sqlstate, 6);
      mi_errmsg((MI_ERROR_DESC *)event_info,
         user_info->error_msg, MSG_SIZE-1);
      }

   return MI_CB_EXC_HANDLED;
}
```

Important: Make sure that you allocate the user-defined error structure with a
memory duration that is compatible with the callback that uses it.
Memory durations longer than PER_COMMAND exist for use with
end-of-statement, end-of-transaction, and end-of-session callbacks.
However, these longer memory durations should be used only in
special cases. For more information, see "Choosing the Memory
Duration" on page 14-4.

───────── **End of Server Only** ─────────

───────── **Client Only** ─────────

The following code fragment from a client LIBMI application registers a callback
named **clntexcpt_callback2( )**, which puts error information in the
DB_ERROR_BUF user-defined structure (which Figure 10-8 on page 10-33 defines).

```
int main (argc, argv)
   int argc;
   char **argv;
{
   MI_CONNECTION *conn = NULL;
   char stmt[300];
   MI_CALLBACK_HANDLE callback_hdnl;
   DB_ERROR_BUF error_buff;
   mi_integer ret;

   /* Open a connection to the database server */
   conn = mi_open(argv[1], NULL, NULL);
   if ( conn == NULL )
      /* do something appropriate */

   /* Register the exception callback, with the user-defined
    * error structure as the fourth argument to
    * mi_register_callback( )
    */
   callback_hndl = mi_register_callback(conn, MI_Exception,
      (MI_VOID *)clntexcpt_callback2;
      (MI_VOID *)&error_buff, NULL);
   if ( callback_hndl == NULL )
      /* do something appropriate */
   ...
   /* Execute the SQL statement that 'stmt' contains */
   ret = send_statement(conn, stmt);
   /* If an exception occurred during the execution of the
    * SQL statement, the exception callback initialized the
    * 'error_buff' structure. Obtain error information from
    * 'error_buff'.
    */
   if ( ret == MI_ERROR )
      {
      if ( error_buff.error_type == MI_Exception )
         {
         if ( error_buf.error_level == MI_EXCEPTION )
```

```
             {
             fprintf(stderr, "MI_Exception: level = %d",
                error_buff.error_level);
             fprintf(stderr, "SQLSTATE='%s'\n",
                error_buff.sqlstate);
             fprintf(stderr, "message = '%s'\n",
                error_buff.error_msg);
             /* discontinue processing */
             }
         else /* error_level is MI_WARNING */
             {
             sprintf(warn_msg, "WARNING: %s\n",
                error_buf.error_msg);
             display_msg(warn_msg);
             }
         }
      /* do something appropriate */
   ...
}
```

The call to **mi_register_callback( )** specifies the address of the user-defined structure as its fourth argument. This structure is, in turn, passed in as the fourth argument of the **clntexcpt_callback2( )** callback. The following code implements the **clntexcpt_callback2( )** callback function.

```
void clntexcpt_callback2(event_type, conn, event_info,
     error_info)
   MI_EVENT_TYPE event_type;
   MI_CONNECTION *conn;
   void *event_info;
   void *error_info; /* user-defined error buffer here */
{
   DB_ERROR_BUF *error_buf = (DB_ERROR_BUF *)error_info;

   /* Fill user-defined structure with error information */
   error_buf->error_type = event_type;
   if ( event_type == MI_Exception )
      {
      error_buf->error_level = mi_error_level(event_info);
      mi_error_sql_state(event_info, error_buf->sqlstate, 6);
      mi_errmsg(event_info, error_buf->error_msg, MSG_SIZE-1);
      }
   else
      fprintf(stderr,
         "Warning! clntexcpt_callback( ) fired for event ",
         "%d", event_type);
   return;
}
```

The **clntexcpt_callback( )** function is an example of an exception callback for a client LIBMI application. This callback returns **void** because the client LIBMI does not interpret the **MI_CALLBACK_STATUS** return value, as does the database server for UDRs.

──────────────── End of Client Only ────────────────

Associating with the Connection: To associate a user-defined error structure with the connection, you:

- Use the **mi_set_connection_user_data( )** function in the calling function to bind the structure to a connection descriptor (an **MI_CONNECTION** structure).
- Use the **mi_get_connection_user_data( )** function in the callback to obtain the structure that is bound to the connection descriptor.

Important: You can associate a user-defined error structure with a connection only if a valid connection exists and this connection does not change between the point at which the callback is registered and the point at which the exception event occurs. If you cannot guarantee that these two conditions exist, associate the user-defined error structure with the registered callback (page 10-33).

Client Only

The following code fragment from a client LIBMI application binds the DB_ERROR_BUF user-defined structure (Figure 10-8 on page 10-33) to a connection:

```
int main (argc, argv)
   int argc;
   char **argv;
{
   MI_CONNECTION *conn = NULL;
   MI_CALLBACK_HANDLE *cback_hndl;
   char query[300];
   mi_integer ret;
   DB_ERROR_BUF error_buff;

   conn = mi_open(argv[1], NULL, NULL);
   if ( conn == NULL )
      /* do something appropriate */
      ...
   cback_hndl = mi_register_callback(conn, MI_Exception,
      (MI_VOID)clntexcpt_callback2, NULL, NULL);
   ret = mi_set_connection_user_data(conn,
      (MI_VOID)&error_buff);
   if ( ret == MI_ERROR )
      /* do something appropriate */
      ...
   ret = send_command(conn, query);
   if ( ret == MI_ERROR)
      {
      fprintf(stderr, "MI_Exception: level = %d",
         error_buff.error_level);
      fprintf(stderr, "SQLSTATE='%s'\n",
         error_buff.sqlstate);
      fprintf(stderr, "message = '%s'\n",
         error_buff.error_msg);
      }
   /* do something appropriate */
   ...
}
```

The call to **mi_register_callback( )** does not specify the address of the user-defined structure as its fourth argument because this structure is associated with the connection. The following code implements the **clntexcpt_callback( )** callback function, which uses the **mi_get_connection_user_data( )** function to obtain the user-defined structure:

```
void clntexcpt_callback2(event_type, conn, event_info,
      user_data)
   MI_EVENT_TYPE event_type;
   MI_CONNECTION *conn; /* user-defined error buffer here */
   void *event_info;
   void *user_data;
{
   DB_ERROR_BUF *error_buf

   mi_get_connection_user_data(conn, (void **)&error_buf)
   error_buf->error_type = event_type;
```

```
      if ( event_type == MI_Exception )
        {
        error_buf->error_level = mi_error_level(event_info);
        mi_error_sql_state(event_info, error_buf->sqlstate, 6);
        mi_errmsg(event_info, error_buf->error_msg, MSG_SIZE-1);
        }
      else
        fprintf(stderr,
          "Warning! clntexcpt_callback2( ) fired for event "
          "%d", event_type);
      return;
}
```

In the preceding code fragment, the italicized portion is the only difference
between this client LIBMI application and the one that registers a user-defined
variable with the callback (in "Associating with a Callback" on page 10-33).

─── **End of Client Only** ───

Handling Multiple Exceptions

The database server can generate multiple exceptions for a single SQL statement. A
single statement might generate multiple exceptions when any of the following
conditions have occurred:

- Multiple warnings occur.
- Multiple details are associated with a single error occurrence.

  For example, a DROP TABLE statement might set both the **SQLCODE** value
  and the ISAM error value. Similarly, nested UDRs might generate errors at many
  levels.

The database server normally calls a registered exception callback once for each
exception message. If a single error causes multiple exceptions, you must use the
following DataBlade API functions in the callback to process multiple messages in
a single invocation.

DataBlade API Function	Description
mi_error_desc_next( )	Obtains the *next* error descriptor from the current exception list
	The list of exceptions that the current statement generates is called the *current exception list*.
mi_error_desc_finish( )	Completes the processing of the current exception list
	A callback can use this function to prevent its being called again for any more exceptions currently associated with the current statement.

A callback is not called again for any messages that have already been processed.
The database server presents exceptions from highest message level to lowest
message level. Therefore, a UDR or SQL message occurs first, followed by any
ISAM message.

The smart-large-object functions (**mi_lo_***) raise an MI_Exception event if they
encounter a database server exception. However, the smart-large-object error is the
second message that the database server returns. Therefore, an exception callback
needs to include the following steps to obtain an exception from an **mi_lo_***
function:

1. Call **mi_error_sqlcode( )** to get the high-level **SQLCODE** value.
2. Call **mi_error_desc_next( )** to get the next error descriptor.
3. Call **mi_error_sqlcode( )** again to get the detailed smart-large-object error (and ISAM error code).

The following callback function, **excpt_callback3( )**, is a modified version of the **excp_callback2( )** callback that handles multiple exceptions:

```
MI_CALLBACK_STATUS excpt_callback3(event_type, conn,
      event_info, user_data)
   MI_EVENT_TYPE event_type;
   MI_CONNECTION *conn;
   void *event_info;
   void *user_data; /* user-defined error buffer gets
                     * passed here
                     */
{
   DB_ERROR_BUF *user_info;
   mi_integer state_type;
   mi_string *msg;

   mi_integer i=0;
   /* Pointer to multiple error messages */
   MI_ERROR_DESC *err_desc=NULL;

   user_info = ((DB_ERROR_BUF *)user_data);

   user_info->error_type = event_type;
   if ( event_type != MI_Exception )
      {
      user_info->sqlstate[0] = '\0';
      sprintf(user_info->error_msg,
         "excpt_callback3 called with wrong event type ",
          "%d", event_type);

      /* Send trace message for default trace class */
      DPRINTF("__myErrors__", 1, ("<<<<>>>> mesg=%s",
         user_info->error_msg));
      return MI_CB_CONTINUE;
      }

   err_desc = (MI_ERROR_DESC *)event_info;
   i++;
   mi_error_sql_state(err_desc, user_info->sqlcode, 6);
   mi_errmsg(err_desc, user_info->error_msg, MSG_SIZE-1);

   DPRINTF("__myErrors__", 1,
      ("<<<<>>>> mesg %d: sqlcode=%s, mesg=%s", i,
      user_info->sqlcode, user_info->error_msg));

/* Overwrites previous error. Another approach would be to
 * allocate enough 'user_info' space to store all errors
 */
   if ( (err_desc=
         mi_error_desc_next((MI_ERROR_DESC *)event_info))
         != NULL )
      {
      i++;
      mi_error_sql_state(err_desc, user_info->sqlcode, 6);
      mi_errmsg(err_desc, user_info->error_msg,
         MSG_SIZE-1);

      DPRINTF("__myErrors__", 1,
         ("<<<<>>>> mesg %d: sqlcode=%s, mesg=%s", i,
```

```
                    user_info->sqlcode, user_info->error_msg));
        }
    return MI_CB_CONTINUE;
}
```

This callback also uses the DPRINTF macro to send trace messages to an output file. For more information on tracing, see "Using Tracing" on page 12-28.

Raising an Exception

If a DataBlade API module detects an error, it can use the **mi_db_error_raise( )** function to raise an exception.

---- Server Only ----

In a C UDR, the **mi_db_error_raise( )** function raises an exception to the database server.

---- End of Server Only ----

---- Client Only ----

In a client LIBMI application, the **mi_db_error_raise( )** function sends the exception over to the database server.

---- End of Client Only ----

When the **mi_db_error_raise( )** function raises an exception, the database server handles this exception in the same way it would if a database server exception occurred in a DataBlade API function. If the DataBlade API module has registered an exception callback, this call to **mi_db_error_raise( )** invokes the exception callback. If no exception callback has been registered, the DataBlade API uses the default behavior for the handling of exceptions.

Specifying the Connection

The first argument to the **mi_db_error_raise( )** function is a connection descriptor. This connection descriptor can be either a NULL-valued pointer or a pointer to a valid connection. Which values are valid depend on whether the calling module is a UDR or a client LIBMI application.

---- Server Only ----

In a C UDR, you can specify the connection descriptor to **mi_db_error_raise( )** as either of the following values:

- A NULL-valued pointer, which raises an exception on the parent connection
- A pointer to the current connection descriptor, which raises an exception on the current connection

Raising an Exception on the Parent Connection:

When you specify a NULL-valued connection descriptor to the **mi_db_error_raise( )** function, this function raises the exception against the *parent connection*, which is the connection on which the C UDR was invoked. This connection might be a client connection or a UDR-owned connection that was passed to **mi_exec( )**, **mi_exec_prepared_statement( )**, or **mi_routine_exec( )**.

If the raised exception has an MI_EXCEPTION exception level, the database server aborts both the UDR and the current SQL expression. For both exception levels (MI_EXCEPTION and MI_MESSAGE), the database server passes the event message to the module on the parent connection and returns control to this module. If the UDR needs control of the exception, it must call **mi_db_error_raise( )** with a pointer to the current connection descriptor.

The following example shows how the MI_EXCEPTION exception level causes the **my_function( )** UDR to abort when **mi_db_error_raise( )** specifies a NULL-valued pointer as its connection descriptor:

```
void MI_PROC_VACALLBACK my_function( )
{
   ... Processing ...
   if ( error condition )
      {
      ... do any clean-up here ...
      ret = mi_db_error_raise ((MI_CONNECTION *)NULL,
         MI_EXCEPTION, "FATAL ERROR in my_function( )!");
      }
   ... These lines never get reached ...
}
```

Execution returns to the code that called **my_function( )**. If this code has an exception callback, this callback determines how the exception handling continues.

Raising an Exception on the Current Connection:

When you specify a valid connection descriptor to the **mi_db_error_raise( )** function, this function raises the exception against the specified connection. The DataBlade API invokes any callbacks that are registered for the MI_Exception event on this same connection. If a registered callback returns the MI_CB_EXC_HANDLED status, control returns to the UDR. (For more information, see "Determining How to Handle the Exception" on page 10-29.

When the **my_function( )** routine registers a callback, the callback can catch the exception with an MI_EXCEPTION exception level, as the following example shows:

```
void MI_PROC_VACALLBACK
my_function( )
{
   conn = mi_open(NULL, NULL, NULL);
   ...
   cback_hndl = mi_register_callback(conn, MI_Exception,
      excpt_callback, NULL, NULL);
   ... Processing ...
   if ( error condition )
      {
      ... do any clean-up here ...
      ret = mi_db_error_raise (conn, MI_EXCEPTION,
         "The excpt_callback( ) function is invoked from \
         my_function( ).");
      }
   ... These lines do get reached if excpt_callback( )
      returns MI_CB_EXC_HANDLED...
}
```

For a sample implementation of the **excpt_callback( )** function, see Figure 10-7 on page 10-30.

In a client LIBMI application, you must specify a valid connection descriptor to the **mi_db_error_raise( )** function. For an exception callback to be invoked when the **mi_db_error_raise( )** function raises an exception, specify the same connection descriptor as the one on which the callback was registered.

For example, in the following code fragment, the call to **mi_db_error_raise( )** causes the **excpt_callback( )** function to be invoked when an MI_Exception event occurs:

```
conn1 = mi_open(argv[1], NULL, NULL);
cback_hndl = mi_register_callback(conn1, MI_Exception,
   clnt_callback, (void *)&error, NULL);
...
mi_db_error_raise(conn1, MI_EXCEPTION,
   "The clnt_callback( ) callback is invoked.");
```

Both **mi_register_callback( )**, which registers the callback for the MI_Exception event, and **mi_db_error_raise( )**, which raises the MI_Exception event, specify **conn1** as their connection descriptor.

Specifying the Message

The message that **mi_db_error_raise( )** passes to an exception callback can be either of the following types:

- A *literal message*, which you provide as the third argument to **mi_db_error_raise( )**
- A *custom message* that is associated with a **SQLSTATE** value, which you provide as the third argument to **mi_db_error_raise( )**

Passing Literal Messages: To raise an exception whose message text you provide, the **mi_db_error_raise( )** function requires the following information:

- A message type of MI_MESSAGE or MI_EXCEPTION
- The associated message text

When you pass the MI_MESSAGE or MI_EXCEPTION message type to the **mi_db_error_raise( )** function, the function raises an MI_Exception event whose error descriptor contains the following information.

Error Descriptor Field	Warning	Runtime Error
Exception level (2nd argument)	MI_MESSAGE	MI_EXCEPTION
SQLSTATE value	″01U01″	″U0001″
Message text (3rd argument)	Specified warning text	Specified error text

For example, the following call to **mi_db_error_raise( )** raises an MI_Exception event with an exception level of MI_MESSAGE, an **SQLSTATE** value of ″01U01″, and the ″Operation Successful″ warning message:

```
mi_db_error_raise(conn, MI_MESSAGE, "Operation Successful");
```

For the following line, **mi_db_error_raise( )** raises an MI_Exception event with an exception level of MI_EXCEPTION, an **SQLSTATE** value of ″U0001″**,** and the ″Out of Memory!!!″ error message:

```
mi_db_error_raise(conn, MI_EXCEPTION, "Out of Memory!!!");
```

If any exception callback is registered for the same connection, the DataBlade API sends this error descriptor to the callback when the MI_Exception event is raised.

--------------------------- Server Only ---------------------------

If the C UDR (or any if its calling routines) has *not* registered an exception callback when the MI_Exception event is raised, the DataBlade API performs the default exception handling, which depends on the exception level of the exception:

- If the exception has an MI_EXCEPTION exception level, the database server aborts the UDR and returns control to the calling module.
- If the exception has an MI_MESSAGE exception level, the database server sends the warning message to the calling module and continues execution of the UDR.

------------------------ End of Server Only ------------------------

--------------------------- Client Only ---------------------------

If the client LIBMI application has not registered an exception callback when the MI_Exception event is raised, the client LIBMI calls the system-default callback, which provides the following information:

- The connection
- The exception type: MI_MESSAGE or MI_EXCEPTION
- The message text that is associated with the exception

For more information on the actions of the system-default callback, see "Using Default Behavior" on page 10-11.

------------------------- End of Client Only -------------------------

Raising Custom Messages: The **mi_db_error_raise( )** function can raise exceptions with custom messages, which DataBlade modules and UDRs can store in the **syserrors** system catalog table. The **syserrors** table maps these messages to five-character **SQLSTATE** values.

To raise an exception whose message text is stored in **syserrors**, you provide the following information to the **mi_db_error_raise( )** function:

- A message type of MI_SQL
- The value of the **SQLSTATE** variable that identifies the custom exception
- Optionally, values specified in parameter pairs that replace markers in the custom exception message

When you pass the MI_SQL message type to the **mi_db_error_raise( )** function, the function raises an MI_Exception event whose error descriptor contains the following information:

Error Descriptor Field	Warning	Runtime Error
Exception level	MI_MESSAGE	MI_EXCEPTION
	(If the SQLSTATE value has a class code of "01")	(If the SQLSTATE value has a class code of "02" or greater)
SQLSTATE value (3rd argument)	Specified warning value:	Specified error value:
	"01xxx"	"xxxxx"
		(class code of "02" or greater)
Message text	Associated warning text from **syserrors** table	Associated error text from **syserrors** table

If any exception callback is registered for the same connection, the DataBlade API sends this error descriptor to the callback when the MI_Exception event is raised. For example, assume that the following predefined error message is under an **SQLSTATE** value of "03I01" in the **syserrors** table:

```
Operation Interrupted.
```

The following call to **mi_db_error_raise( )** sends this predefined error message to a registered (and enabled) callback that handles the MI_Exception event:

```
mi_db_error_raise (conn, MI_SQL, "03I01", NULL);
```

The exception level for this exception would be MI_EXCEPTION because any **SQLSTATE** value whose class code is greater than "02" is considered to represent a runtime error. If no such callback was registered (or enabled), the database server would take its default exception-handling behavior.

If the **SQLSTATE** value had a class code of "01", **mi_db_error_raise( )** would raise a warning instead of an error. (For more information on **SQLSTATE** values, see "SQLSTATE Status Value" on page 10-22.) The following **mi_db_error_raise( )** call raises an MI_Exception event whose exception level is MI_MESSAGE:

```
mi_db_error_raise(conn, MI_SQL, "01877", NULL);
```

When this exception is raised, execution continues at the next line after this call to **mi_db_error_raise( )**.

Tip: Both of the preceding **mi_db_error_raise( )** examples specify NULL as the last argument because neither of their **syserrors** messages contains parameter markers. For more information on parameter markers, see "Specifying Parameter Markers" on page 10-46.

Searching for Custom Messages: When the **mi_db_error_raise( )** function initiates a search of the **syserrors** table, it requests the message in which all components of the locale (language, territory, code set, and optional modifier) are the same in the current processing locale and the **locale** column of **syserrors**.

Tip: For more information on the columns of the **syserrors** system catalog table, see the chapter on the system catalog tables in the *IBM Informix Guide to SQL: Reference*. For more information on **SQLSTATE**, see "SQLSTATE Status Value" on page 10-22.

For DataBlade API modules that use the default locale, the current processing locale is U.S. English (**en_us**). (The name of the default code set depends upon the platform you use. For more information on default code sets, see the *IBM Informix*

GLS User's Guide.) When the current processing locale is U.S. English, **mi_db_error_raise( )** looks only for messages that use the U.S. English locale.

Global Language Support

For DataBlade API modules that use nondefault locales, the current processing locale is one of the following locales:

Server Only

- For C UDRs, the current processing locale is the server-processing locale.

——————— **End of Server Only** ———————

Client Only

- For client LIBMI applications, the current processing locale is the client locale.

——————— **End of Client Only** ———————

For more information on the client, database server, and server-processing locales, see the *IBM Informix GLS User's Guide*.

——————— **End of Global Language Support** ———————

A GLS locale name has the format *ll_tt.codeset@modf*, in which *ll* is the name of the language, *tt* is the name of the territory, *codeset* is the name of the code set, and *modf* is the 4-character name of the optional locale modifier. (For more information on locale names, see the *IBM Informix GLS User's Guide*.) The *ll_tt.codeset@modf* format is the standard GLS locale name. Locale names can take other forms. However, **mi_db_error_raise( )** first attempts to convert the names of the current processing locale and the **syserrors** locale into this standard GLS format.

The **mi_db_error_raise( )** function then performs a string comparison on these two locale names. The function attempts to match a value in the **locale** column of **syserrors** with the GLS name of the current processing locale as follows:

1. Convert the current processing locale and **syserrors** locale names into standard GLS names, if possible.

   If **mi_db_error_raise( )** cannot map the current processing locale name to a standard name, it cannot perform the match.

2. Match the current processing locale name with an entire locale name, if possible.

   Locate a row in **syserrors** whose **locale** column has a value that matches the full *ll_tt.codeset@modf* locale name.

3. Match the current processing locale name with only the language and territory part of a locale name, if possible.

   Locate a row in **syserrors** whose **locale** column starts with the value *ll_tt* (only language and territory names match).

4. Match the current processing locale name with only the language part of a locale name, if possible.

   Locate a row in **syserrors** whose **locale** column starts with the value *ll* (only language name matches).

5. Match the current processing locale name with the default locale (U.S. English), if it is available.

Locate a row in **syserrors** whose **locale** column matches the standard GLS name of the default locale.

When **mi_db_error_raise( )** finds a matching locale name for the specified **SQLSTATE** value, it then verifies that the code set of the locale name from **syserrors** is compatible with the code set of the current processing locale. A *compatible* code set is one that is either the same as or can be converted to the current processing code set. If the two code sets are *not* compatible, **mi_db_error_raise( )** continues to search the **syserrors** table for rows that match the specified **SQLSTATE** value. Once **mi_db_error_raise( )** finds a matching row, it obtains the text from the corresponding **message** column of **syserrors**.

For example, suppose the current processing locale is the French Canadian locale, **fr_ca.8859-1**, and you issue the following call to **mi_db_error_raise( )**:

```
mi_db_error_raise(conn, MI_SQL, "08001", NULL);
```

The **mi_db_error_raise( )** function performs the following search process to locate entries in **syserrors**:

1. Is there a row with the **sqlstate** column of ″08001″ and a **locale** value that matches ″fr_ca.8859-1″?
2. Is there a row with the **sqlstate** column of ″08001″ and a **locale** value that starts with ″fr_ca″?
3. Is there a row with the **sqlstate** column of ″08001″ and a **locale** value that starts with ″fr″?
4. Is there a row with the **sqlstate** column of ″08001″ and a **locale** value that starts with ″en_us″?

Suppose **mi_db_error_raise( )** finds a row in **syserrors** with an **sqlstate** value of ″08001″ and a **locale** of ″fr_ca.1250″. The function obtains the associated text from the **message** column of this row if it can find valid code-set conversion files between the ISO8859-1 code set (8859-1) and the Microsoft® 1250 code set (1250).

--- **Server Only** ---

For C UDRs, these code-set conversion files must exist on the server computer.

--- **End of Server Only** ---

--- **Client Only** ---

For client LIBMI applications, these code-set conversion files must exist on the client computer.

--- **End of Client Only** ---

Specifying Parameter Markers: The custom message in the **syserrors** system catalog table can contain parameter markers. These *parameter markers* are sequences of characters enclosed by a single percent sign on each end (for example, %TOKEN%). A parameter marker is treated as a variable for which the **mi_db_error_raise( )** function can supply a value.

For messages with parameter markers, **mi_db_error_raise( )** can handle a variable-length parameter list, as follows:

• Values specified in parameter pairs can replace parameter markers in the **syserrors** error or warning message.

- The function passes in NULL to terminate the list of parameter pairs.

The **mi_db_error_raise( )** function requires a parameter pair for each parameter marker in the message. A parameter pair has the following values:
- The first member of the pair is a null-terminated string that represents the name and format of the parameter marker.
- The second member of the pair is the value to assign the parameter.

Parameter pairs do not have to be in the same order as the markers in the string.

Important: Terminate the parameter list arguments with a NULL pointer. If a NULL pointer does not terminate the list, the results are unpredictable.

The first member of the parameter pair has the following syntax:

`parameter_name%format_character`

The **mi_db_error_raise( )** function supports following *format_character* values.

Format Character	Meaning
d	Integer
f,g,G,e,E	Double (by reference)
T	Text (mi_lvarchar type, that is, a pointer to mi_lvarchar)
t	Length followed by string (two separate parameters)
s	Null-terminated C string

For example, suppose that the following message is under an **SQLSTATE** value of ″2AM10″ in the **syserrors** table:

`"Line %LINE%: Syntax error at '%TOKEN%':%CMD%"`

This message contains the following parameter markers: LINE, TOKEN, and CMD. The following call to **mi_db_error_raise( )** assigns the string ″selecl″ to the TOKEN parameter, 500 to the LINE parameter, and the text of the query to the CMD parameter in the message text:

```
mi_db_error_raise (conn, MI_SQL, "2AM10",
   "TOKEN%s", "selecl",
   "LINE%d", (mi_integer)500,
   "CMD%s", "selecl * from tables\;",
   NULL);
```

The string ″TOKEN%s″ indicates that the value that replaces the parameter marker %TOKEN% in the message string is to be formatted as a string (%s). The next member of the parameter pair, the string "selecl", is the value to format.

This **mi_db_error_raise( )** call sends the following message to an exception callback:

`"Line 500: Syntax error at 'selecl':selecl * from tables;"`

The **mi_db_error_raise( )** function assumes that any message text or message parameter strings that you supply are in the current processing locale.

Adding Custom Messages: You can store custom status codes and their associated messages in the **syserrors** system catalog table.

To add a custom message:
1. Determine the **SQLSTATE** code for the message you want to add.
2. Insert a row into the **syserrors** system catalog table for the new message.

Choosing an SQLSTATE Code: The **syserrors** system catalog table holds custom messages for DataBlade modules and UDRs. A unique **SQLSTATE** value identifies each row in the **syserrors** system catalog table. Therefore, to store a custom message in **syserrors**, you assign it an **SQLSTATE** value.

You must ensure that this **SQLSTATE** value is unique within **syserrors**. When you choose an **SQLSTATE** value, keep the following restrictions in mind:

- The database server has its own set of system messages.

  Messages that the database server provides are not stored in the **syserrors** system catalog table. However, any special modules that are included with the database server (such as R-tree support) might have their own messages in **syserrors**.

- SQL reserves various **SQLSTATE** codes for its own use.

  These messages are not stored in the **syserrors** table. For a list of the reserved values of **SQLSTATE**, see "SQLSTATE Status Value" on page 10-22.

- All **SQLSTATE** values for warnings begin with the ″01″ class code.

  To define a custom warning message, you must define an **SQLSTATE** value that has a ″01″ class code and an unused subclass code.

- An installed DataBlade module might have stored its messages in **syserrors**.

  When a DataBlade module is installed, associated messages might be added to **syserrors**. Avoid the use of any **SQLSTATE** values that an installed DataBlade module might use. You must also take care not to delete installed messages, or they must be re-created by a restore from a backup or a reinstallation.

- If you are developing a DataBlade module, coordinate your **SQLSTATE** values with other DataBlade modules that you are using.

  You can group warnings and errors for your DataBlade modules into the same class code, each with a different subclass code.

You can use the following query to determine the current list of **SQLSTATE** message strings in **syserrors**:

```
SELECT sqlstate, locale, message FROM syserrors
ORDER BY sqlstate, locale;
```

The **locale** column is used for the internationalization of error and warning messages. For more information, see "Searching for Custom Messages" on page

10-44.

Adding New Messages: To create a new message, insert a row into the **syserrors** system catalog table. By default, all users can view this table, but only users with the DBA privilege can modify it. For more information on columns of the **syserrors** system catalog table, see the *IBM Informix Guide to SQL: Reference*.

For example, the following INSERT statement inserts a new message into **syserrors** whose **SQLSTATE** value is ″03I01″:

```
INSERT INTO syserrors
VALUES ("03I01", "en_us.8859-1", 0, 1,
   "Operation Interrupted.");
```

Enter message text in the language of the target locale, with the characters in the locale code set. By convention, do not include any newline characters in the message. Make sure you also update the **locale** column of **syserrors** with the name of the target locale of the message text. For information on locale names, see the *IBM Informix GLS User's Guide*.

Do not allow any code-set conversion to take place when you insert the message text. If the code sets of the client and database locales differ, temporarily set both the **CLIENT_LOCALE** and **DB_LOCALE** environment variables in the client environment to the name of the database locale. This workaround prevents the client application from performing code-set conversion.

If you specify any parameters in the message text, include only ASCII characters in the parameters names. Following this convention means that the parameter name can be the same for all locales. All code sets include the ASCII characters.

State-Transition Events

When the database server raises a state-transition event, the database server invokes any callbacks registered for the state transition. This section provides information about state-transition handling in DataBlade API modules, including an explanation of state-transition events and a description of how to handle a state-transition event in a C UDR and a client LIBMI application.

Understanding State-Transition Events

State-transition events occur when the database server changes its processing state. The DataBlade API represents a state transition as one of the enumerated values of the **MI_TRANSITION_TYPE** data type. The following table shows the transitions in the server-processing state and the corresponding **MI_TRANSITION_TYPE** values.

State-Transition Type	Description
MI_BEGIN	The database server is beginning a new transaction.
MI_NORMAL_END	The database server just completed the current event successfully.

State-Transition Type	Description
MI_ABORT_END	The database server just rolled back the current event. (The statement failed, or the transaction was aborted or rolled back.)

The **milib.h** header file defines the **MI_TRANSITION_TYPE** data type and its state-transition values.

The following table shows the state-transition types and the state-transition events that they can cause.

State-Transition Type	Event in Client LIBMI Application	Event in C UDR
Begin transaction or savepoint (MI_BEGIN)	MI_Xact_State_Change	None
Event end: commit (MI_NORMAL_END)	MI_Xact_State_Change	MI_EVENT_SAVEPOINT MI_EVENT_COMMIT_ABORT MI_EVENT_POST_XACT MI_EVENT_END_STMT MI_EVENT_END_XACT MI_EVENT_END_SESSION
Event end: rolled back (MI_ABORT_END)	MI_Xact_State_Change	MI_EVENT_SAVEPOINT MI_EVENT_COMMIT_ABORT MI_EVENT_POST_XACT MI_EVENT_END_STMT MI_EVENT_END_XACT MI_EVENT_END_SESSION

Beginning a Transaction

─── **Client Only** ───

When the database server begins a transaction block, it raises *only* the MI_Xact_State_Change event. The MI_Xact_State_Change event occurs *only* in the context of a client LIBMI application when the database server enters and leaves a transaction block. Only client callback functions can catch this begin-transaction event.

You handle the MI_Xact_State_Change event only in the context of a client LIBMI application. It occurs within a client LIBMI application when the current transaction ends with either a commit or a rollback. The MI_Xact_State_Change event also occurs when the database server begins a transaction.

─── **End of Client Only** ───

A state-transition callback executes when the following state-transition event occurs.

State-Transition Event Type	Callback Type
MI_Xact_State_Change	State-change callback

A C UDR does not begin transactions. It inherits the transaction of the client application that calls the SQL statement that contains the UDR.

Ending a Session (Server)

The MI_EVENT_END_SESSION event occurs when the database server reaches the end of the current session. A *session* begins when the client application opens a database connection and ends when the client application closes the connection (or when the client application ends). For more information, see "Closing a Connection" on page 7-18.

These events occur *only* within the context of a C UDR. Their main purpose is to clean up resources that the UDR might have allocated. The database server does not throw the MI_EVENT_END_SESSION event when it terminates abnormally.

Providing State-Transition Handling

The database server throws a state-transition event when it changes its processing state.

To handle a transition in the processing state:

1. Write a state-transition callback.

   Within a state-transition callback, use the **mi_transition_type( )** function on the transition descriptor to determine the state-transition type (begin, normal end, or abort end) that caused the event. The processing required is typically different for each transition type.

2. Register the state-transition callback in the DataBlade API module that needs the state-transition handling.

   To provide required DataBlade processing at a state-transition point, your DataBlade API module must register state-transition callbacks with the **mi_register_callback( )** function.

The way that your DataBlade API module handles a state-transition event depends on whether the DataBlade API module is a C UDR or a client LIBMI application.

State Transitions in a C UDR (Server)

In a C UDR, the following state-transition events might occur:

- MI_EVENT_SAVEPOINT
- MI_EVENT_COMMIT_ABORT
- MI_EVENT_POST_XACT
- MI_EVENT_END_STMT
- MI_EVENT_END_XACT
- MI_EVENT_END_SESSION

For these state-transition events, the **mi_transition_type( )** function returns a state-transaction type of normal end (MI_NORMAL_END) or abort end (MI_ABORT_END). In a UDR, state-transition callbacks are always called at the end of a transaction (normal or aborted), never at the beginning.

Managing Memory Allocations: If your code allocates memory for user data that the callback function needs, this memory must have a duration long enough to persist until the execution of the callback. Otherwise, the callback cannot access the user data. This user data might include information that the callback function needs to handle the event or to notify users of the cause of the event.

The following table shows the memory durations associated with callback and event types.

Callback or Event Type	Memory Duration to Use
End-of-statement	PER_STMT_EXEC
End-of-transaction	PER_TRANSACTION
End-of-session	PER_SESSION
MI_EVENT_POST_XACT	PER_TRANSACTION
MI_EVENT_SAVEPOINT	PER_TRANSACTION
MI_EVENT_COMMIT_ABORT	PER_TRANSACTION

At the end of the memory duration associated with the callback, the database server deallocates the memory as part of its final cleanup.

For more information on memory durations, see "Choosing the Memory Duration" on page 14-4.

Managing the Transaction: The transaction system of the database server only guarantees transaction semantics on all objects that are internal to the database. However, transactions might also include operations on external objects. A UDR might perform such operations, such as creating a temporary file or sending a message.

For transactions that consist of operations on both internal and external objects, you can use one of the following types of callbacks to commit or to undo (if possible) the operations on the external objects, based on the transaction status:
- Commit-abort callback (MI_EVENT_COMMIT_ABORT)
- End-of-statement callback (MI_EVENT_END_STMT)

  Each SQL statement behaves like a subtransaction when in a transaction block or like a transaction when not in a transaction block.
- End-of-transaction callback (MI_EVENT_END_XACT)

  Registration of a commit-abort callback is preferable.
- Savepoint callback (MI_EVENT_SAVEPOINT)

  Each cursor flush behaves like a subtransaction when in a transaction block.

To enable these callbacks to roll back a transaction, the DataBlade API allows end-of-statement and end-of-transaction callbacks to raise an exception and register their own exception callbacks.

The database server calls an end-of-statement or end-of-transaction callback *before* it attempts to commit the transaction. When called before a commit, these callbacks receive a transition descriptor that has a transition state of MI_NORMAL_END. However, if either of these callbacks encounters an error during its execution, you probably do *not* want to allow the transaction to commit.

To cause an event to be aborted or rolled back, you can raise an exception from a state-transition callback by calling the **mi_db_error_raise( )** function. When a state-transition callback raises an exception, the DataBlade API takes the following actions:

1. Terminates further processing of the end-of-statement or end-of-transaction callback

2. Terminates the current transaction and changes the transition state from MI_NORMAL_END (commit) to MI_ABORT_END (rollback)

3. Invokes any end-of-statement or end-of-transaction callbacks again, this time with the MI_ABORT_END transition state

4. Invokes any exception callback that the end-of-statement or end-of-transaction callback has registered to handle the exception

An end-of-transaction callback executes within a C UDR when the MI_EVENT_END_XACT event occurs. The following code implements an end-of-transaction callback named **endxact_callback( )**, which inserts a row into a database table, **tran_state**, to indicate the state of the current transaction:

```
MI_CALLBACK_STATUS MI_PROC_CALLBACK
endxact_callback(event_type, conn, event_data, user_data)
   MI_EVENT_TYPE event_type;
   MI_CONNECTION *conn;
   void *event_data;
   void *user_data;
{
   MI_CONNECTION *cb_conn;
   cb_conn = mi_open(NULL, NULL, NULL);
   (void) mi_register_callback(cb_conn, MI_Exception,
      eox_excpt_callback( ), NULL, NULL);
   if ( event_type == MI_EVENT_END_XACT )
      {
      mi_integer change_type;

      change_type = mi_transition(event_data);
      switch ( change_type )
         {
         case MI_NORMAL_END:
            ret = mi_exec(cb_conn,"insert \
               into tran_state \
               values(\"Transaction Committed.\")\;",
               0);
            if ( ret == MI_ERROR )
               mi_db_error_raise(cb_conn, MI_EXCEPTION,
                  "Unable to save transaction \
                  state: Commit.");
            break;
         case MI_ABORT_END:
            ret = mi_exec(cb_conn,"insert \
               into tran_state \
               values(\"Transaction Aborted.\")\;",
               0);
            if ( ret == MI_ERROR )
               make_log_entry(log_file,
                  "Unable to save transaction state: \
                  Roll Back.");
            break;

         default:
            mi_exec(cb_conn,"insert into tran_state \
               values(\"Unhandled Transaction \
                  Event.\")\;", 0);
            break;
         }
```

```
        }
    else
        {
        mi_exec(cb_conn, "insert into tran_state \
            values(\"Unhandled Event.\");", 0);
        break;
        }

    mi_close(cb_conn);
    return MI_CB_CONTINUE;
}

MI_CALLBACK_STATUS MI_PROC_CALLBACK
eox_excpt_callback(event_type, conn, event_data, user_data)
    MI_EVENT_TYPE event_type;
    MI_CONNECTION *conn;
    void *event_data;
    void *user_data;
{
    ... Perform clean-up tasks ...
    return MI_CB_CONTINUE;
}
```

The database server invokes the **endxact_callback( )** callback with a transition state of MI_NORMAL_END just before it commits the transaction. The **endxact_callback( )** function executes as follows:

1. It executes the MI_NORMAL_END case in the **switch** statement.
2. If **mi_exec( )** in the MI_NORMAL_END case fails, **mi_exec( )** returns MI_ERROR, as does any DataBlade API function *except* **mi_db_error_raise( )**.
3. The condition in the **if** statement evaluates to TRUE and **mi_db_error_raise( )** executes, which raises an exception.
4. The DataBlade API invokes the registered exception callback, **eox_excpt_callback( )**.
5. Because **eox_excpt_callback( )** returns a status of MI_CB_CONTINUE, the database server aborts the transaction.

   If **eox_excpt_callback( )** had instead returned MI_CB_EXC_HANDLED, execution would continue at the next statement of the **endxact_callback( )** callback:

   ```
   mi_close(cb_conn);
   ```

   The **endxact_callback( )** function would then return MI_CB_CONTINUE, which would cause the database server to commit current transaction.
6. Before the database server aborts the transaction, it invokes **endxact_callback( )** with a transition state of MI_ABORT_END. This second invocation of **endxact_callback( )** proceeds as follows:

   a. It executes the MI_ABORT_END case in the **switch** statement.
   b. It calls **mi_exec( )** to execute the INSERT statement.
   c. If **mi_exec( )** fails, the database server invokes the registered exception callback **eox_excpt_callback( )**. The **mi_exec( )** function does *not* return MI_ERROR. Because **eox_excpt_callback( )** returns MI_CB_CONTINUE, control does not return to **endxact_callback( )**.

If **endxact_callback( )** had *not* registered its own exception callback, then when **mi_exec( )** in the MI_NORMAL_END case fails, execution would proceed as follows:

1. The **mi_exec( )** function returns MI_ERROR.

2. The condition in the **if** statement evaluates to TRUE and **mi_db_error_raise( )** executes, which raises an exception.

3. When **mi_db_error_raise( )** throws an exception, the database server aborts the transaction.

4. Before the database server aborts the transaction, it invokes **endxact_callback( )** with a transition state of MI_ABORT_END. This second invocation of **endxact_callback( )** proceeds as follows:

   a. Execute the MI_ABORT_END case in the **switch** statement.

   b. Call **mi_exec( )** to execute the INSERT statement.

   c. If **mi_exec( )** fails, it returns MI_ERROR.

   d. The condition in the **if** statement evaluates to TRUE and the user-defined **make_log_entry( )** function makes a log entry in a text file.

Unlike some other types of callbacks, an end-of-transaction callback can register callbacks of its own. The preceding end-of-transaction callback registers the **excpt_callback( )** to handle database server exceptions that might arise from the INSERT statements that **mi_exec( )** executes. For a sample implementation of the **excpt_callback( )** callback, see Figure 10-7 on page 10-30.

State Transitions in a Client LIBMI Application

In a client LIBMI application, the only state-transition event that might occur is MI_Xact_State_Change. The MI_Xact_State_Change event occurs *only* within a client application.

In a client LIBMI application, a state-change callback is invoked for the following state-transition types:

- An explicit begin transaction

  You execute an explicit begin with the SQL statement, BEGIN WORK.

  American National Standards Institute

- An ANSI-standard implicit begin

  In databases that are ANSI compliant, every SQL statement is a separate transaction. Therefore, ANSI-compliant databases execute an implicit begin for each SQL statement.

  End of American National Standards Institute

- An explicit end transaction

  You execute an explicit end transaction with one of the following SQL statements: COMMIT WORK (normal end) or ROLLBACK WORK (aborted end).

Client LIBMI Errors

The DataBlade API throws the MI_Client_Library_Error event to indicate an error in the client LIBMI library. The MI_Client_Library_Error event indicates the type of error that has occurred with one of the *error levels* that Table 10-7 shows.

Table 10-7. Client LIBMI Error Levels

Error Level	Description
MI_LIB_BADARG	Raised when a DataBlade API function receives an incorrect argument, such as a bad connection descriptor or a NULL value where a pointer is required

Table 10-7. Client LIBMI Error Levels (continued)

Error Level	Description
MI_LIB_BADSERV	Raised when the DataBlade API client library is unable to connect to a database server
MI_LIB_DROPCONN	Raised when the DataBlade API client library has lost the connection to the database server
MI_LIB_INTERR	Raised when an internal DataBlade API error occurs
MI_LIB_NOIMP	Raised when the called function or feature has not yet been implemented
MI_LIB_USAGE	Raised when a DataBlade API function is called out of sequence; for example, a call to mi_next_row( ) occurs when the statement did not return row data

To handle a client LIBMI error:

1. Write a callback that handles the MI_Client_Library_Error event.

   To handle an MI_Client_Library_Error event, you can write either of the following types of callback function:

   - A client LIBMI callback executes *only* when the MI_Client_Library_Error event occurs.
   - An all-events callback executes when many events occur and can include handling for the MI_Client_Library_Error event.

2. Register the callback function that handles the MI_Client_Library_Error event in the client LIBMI application that requires the error handling.

   Use the **mi_register_callback( )** function to register callback functions. After you register a callback that handles the MI_Client_Library_Error event, the client LIBMI invokes this callback instead of performing its default error handling.

Write a callback that handles MI_Client_Library_Error when you need to provide special handling for one or more client LIBMI errors, which Table 10-7 on page 10-55 shows. Within the callback, the **mi_error_level( )** function returns the error level for the client LIBMI error. You can also use the following DataBlade API functions to get more details about the client LIBMI error from its error descriptor:

- The **mi_error_sql_state( )** function returns an **SQLSTATE** value of "IX000" to indicate an Informix-specific error.
- The **mi_error_sqlcode( )** function returns the Informix-specific error.
- The **mi_errmsg( )** function returns the error message text.

For more information, see "Accessing an Error Descriptor" on page 10-18.

Important: Client LIBMI callbacks are subject to some restrictions on what tasks they can perform. For more information, see "Writing a Callback Function" on page 10-16.

The following sample code uses special-purpose handlers (not shown) to handle messages (**message_handler( )**) and database server exceptions (**exception_handler( )**). The **message_handler( )** routine might simply display a message on standard error, while the other handlers could take some specific user-defined action based on the type of exception.

```c
/* This routine dispatches callback events for the following
 * events:
 *      MI_Exception (client-side & server-side),
 *      MI_Client_Library_Error (client-side only)
 *      MI_Xact_State_Change (client-side only)
 */
#include "mi.h"

MI_CALLBACK_STATUS MI_PROC_CALLBACK
all_callback(event_type, conn, event_data, user_data)
   MI_EVENT_TYPE event_type;
   MI_CONNECTION *conn;
   void *event_data;
   void *user_data;
{
   mi_integer elevel;
   char err_msg[200];
   char *msg;

   switch ( event_type )
      {
      /* A database server exception calls a special-purpose
      ** handler to handle a message (warning) or an
      ** exception. */

      case MI_Exception:
         /* Obtain exception level from event */
         elevel = mi_error_level(event_data);

         switch ( elevel )
            {
            case MI_MESSAGE:
               message_handler(event_data, user_data);
               break;
            case MI_EXCEPTION:
               exception_handler(event_data,
                  user_data);
               break;
            }
         break;

      /* A client LIBMI error is any type of internal
      ** client-library error, library-usage problem, or
      ** a dropped connection. */
      case MI_Client_Library_Error:
         /* Obtain error level from event */
         elevel = mi_error_level(event_data);

         switch ( elevel )
            {
            case MI_LIB_BADARG:
               msg = "MI_LIB_BADARG";
               break;

            case MI_LIB_USAGE:
               msg = "MI_LIB_USAGE";
               break;

            case MI_LIB_INTERR:
               msg = "MI_LIB_INTERR";
               break;

            case MI_LIB_NOIMP:
               msg = "MI_LIB_NOIMP";
               break;

            case MI_LIB_DROPCONN:
```

```
                            msg = "MI_LIB_DROPCONN";
                            break;

                       default:
                           msg = "UNKNOWN";
                           break;
                       }
                   mi_errmsg(event_data, err_msg, 200);
                   fprintf(stderr, "%s: %s\n", msg, err_msg);
                   break;

               /* A transaction-state-change event occurs whenever
               ** the client LIBMI module begins or ends a
               ** transaction block. */
               case MI_Xact_State_Change:
                   {
                   mi_integer   change_type;

                   /* Obtain transition type from event */
                   change_type = mi_transition_type(event_data);

                   switch ( change_type )
                       {
                       case MI_BEGIN:
                           msg = "Transaction started.";
                           break;

                       case MI_NORMAL_END:
                           msg = "Transaction committed.";
                           break;

                       case MI_ABORT_END:
                           msg = "Transaction aborted!";
                           break;

                       default:
                           msg = "Unknown transaction type!";
                           break;
                       }
                   fprintf(stderr, "%s\n", msg);
                   break;
                   }

               /* No other types of events are expected here,
                * although they could happen. Let the user know
                * what happened and continue.
                */
               default:
                   fprintf(stderr,
                       "Caught an unexpected event type.\n");
                   break;
               }
           return MI_CB_CONTINUE;
        }
```

Server Only

The **all_callback( )** callback returns a status of MI_CB_CONTINUE when it is
invoked from a C UDR. Therefore, the database server would check for additional
callbacks that are registered for the event once it completed execution of
all_callback( ). If no additional callbacks existed, the database server would abort
the UDR.

End of Server Only

Chapter 11. Working with XA-Compliant External Data Sources

Overview of Integrating XA-Compliant Data Sources in Transactions

The Dynamic Server transaction manager, which is an integral part of the database server, not a separate module, can invoke support routines for each XA-compliant, external data source that participates in a distributed transaction at a particular transactional event, such as prepare, commit, or rollback. This interaction conforms to X/Open XA interface standards.

Transaction support in Dynamic Server for XA-compliant, external data sources, which are also called *resource managers*, enables you to:

- Create XA-compliant, external data source types and instances of XA-compliant, external data sources.
- Create XA purpose functions, such as xa_prepare, xa_commit, and xa_rollback, for each XA data source type to keep external data in sync with Dynamic Server transactional semantics.
- Register XA-compliant, external data sources with a Dynamic Server transaction.
- Unregister XA-compliant, external data sources.

- Use multiple XA-compliant, external data sources within the same global transaction.

The transaction coordination with an XA-compliant, external data source is supported only in Dynamic Server logged databases and ANSI-compliant databases. These databases support transactions. Transaction coordination with an XA-compliant, external data source is not supported in non-logged databases.

This chapter contains information on the following processes:
1. Creating user-defined XA-support routines
2. Creating and dropping XA-compliant, external data source types.
3. Creating and dropping XA-compliant, external data sources.
4. Registering and unregistering XA-compliant, external data sources as needed.

The IBM Informix MQ DataBlade Module, which provides a set of user-defined routines (UDRs) to enable Dynamic Server applications to exchange messages between Dynamic Server and IBM WebSphere® MQ, uses the XA data-source functionality described in this chapter. For information on the MQ DataBlade Module, see the *IBM Informix Database Extensions User's Guide*.

For general information on XA specifications, refer to the "Distributed Transaction Processing: The XA Specification." This is the X/Open standard specification that is available on the Internet.

Note: At the present time, the Dynamic Server transaction manager does not support the asychronous execution of purpose functions, thread migration, and transaction branch features.

Support for the Two-Phase Commit Protocol

Dynamic Server supports the two-phase commit protocol with registered XA-compliant data sources. The two-phase commit protocol ensures that transactions are uniformly committed or rolled back across multiple database servers. This protocol governs the order in which commit transactions are performed and provides a recovery mechanism in case a transaction does not execute.

XA-Compliant Data Sources and Data Source Types

An XA-compliant *data source type* is a type of data source that is capable of supporting XA protocol requirements for a resource manager that is participating in a transaction.

A *data source* is an instance of a data source type. An XA-compliant data source must be created in each database in which you want it, and the data source must be registered into the transaction manager when it participates in the transaction.

The transaction manager maintains a list of XA data sources participating in each transaction. Each XA data source participating in each transaction must dynamically register itself with the transaction manager by calling the **mi_xa_register_xadatasource( )** or **ax_reg( )** function at least once per transaction. If, for some reason, the XA data source does not participate in the two-phase commit protocol, it can be unregistered by calling the **mi_xa_unregister_xadatasource( )** or **ax_unreg( )** function.

Infrastructure for Creating Support Routines for XA Routines

The interaction between the Dynamic Server transaction manager and an XA data source occurs through a set of XA-support purpose functions, such as **xa_open( )**, **xa_close( )**, **xa_start( )**, **xa_end( )**, **xa_prepare( )**, **xa_commit( )**, **xa_rollback( )**, **xa_forget( )**, **xa_recover( )**, and **xa_complete( )**. These UDRs contain XA data source information and operate for the global transaction ID, which is passed as a parameter to a UDR.

The support routines must exist in every database in which the application creates XA data source types and their instances. You should create these UDRs before creating XA-compliant data source types. For more information, see "Creating User-Defined XA-Support Routines."

Global Transaction IDs

The Dynamic Server transaction manager generates and maintains a global transaction ID (of type **XID**) for each distributed XA transaction in the system. If a transaction already has a current transaction ID, the transaction manager uses that transaction ID for the global transaction and fills the XID structure with the transaction ID.

The XID structure is defined in the **xa.h** file in the in the **$INFORMIXDIR/incl/ public** directory.

System Catalog Tables

The **sysxasourcetypes** system catalog table stores information about XA data-source types. The **sysxadatasources** system catalog table stores information about XA data-source instances.

Files Containing Necessary Components

DataBlade modules obtain the XA-defined return values, the XID structure definition, and other necessary information from the from the **$INFORMIXDIR/incl/public/xa.h**file.

The prototypes for **mi_xa_register_xadatasource( )**, **mi_xa_unregister_xadatasource( )**, **mi_xa_get_xadatasource_rmid( )**, and **mi_xa_get_current_xid( )** are in the **$INFORMIXDIR/incl/public/milib.h** file.

Creating User-Defined XA-Support Routines

You can create user-defined XA-support routines, such as **xa_open( )**, **xa_start( )**, **xa_prepare( )**, **xa_rollback( )**, **xa_commit( )**, **xa_recover( )**, **xa_complete( )**, **xa_forget( )**, **xa_close( )**, and **xa_end( )**. These purpose functions, which are described in this section, are used for transaction management.

You use these functions when you create new XA data source types for external XA-compliant data sources.

Important: Even though Dynamic Server does not call all of the routines described in this section, you must have the routines to use as parameters when you create an XA data source type. In the CREATE XADATASOURCE TYPE statement, all of the parameters are mandatory. For an example of the CREATE XADATASOURCE TYPE statement, see "Creating an XA Data Source Type" on page 11-9.

There is no reason to invoke these functions directly from SQL or from user-defined routines.

The functions that you create must follow X/Open XA standards.

The xa_open( ) function

The **xa_open( )** function is called once per database session for each XA data source that participates in a Dynamic Server transaction. The **xa_open( )** function is called when a user-defined function registers the XA data source with a transaction by calling **mi_ register_xadatasource( )** or using **ax_reg( )** for the first time after the database opens.

Subsequent calls to **mi_register_xadatasource( )** or **ax_reg( )** in the same the same database session do not result in the invocation of the **xa_open( )** function.

The syntax for the function is:

```
mint xa_open(char *xa_info, /* IN */
          mint rmid,      /* IN */
          int4 flags)    /* IN */
```

Table 11-1. xa_open( ) Parameters

Parameter	Description
xa_info	Information string (*"session-id:databasename@servername"*)
rmid	Unique resource manager identifier
flags	TMNOFLAGS, the valid value

The following code fragment contains the **xa_open( )** function:

```
#include "xa.h"
mint mqseries_open(char *xa_info, mint rmid, int4 flags)
  {
/* setup the datastructures/etc.  */
if ( (myloc = mi_dalloc(sizeof(struct my_location), PER_SESSION))
    == (char *) NULL)
    {
    return XAER_RMERR;
    }
....
Return XA_OK;
  }
```

For valid return values, refer to X/Open information, including *Distributed Transaction Processing: The XA Specification*.

The xa_close( ) function

When the database is closed or the session ends, whichever occurs first, Dynamic Server calls the **xa_close( )** function for each XA data source used in the database session. This occurs when a new local database is opened, a database is closed, or a client session ends.

The **xa_close( )** function is also called when the XA data source is created and registered in a transaction and the transaction is rolled back.

When a database session is closed, all of the XA resources are freed and Dynamic server calls **xa_close( )** for any registered XA datasources in the database session.

The syntax for the function is:

```
mint xa_close(char *xa_info, /* IN */
         mint rmid,     /* IN */
         int4 flags)   /* IN */
```

Table 11-2. xa_close( ) Parameters

Parameter	Description
xa_info	Information string (*"session-id:databasename@servername"*)
rmid	Unique resource manager identifier
flags	TMNOFLAGS, the valid value

The following code fragment contains the **xa_close( )** function:

```
#include "xa.h"
mint mqseries_close(char *xa_info, mint rmid, int4 flags)
  {
/*  Error */
   return XAER_RMERR;
/* Success */
  return XA_OK;
}
```

For valid return values, refer to X/Open information, including *Distributed Transaction Processing: The XA Specification.*

The xa_start( ) function

In each transaction, when an XA data source registers statically with a Dynamic Server transaction, Dynamic Server invokes the **xa_start( )** function.

The syntax for the function is:

```
mint xa_start(XID *xid,   /* IN */
         mint rmid,     /* IN */
         int4 flags)   /* IN */
```

Table 11-3. xa_start( ) Parameters

Parameter	Description
xid	Pointer to the XID data structure that is defined in the **xa.h** file and used for the current transaction
rmid	Unique resource manager identifier
flags	TMNOFLAGS, which is the valid value

For valid return values, refer to X/Open information, including *Distributed Transaction Processing: The XA Specification.*

Note: Currently, Dynamic Server does not call the **xa_start( )** function because the XA data source should be created with the TMREGISTER flag.

The xa_end( ) function

For each XA data source participating in a transaction, the **xa_end( )** function is called before a direct rollback or before the prepare stage for a commit operation.

The syntax for the function is:

```
mint xa_end (XID *xid,     /* IN */
        mint rmid,     /* IN */
        int4 flags)   /* IN */
```

Table 11-4. xa_end( ) Parameters

Parameter	Description
xid	Pointer to the XID data structure that is defined in the **xa.h** file and used for the current transaction
rmid	Unique resource manager identifier
flags	TMSUCCESS, when commit issued, or TMFAIL, indicating the transaction will be rolled back

The following code fragment contains the **xa_end( )** function:

```
#include "xa.h"
mint mqseries_end(XID *xid, mint rmid, int4 flags)
  {
/*  Error  */
   return XAER_RMERR;
/* Success */
  return XA_OK;
}
```

For valid return values, refer to X/Open information, including *Distributed Transaction Processing: The XA Specification.*

The xa_prepare( ) function

The **xa_prepare( )** function prepares XA data source transaction changes for a commit or rollback operation. A successful return from **xa_prepare( )** indicates that the XA data source will successfully commit or rollback when requested.

The syntax for the function is:

```
mint xa_prepare (XID *xid,     /* IN */
        mint rmid,     /* IN */
        int4 flags)   /* IN */
```

Table 11-5. xa_prepare( ) Parameters

Parameter	Description
xid	Pointer to the XID data structure that is defined in the **xa.h** file and used for the current transaction
rmid	Unique resource manager identifier
flags	TMNOFLAGS, which is the valid value.

The following code fragment contains the **xa_prepare( )** function:

```
#include "xa.h"
mint mqseries_prepare(XID *xid, mint rmid, int4 flags)
  {
/*  Error  */
   return XAER_RMERR;
/* Success */
  return XA_OK;
}
```

For valid return values, refer to X/Open information, including *Distributed Transaction Processing: The XA Specification.*

The xa_rollback( ) function

The **xa_rollback( )** function is called if an application rolls back the transaction or if the prepare stage of the transaction fails.

The syntax for the function is:

```
mint xa_rollback (XID *xid,   /* IN */
          mint rmid,    /* IN */
          int4 flags)   /* IN */
```

Table 11-6. xa_rollback( ) Parameters

Parameter	Description
xid	Pointer to the XID data structure that is defined in the **xa.h** file and used for the current transaction
rmid	Unique resource manager identifier
flags	TMNOFLAGS, which is the valid value.

The following code sample code fragment contains the **xa_rollback( )** function:

```
#include "xa.h"
mint mqseries_rollback(XID *xid, mint rmid, int4 flags)
  {
/*  Error  */
   return XAER_RMERR;
/* Success */
  return XA_OK;
}
```

For valid return values, refer to X/Open information, including *Distributed Transaction Processing: The XA Specification*.

The xa_commit( ) function

The **xa_commit( )** function requests each participating XA data source to commit a transaction. If all of the XA data sources return TMSUCCESS on calls to **xa_end( )** and to **xa_prepare( )**, the database server calls **xa_commit( )** on each participating XA data source.

The syntax for the function is:

```
mint xa_commit (XID *xid,   /* IN */
         mint rmid,    /* IN */
        int4 flags)    /* IN */
```

Table 11-7. xa_commit( ) Parameters

Parameter	Description
xid	Pointer to the XID data structure that is defined in the **xa.h** file and used for the current transaction
rmid	Unique resource manager identifier
flags	TMNOFLAGS, which is the valid value.

The following code sample code fragment contains the **xa_commit( )** function:

```
#include "xa.h"
mint mqseries_commit(XID *xid, mint rmid, int4 flags)
  {
/*  Error  */
```

```
    return XAER_RMERR;
/* Success */
  return XA_OK;
}
```

For valid return values, refer to X/Open information, including *Distributed Transaction Processing: The XA Specification*.

The xa_recover( ) function

The **xa_recover( )** function obtains a list of prepared transaction branches from a resource manager.

The syntax for the function is:

```
mint xa_recover (XID *xids,      /* IN/OUT */
                      int4 count, /* IN */
                  mint rmid,      /* IN */
                  int4 flags)      /* IN */
```

Table 11-8. xa_recover( ) Parameters

Parameter	Description
xids	Array of the XID data structure that is defined in the **xa.h** file and used for the current transaction
count	Size of the array defined by the *count* argument
rmid	Unique resource manager identifier
flags	TMNOFLAGS, which is the valid value.

Note: Currently Dynamic Server does not call the **xa_recover( )** function.

The xa_forget( ) function

The **xa_forget( )** function enables an XA data source to disregard an heuristically completed transaction.

The syntax for the function is:

```
mint xa_forget (XID *xid,   /* IN */
         mint rmid,      /* IN */
         int4 flags)     /* IN */
```

Table 11-9. xa_forget( ) Parameters

Parameter	Description
xid	Pointer to the XID data structure that is defined in the **xa.h** file and used for the current transaction
rmid	Unique resource manager identifier
flags	TMNOFLAGS, which is the valid value.

The following code sample code fragment contains the **xa_forget( )** function:

```
#include "xa.h"
mint mqseries_forget(XID *xid, mint rmid, int4 flags)
  {
/*  Error  */
   return XAER_RMERR;
/* Success */
  return XA_OK;
  }
```

For valid return values, refer to X/Open information, including *Distributed Transaction Processing: The XA Specification.*

The xa_complete( ) function

The **xa_complete( )** function waits for an asynchronous operation to complete. If an operation is still pending, **xa_complete( )** waits for the completion and returns the status.

The syntax for the function is:

```
mint xa_complete (char  *handle,      /* IN*/
            int * retval,  /* OUT */
        mint rmid,        /* IN */
        int4 flags)      /* IN */
```

Table 11-10. xa_complete( ) Parameters

Parameter	Description
handle	Handle returned by the **xa_** call
retval	Return value of the **xa_** call that returned `handle`
rmid	Unique resource manager identifier
flags	`TMNOFLAGS`, which indicates no flags

For valid return values, refer to X/Open information, including *Distributed Transaction Processing: The XA Specification.*

Note: Currently Dynamic Server does not call the **xa_complete( )** function because Dynamic Server does not support asynchronous completion.

Dropping an XA Support User-Defined Routine

You cannot drop an XA support user-defined routine until all of the XA data source types that use the support function are dropped.

Managing XA Data Sources and Data Source Types

This section contains information on creating and dropping XA-compliant data source types and instances of XA-compliant data sources.

Creating an XA Data Source Type

You must create an XA data source type, before you create an XA data source.

To create an XA data source type, use the CREATE XADATASOURCE TYPE statement, as follows:

```
CREATE XADATASOURCE TYPE datasourcetypename (purpose options);
```

For example, in your code, you could specify information as shown in the following example:

```
create function "informix".mqseries_open(lvarchar(256), int, int) returns int;
    external name '$USERFUNCDIR/mqseries.udr(mqseries_open)' language c;

grant execute on function "informix".mqseries_open(lvarchar(256), int, int)
    to public;

create function "informix".mqseries_close(lvarchar(256), int, int) returns int;
    external name '$USERFUNCDIR/mqseries.udr(mqseries_close)' language c;
```

```
grant execute on function "informix".mqseries_close(lvarchar(256), int, int)
    to public;

create function "informix".mqseries_start(informix.pointer, int, int) returns int;
    external name '$USERFUNCDIR/mqseries.udr(mqseries_start)' language c;

grant execute on function "informix".mqseries_start(informix.pointer, int, int)
    to public;

create function "informix".mqseries_end(informix.pointer, int, int) returns int;
    external name '$USERFUNCDIR/mqseries.udr(mqseries_end)' language c;

grant execute on function "informix".mqseries_end(informix.pointer, int, int)
    to public;

create function "informix".mqseries_rollback(informix.pointer, int, int)
    returns int;
     external name '$USERFUNCDIR/mqseries.udr(mqseries_rollback)' language c;
grant execute on function "informix".mqseries_rollback(informix.pointer, int, int)
    to public;

create function "informix".mqseries_prepare(informix.pointer, int, int) returns int;
     external name '$USERFUNCDIR/mqseries.udr(mqseries_prepare)' language c;

grant execute on function "informix".mqseries_prepare(informix.pointer, int, int)
    to public;

create function "informix".mqseries_commit(informix.pointer, int, int) returns int;
     external name '$USERFUNCDIR/mqseries.udr(mqseries_commit)' language c;

grant execute on function "informix".mqseries_commit(informix.pointer, int, int)
    to public;

create function "informix".mqseries_forget(informix.pointer, int, int) returns int;
     external name '$USERFUNCDIR/mqseries.udr(mqseries_forget)' language c;

grant execute on function "informix".mqseries_forget(informix.pointer, int, int)
    to public;

create function "informix".mqseries_recover(informix.pointer, int, int, int)
    returns int;
     external name '$USERFUNCDIR/mqseries.udr(mqseries_recover)' language c;

grant execute on function "informix".mqseries_recover(informix.pointer, int, int,
    int) to public;

create function "informix".mqseries_complete(informix.pointer, informix.pointer,
    int, int) returns int;
     external name '$USERFUNCDIR/mqseries.udr(mqseries_complete)' language c;

grant execute on function "informix".mqseries_complete(informix.pointer,
    informix.pointer int, int) to public;
CREATE XADATASOURCE TYPE informix.MQSeries
(xa_flags = 1,
xa_version  = 0,
xa_open     = informix.mqseries_open,
xa_close    = informix.mqseries_close,
xa_start    = informix.mqseries_start,
xa_end      = informix.mqseries_end,
xa_rollback = informix.mqseries_rollback,
xa_prepare  = informix.mqseries_prepare,
xa_commit   = informix.mqseries_commit,
xa_recover  = informix.mqseries_recover,
xa_forget   = informix.mqseries_forget,
xa_complete = informix.mqseries_complete);
```

In this statement, *xa_flags* must be set to 1.

You must provide one value for each of the options listed above, but not necessarily in the sequence shown above.

ANSI database and non-ANSI database namespace rules apply for the XA data source type, XA data source names, and user-defined function names.

After you create a data source type, information on the data source type is stored in the **sysxasourcetypes** system catalog table.

For syntax details, see the *IBM Informix Guide to SQL: Syntax*.

Dropping an XA Data Source Type

You cannot drop an XA data source type until all of the XA data source instances that use the type are dropped.

To drop an XA data source type, use the DROP XADATASOURCE TYPE statement as follows:

```
DROP XADATASOURCE TYPE datasourcetypename RESTRICT;
```

For example:

```
DROP XADATASOURCE TYPE informix.MQSeries RESTRICT;
```

For syntax details, see the *IBM Informix Guide to SQL: Syntax*.

Creating an XA Data Source

To create an XA data source, use the CREATE XADATASOURCE statement, as follows:

```
CREATE XADATASOURCE datasourcename USING datasourcetypename;
```

The *datasourcename* and *datasourcetypename* optionally include the owner name separated from the data source type name by a period.

For example:

```
CREATE XADATASOURCE informix.NewYork USING informix.MQSeries;
```

ANSI database and non-ANSI database namespace rules apply for the XA data source type, XA data source names, and user-defined function names.

Each new XA data source will have a unique ID in the **sysxadatasources** system catalog table.

For syntax details, see the *IBM Informix Guide to SQL: Syntax*.

Dropping an XA Data Source

Use the DROP XADATASOURCE statement to drop an XA data source.

DROP XADATASOURCE *datasourcename* RESTRICT;

For example:

```
DROP XADATASOURCE informix.NewYork RESTRICT;
```

The XA data source must already exist in the system because it was previously created with a CREATE XADATASOURCE TYPE statement. Information on the data source type is stored in the **sysxasourcetypes** system catalog table.

If an XA data source has been registered with a transaction that is not complete, the data source can be dropped only if the database is closed or the database session exits.

For syntax details, see the *IBM Informix Guide to SQL: Syntax*.

Registering and Unregistering XA-Compliant Data Sources

After you create an external XA-compliant data source, transactions can register and unregister the data source using the **mi_xa_register_xadatasource( )** or **ax_reg( )** and **mi_xa_unregister_xadatasource( )** or **ax_unreg( )** functions.

The **mi_xa_register_xadatasource( )** function and the **ax_reg( )** function both register XA-compliant, external data sources. However, these functions use different parameters and have different return values.

Similarly, the **mi_xa_unregister_xadatasource( )** and **ax_unreg( )** functions perform the same operation, but use different parameters and have different return values.

Unlike SQL CREATE operations, which create entries in system catalog tables, registration is transient, lasting only for the duration of the transaction. A transaction must be started, implicitly or explicitly, before the application can register the XA data source.

In a distributed environment, you must register a data source using the local coordinator server.

Multiple registrations of the same XA data source in a transaction have the same effect as a single registration. Dynamic Server does not maintain a count of the number of times an application has registered. A single call to **ax_unreg( )** or **mi_xa_unregister_xadatasource( )** unregisters the data source from the transaction.

Using ax_reg( )

The **ax_reg( )** function registers an XA data source with the current transaction. This function must be repeated with each new transaction.

Use the following syntax for an **ax_reg( )** function:

```
int ax_reg(int rmid, XID *xid, int4 flags)
```

For example:

```
   #include "xa.h"
    #include "milib.h"
    int   rmid, retcode;
   XID  *xid;
  if ( (rmid = mi_xa_get_xadatasource_rmid("informix.Newyork")) <= 0)
      {
        /* Error while getting XA data source id */
      }
   if ( !(xid = (XID *)mi_alloc(sizeof(XID)) ))
        {
         /* Memory allocation error */
        }
  retcode  = ax_reg(rmid, xid, TMNOFLAGS);
```

```
if (retcode != TM_OK )
    {
        /* ax_reg() Error  */
    }
 /* ax_reg() is Successful */
```

When you use the **ax_reg( )** function, follow these guidelines:

- Be sure the correct *rmid* (resource manager ID) is correct.

  You can use the **mi_xa_get_xadatasource_rmid( )** function to enable the DataBlade module to get the ID the correct *rmid* value.

  The resource manager ID must be present in a row in the **sysxadatasources** system catalog table that was created with the CREATE XADATASOURCE statement of SQL.

- *xid* is a valid pointer to the XID data structure, which is defined in the **$INFORMIXDIR/incl/public/xa.h** file. Make sure that memory for *xid* is allocated.

- Set *flags* to TMNOFLAGS. The value for TMNOFLAGS is defined in the **$INFORMIXDIR/incl/public/xa.h** file.

- Only call the **ax_reg( )** function within an explicit or implicit transaction.

- Do not call the **ax_reg( )** function in these contexts:
  - From the sub-ordinator of a distributed transaction or from within a resource manager global transaction

    Dynamic Server can operate as a resource manager in a global transaction managed by a third party transaction manager. The **ax_reg( )** function should not be used if Dynamic Server is operating as a resource manager.
  - In a non-logging database
  - From any of the XA purpose functions that are specified in a CREATE XADATASOURCE TYPE statement, which creates a type of XA-compliant external data source.

Multiple registrations of the same XA data source in a single transaction do not effect either the transaction or the XA data source.

For more information on this function, see the *IBM Informix DataBlade API Function Reference*.

Using ax_unreg( )

The **ax_unreg( )** function unregisters the previously registered XA data source from the transaction.

By default, all XA-compliant external data sources are unregistered at the end of a transaction. Use the **ax_unreg( )** function to unregister the data source before the end of the transaction so the data source does not participate in the transaction.

Use the following syntax for an **ax_unreg( )** function:

```
int ax_unreg(int rmid, int4 flags)
```

For example:

```
    #include "xa.h"
    #include "milib.h"
    int   rmid, retcode;
  if ( (rmid = mi_xa_get_xadatasource_rmid("informix.Newyork")) <= 0)
      {
       /* Error while getting XA data source id */
```

```
        }
    retcode  = ax_unreg(rmid,  TMNOFLAGS);
    if (retcode != TM_OK )
       {
           /* ax_uunreg() Error  */
       }
 /* ax_unreg() is Successful */
```

When you use the **ax_unreg( )** function, follow these guidelines:

- Get the correct *rmid* (resource manager ID) value to use in the syntax for this function. If you do not know the resource manager ID, you can use the **mi_xa_get_xadatasource_rmid( )** function to get the ID.
- Make sure the *flags* are passed as TMNOFLAGS.
- Only call the **ax_unreg( )** function within an explicit or implicit transaction.
- Do not call the **ax_unreg( )** function from these contexts:
  - From the sub-ordinator of a distributed transaction
  - From within a resource manager global transaction
  - In a non-logging database
  - From any of the XA purpose functions that are specified in a CREATE XADATASOURCE TYPE statement.
- Do not unregister an XA data source that is not registered or already unregistered.

For more information on this function, see the *IBM Informix DataBlade API Function Reference*.

Using mi_xa_register_xadatasource( )

The **mi_xa_register_xadatasource( )** function registers an XA data source with the current transaction. This function must be repeated with each new transaction. This function operates the same way as the **ax_reg( )** function.

Use the following syntax for an **mi_xa_register_xadatasource( )** function:

```
mi_integer mi_xa_register_xadatasource(mi_string *xasrc)
```

For example:

```
    #include "xa.h"
    #include "milib.h"
    int     retcode;
  retcode = mi_xa_register_xadatasource("informix.Newyork"));
  if ( retcode != MI_OK)
     {
       /* Error while registering the XA data source   */
     }
         /* Success fully registered */
```

When you use the **mi_xa_register_xadatasource( )** function, follow these guidelines:

- Get the correct value for *xarsc*, which is the user-defined name of the XA data source. The format of the *xarsc* name is *owner.xadatasourcename*.
- Only call the **mi_xa_register_xadatasource( )** function within an explicit or implicit transaction.
- Do not call the **mi_xa_register_xadatasource( )** function:
  - From the sub-ordinator of a distributed transaction or from within a resource manager global transaction

Dynamic Server can operate as a resource manager in a global transaction managed by a third party transaction manager. The **mi_xa_register_xadatasource( )** function should not be used if Dynamic Server is operating as a resource manager.

- In a non-logging database
- From any of the XA purpose functions that are specified in a CREATE XADATASOURCE TYPE statement.

Multiple registrations of the same XA data source in a single transaction do not effect either the transaction or the XA data source.

For more information on this function, see the *IBM Informix DataBlade API Function Reference*.

Using mi_xa_unregister_xadatasource( )

The **mi_xa_unregister_xadatasource( )** function unregisters the previously registered XA data source from the transaction.

Use the following syntax for an **mi_xa_unregister_xadatasource( )** function:

```
mi_integer mi_xa_unregister_xadatasource(mi_string *xasrc)
```

For example:

```
    #include "xa.h"
    #include "milib.h"
    int    retcode;
 retcode = mi_xa_unregister_xadatasource("informix.Newyork"));
 if ( retcode != MI_OK)
    {
     /* Error while unregistering the XA data source   */
    }
        /* Successfully unregistered */
```

When you use the **mi_xa_unregister_xadatasource( )** function, follow these guidelines:

- Get the correct value for *xarsc*, which is the user-defined name of the XA data source. The format of the *xarsc* name is *owner.xadatasourcename*.

  You can use the **mi_xa_get_current_xid( )** function to return the pointer to the current XID structure for an XA-compliant, external data source.
- Only call the **mi_xa_unregister_xadatasource( )** function within an explicit or implicit transaction.
- Do not call the **mi_xa_unregister_xadatasource( )** function:
  - From the sub-ordinator of a distributed transaction
  - From within a resource manager global transaction
  - In a non-logging database
  - From any of the XA purpose functions that are specified in a CREATE XADATASOURCE TYPE statement.
- Do not unregister an XA data source that is not registered or already unregistered.

For more information on this function, see the *IBM Informix DataBlade API Function Reference*.

Getting the XID Structure

The **mi_xa_get_current_xid( )** function returns the pointer to the current XID structure for an XA-compliant, external data source. The XID structure is defined in the **$INFORMIXDIR/incl/public/xa.h** file.

This XID structure that is returned is valid only until the user-defined routine terminates. The calling user-defined routine must copy this function if it is necessary to keep the data for a longer period of time.

The syntax for the function is:

```
XID * mi_xa_get_current_xid ( )
```

For example:

```
    #include "xa.h"
    #include "milib.h"
XID * xid;
xid = mi_xa_get_current_xid();
if ( (!xid)
        {
          /* Error while getting the curret XID */
        }
 /* Successful */
```

If successful, this function returns the pointer to the XID structure, which is not null.

For more information on this function, see the *IBM Informix DataBlade API Function Reference*.

Getting the Resource Manager ID

When specifying how user-defined routines register and unregister a data source using the **ax_reg( )** or **ax_unreg( )** function, you can use the **mi_xa_get_xadatasource_rmid( )** function to get the resource manager ID that was previously created in the database for an XA-compliant, external data source. The **mi_xa_get_xadatasource_rmid( )** function can be used while invoking the **ax_reg( )** or the **ax_unreg( )** function in subsequent calls.

The syntax for the function is:

```
mi_integer mi_xa_get_xadatasource_rmid(mi_string *xasrc)
```

xasrc is the user-defined name of the XA data source.

For example:

```
    #include "xa.h"
    #include "milib.h"
    int   rmid;
  rmid = mi_xa_get_xadatasource_rmid("informix.Newyork");
  if (rmid  <= 0)
      {
        /* Error while getting XA data source id */
      }
/* Successful */
```

For more information on this function, see the *IBM Informix DataBlade API Function Reference*.

Monitoring Integrated Transactions

Use the following **onstat** options to display information about transactions involving XA-compliant data sources:

Table 11-11.

onstat Option	What XA Data Source Information This Command Displays
onstat -x	Displays information on XA participants in a transaction.
onstat -G	Displays information on XA participants in a global transaction.
onstat -g ses *session id*	Displays session information, including information about XA data sources participating in a transaction.

Part 4. Creating User-Defined Routines

Chapter 12. Developing a User-Defined Routine

In This Chapter

A C user-defined routine (UDR) is a UDR that is written in the C language and uses the server-side implementation of the DataBlade API to communicate with the database server. C UDRs (functions and procedures) are implemented as C-language functions. DataBlade modules often include C UDRs that are made available for use by registering them in the database.

Tip: The terms "C UDR" and "UDR" are used interchangeably in this publication.

The development process for a C UDR follows these steps:

1. Design the use and development process for the UDR
2. Code a C routine that uses the DataBlade API functions to interact with the database server
3. Compile and link the C routine to create a shared-object file
4. Register the C routine with the CREATE FUNCTION or CREATE PROCEDURE statement
5. Execute the UDR
6. Use tracing and the debugging features to work out any problems in the UDR
7. Change any characteristics of the UDR that are required during its lifetime
8. Optimize performance of the UDR

This chapter describes each of these steps in the development of a C UDR. For general information on the development steps of a UDR, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

--- **Client Only** ---

This chapter covers topics specific to the development of a C UDR. This material does *not* apply to the creation of client LIBMI applications with the client-side implementation of the DataBlade API.

--- **End of Client Only** ---

Designing a UDR

This section provides the following design considerations for the development of a C UDR:

- Development tools
- Uses of a UDR
- Portability
- Insert and update operations

Development Tools

The creation of a C UDR involves the production of source code, header files, SQL statements, and many other files. This publication describes how to generate the code for C UDRs yourself, using the DataBlade API and the basic tools available with an operating system.

However, IBM provides a package of development tools, called the *Informix DataBlade Developers Kit* (DBDK), that helps you build and manage the C UDRs of a DataBlade module project. A DataBlade module is a package of software that extends the functionality of the database server. It can include the following objects:

- New or extended data types
- UDRs
- Error messages
- Functional tests
- Interfaces to other DataBlade modules
- Packaging and installation scripts

Windows Only

The DBDK runs on Windows. The following table summarizes the development tools of the DBDK.

DBDK Development Tool	Description
BladeSmith	• Provides an interactive wizard and generates code for many C UDRs • Generates a *project*, with as much code as possible, including source for opaque-type support functions, header files, a makefile for compilation, SQL statements, and functional tests
BladePack	• Understands the contents of a project that BladeSmith produces, enabling it to be extended to include documentation and online help • Produces a releasable package for a DataBlade module
BladeManager	Understands the contents of the releasable package that BladePack creates, enabling it to install a DataBlade module in a database

These development tools include online help to describe their use. The *IBM Informix DataBlade Developers Kit User's Guide* is provided to describe these tools.

End of Windows Only

UNIX/Linux Only

BladeSmith can develop source code and a makefile that can be compiled on UNIX or Linux.

End of UNIX/Linux Only

Consider using the development tools of the DBDK to generate the initial code for your C UDRs. You can then use the information in this publication to enhance and change this code to handle the unique needs of your C UDR or DataBlade module.

Uses of a C UDR

The following table summarizes the tasks that a C UDR can perform. It also describes where you can find additional information in this publication for each of these UDR uses.

Type of UDR	Purpose	More Information
Cast function	A UDR that converts one data type to another	"Writing a Cast Function" on page 15-2
Cost function	A UDR that determines the cost of execution for an expensive UDR	"Writing Selectivity and Cost Functions" on page 15-54
End-user routine	A UDR that performs some common task for an end user	"Writing an End-User Routine" on page 15-2
Iterator function	A function that returns more than one row of data	"Writing an Iterator Function" on page 15-3
Opaque-type support function	One of a group of user-defined functions that tell the database server how to handle the data of an opaque data type	"Creating an Opaque Data Type" on page 16-1
Operator-class function	User-defined functions that define operators to use with a particular secondary access method	See your access-method documentation.
Negator function	A user-defined function that calculates the Boolean NOT operation for a particular operator or function.	"Creating Negator Functions" on page 15-60
Selectivity function	A UDR that determines the percentage of rows likely to be returned by an expensive UDR	"Writing Selectivity and Cost Functions" on page 15-54
Parallelizable UDR	A UDR that can run in parallel when executed within a PDQ statement	"Creating Parallelizable UDRs" on page 15-61
User-defined aggregate	A function that calculates an aggregate value on a particular column or value	"Writing an Aggregate Function" on page 15-11

Portability

To ensure portability of your C UDR, include the following items in the design and implementation of your C UDR:

- Use the DataBlade API data types for data types whose size might vary across computer architectures.
- Use the DataBlade API functions to transfer data between client and server computers.

DataBlade API Data Types

The DataBlade API provides platform-independent data types, such as **mi_smallint** (two-byte integer), **mi_integer** (four-byte integer), and **mi_double_precision** (floating-point values). For a complete list of DataBlade API data types, see Table 1-1 on page 1-8. The **mitypes.h** header file defines these data types.

Tip: The **mi.h** header file automatically includes the **milib.h** header, which in turn includes the **mitypes.h** header file. Therefore, you do not need to explicitly include **mitypes.h** to use the DataBlade API data types.

To ensure maximum portability of your code, use these platform-independent data types instead of their C-language equivalents.

Data Conversion

The DataBlade API provides special functions to handle the following data conversions that a C UDR might need to perform:

- Data conversion between the text and binary representations of the data

The control mode of a query determines whether the query results are in text or binary representation. The DataBlade API provides functions that convert between these two representations. Conversion between these two representations might also be useful in the input and output support functions of an opaque type. For a list of these functions, see "Conversion of Opaque-Type Data Between Text and Binary Representations" on page 16-16.

- Data transfer between a client application and the database server

When opaque-type data is transferred between a client application and the database server, the database server calls the send and receive support functions. For these UDRs, you can handle potential differences in computer architecture that might affect the byte ordering or size with special DataBlade API functions. For more information, see "Conversion of Opaque-Type Data with Computer-Specific Data Types" on page 16-21.

Insert and Update Operations

An SPL routine has the restriction that it cannot perform INSERT or UPDATE operations in an SPL routine that is invoked from a DML statement. This restriction ensures that the SPL routine cannot change the state of the statement that invoked it.

This restriction is relaxed for UDRs. The database server issues an error if the table that is being accessed in the UDR is referenced in the statement that invoked it. If this is a nested UDR invocation, then the database server checks the chain of parent queries.

If a UDR is called as part of an INSERT, UPDATE, DELETE, or SELECT statement and if the referenced table appears in the chain of statements that eventually invoked the UDR, the called routine cannot execute any of the following statements:

- ALTER FRAGMENT
- ALTER INDEX
- ALTER OPTICAL
- ALTER TABLE
- DROP INDEX
- DROP OPTICAL
- DROP SYNONYM
- DROP TABLE
- DROP TRIGGER
- DROP VIEW
- RENAME COLUMN
- RENAME TABLE

Creating UDR Code

This section provides an overview of C UDR development:

- Variable declarations
- Session management
- SQL statement execution
- Routine-state information
- Event handling

- Well-behaved routines

Tip: For a discussion of special implementation issues specific to a C UDR, see Chapter 13, "Writing a User-Defined Routine," on page 13-1.

Variable Declaration

In a C UDR, you declare variables to hold information just as you would in a C function. The DataBlade API provides many data types for variable declaration. The DataBlade API prepends special prefixes to the names of its data types, as the following table shows.

DataBlade API Data Type	Data Type Prefix	More Information
Data types to hold SQL data types	**mi_**	Table 1-1 on page 1-8
Data type structures that hold information for DataBlade API functions	**MI_**	Table 1-4 on page 1-12

Table 1-2 on page 1-10 lists where in this publication each of the SQL data types is discussed in detail.

Use the DataBlade API data types for variable declaration in your C UDR even if there is a C-language equivalent. The DataBlade API data types are more portable if your C UDR moves to different computer architectures. For more information, see "DataBlade API Data Types" on page 12-4.

Session Management

Session management is different in a C UDR than in a client LIBMI application. Unlike a client LIBMI application, which can simultaneously connect to several databases, a UDR inherits a particular session context. That is, it is within an existing session and uses a database that is already opened. For information, see "Establishing a UDR Connection (Server)" on page 7-11.

Because a C UDR can establish only a UDR connection, not a client connection, restrictions apply in the following areas:
- Sessions
- Transactions

Session Restrictions

The following restrictions exist within a UDR with respect to the session:
- Session-connection restrictions

  A UDR runs within an existing session. It can only obtain a connection to this session after a client application has already begun the session. However, a UDR can obtain more than one connection to the session.
- Cursor restrictions
  - Cursors opened in one session are only visible in the context of that session. Cursors defined on one UDR are not visible in other UDRs. Therefore, multiple UDRs can use the same cursor names.
  - A cursor opened in a session persists until an **mi_close( )** or an **mi_drop_prepared_statement( )** function, or until the end of the statement that called the UDR (if **mi_close( )** was not called). Therefore, cursors last across routine invocations.
- Database restrictions
  - A UDR cannot connect to a remote database server.

– A UDR uses the default database, which the client application has established. However, it cannot change the database.
- Table restrictions

  The scope of temporary tables created in a logging database is the current session. Temporary tables created in a database that does not use logging or with a CREATE TABLE statement that includes the WITH NO LOG clause persist until beyond the CLOSE DATABASE statement.
- Constraint restrictions

  Violations that are associated with the execution of the UDR are added to a violation temporary table. Therefore, if the SET CONSTRAINTS statement sets the constraint mode to IMMEDIATE, constraint checking is done *per statement*. Otherwise, constraint checking is deferred to the time when the transaction is committed. If the constraint mode is set to IMMEDIATE, the constraint is checked after each statement in the UDR. If you want per-UDR constraint checking, change the constraint mode to DEFERRED at the *beginning* of the UDR and back to IMMEDIATE at the end of the UDR.

Transaction Management

Against databases that use logging, a UDR inherits the transaction state that is started by the SQL statement that invoked the UDR. All statements in a UDR occur inside a transaction because the UDR is called from within an SQL statement. An SQL statement is *always* part of a transaction. The type of transaction that the SQL statement is part of is based on the type of database and whether it uses transaction logging, as Table 12-1 shows.

Table 12-1. Types of Transactions

Status of Database	Status of SQL Statement	Description
Database is not ANSI-compliant:		
Database does *not* use transaction logging. Database does use transaction logging.	No transactions exist.	The database server does *not* log changes to the database that SQL statements might make. Any UDRs that are part of the SQL statement are not logged and their actions cannot be rolled back.
	Each SQL statement is within either an explicit transaction or a single-statement transaction:	
	• Explicit transaction	The client application begins an *explicit transaction* with the BEGIN WORK statement and ends it with either the COMMIT WORK statement (transaction successful) or the ROLLBACK WORK statement (transaction not successful). Operations within a single cursor (from OPEN to CLOSE) constitute a transaction as well. SQL statements between the BEGIN WORK and COMMIT WORK or ROLLBACK WORK statements (or within a cursor) execute within the explicit transaction. If these SQL statements contain any UDRs, each of the UDRs executes within the explicit transaction.
	• Single-statement transaction	The client application begins a *single-statement transaction* for any SQL statement that is *not* contained within a BEGIN WORK statement and a COMMIT WORK or ROLLBACK WORK statement. Any UDRs that are part of the SQL statement are within this single-statement transaction. The only exception to this rule is the EXECUTE FUNCTION statement; it does *not* execute within a transaction.
Database is ANSI-compliant:		

Table 12-1. Types of Transactions (continued)

Status of Database	Status of SQL Statement	Description
Database logging is always in effect.	Each SQL statement executes within an *implicit transaction*, which is always in effect.	The client application invokes an SQL statement, which begins the implicit transaction, and the transaction ends explicitly with COMMIT WORK or ROLLBACK WORK. Any UDRs within the SQL statement that began the implicit transaction are automatically part of the transaction. In addition, any SQL statements that execute before the COMMIT WORK or ROLLBACK WORK statement ends the transaction are also part of the implicit transaction.

You can obtain the transaction ID of the current transaction with the **mi_get_transaction_id( )** function.

As a rule, a C UDR must *not* issue any of the following SQL transaction statements because they interfere with transaction boundaries:

- BEGIN WORK
- COMMIT WORK
- ROLLBACK WORK

In all databases that use logging, an SQL statement is within a transaction. In such databases, a DML statement (SELECT, INSERT, UPDATE, DELETE) implicitly starts a transaction, if a transaction is not already in effect. If a UDR that executes one of these SQL transaction statements is called from a DML statement, the database server raises an error (-535).

However, the EXECUTE PROCEDURE and EXECUTE FUNCTION statements do *not* implicitly start another transaction, if they are not already in a transaction. If a UDR is called from an EXECUTE PROCEDURE or EXECUTE FUNCTION statement, the database server only raises an error if the UDR interferes with the current transaction boundaries.

For example, suppose you have a UDR named **udr1( )** that uses the **mi_exec( )** function to execute two SQL statements:

```
void udr1(...)
{
   mi_exec(...DML statement 1...);
   mi_exec(...DML statement 2...);
}
```

Suppose also that you execute this UDR with the EXECUTE PROCEDURE statement, as follows:

```
EXECUTE PROCEDURE udr1( );
```

If a transaction has not already been started, this UDR would have two transactions, one for each call to **mi_exec( )**.

To get a single transaction, you could surround these SQL statements with a begin and end work, as **udr2( )** shows:

```
void udr2(...)
{
   mi_exec(...'begin work'..);
```

```
   mi_exec(...DML statement 1...);
   mi_exec(...DML statement 2...);
   mi_exec(...'commit work'...);
}
```

However, you can *only* start a transaction within a UDR if you are *not* already in a transaction. Therefore, you can only invoke a UDR that starts a transaction when the following restrictions are met:

- You *must* invoke the UDR with the EXECUTE PROCEDURE or EXECUTE FUNCTION statement.

  Because **udr2( )** is a user-defined procedure, you must use EXECUTE PROCEDURE to invoke it, as follows:

  ```
  EXECUTE PROCEDURE udr2( );
  ```

  Suppose you tried to invoke **udr2( )** with the following SELECT statement:

  ```
  SELECT udr2( ) FROM tab WHERE x=y;
  ```

  If a transaction had not started, the SELECT operation starts its own implicit transaction. The database server raises an error when execution reaches the first call to **mi_exec( )** in **udr2( )**:

  ```
  mi_exec(...'begin work'..);
  ```

- The UDR calling context must *not* have already started a transaction.

  The following code fragment *fails* because the EXECUTE PROCEDURE statement is already within a transaction block and **udr2( )** attempts to start another transaction:

  ```
  BEGIN WORK;
  ...
  EXECUTE PROCEDURE udr2( );  /* This statement fails. */
  ...
  COMMIT WORK:
  ```

  The database server raises an error when execution reaches the first call to **mi_exec( )** in **udr2( )**:

  ```
  mi_exec(...'begin work'..);
  ```

Important: Unless a UDR knows its calling context, it should not issue an SQL transaction statement. If the caller has already begun a transaction, the UDR fails.

You can execute an SQL transaction statement in a UDR that you call directly from a DataBlade API module (not from within an SQL statement). You can also choose whether to commit or rollback the current transaction from within an end-of-statement or end-of-transaction callback function. For more information, see "State Transitions in a C UDR (Server)" on page 10-51.

In a database with logging, the database server creates an internal savepoint before execution of each statement within a UDR that might affect the database state. If one of these statements fails, the database server performs a rollback to this internal savepoint. At this point, the database server does *not* release table locks. However, the same user can obtain a lock on the same table in the same transaction. The database server releases the table lock when the entire transaction ends (commit or rollback).

Warning: For databases that do not use logging, no changes to the database that a UDR might make are logged. Therefore, none of these changes can be rolled back. Consider carefully whether you want to use logging for your database.

SQL Statement Execution

The differences in the execution of SQL statements in a C UDR and a client LIBMI application are because of the differences in passing mechanisms that they use for the contents of an **MI_DATUM** structure. In a C UDR, you must consider the data type of the value in the **MI_DATUM** structure to determine how to obtain the value. For more information on the passing mechanism for an **MI_DATUM** value, see "Contents of an MI_DATUM Structure" on page 2-33.

In SQL statement execution, the DataBlade API uses an **MI_DATUM** structure for the following values:

- Input-parameter value that the DataBlade API sends to a prepared statement
- Column value that the DataBlade API retrieves from a query

Setting Input Parameters

When you send a prepared statement for execution, you pass any input-parameter values in **MI_DATUM** structures. Therefore, the data type of the column associated with an input parameter determines the passing mechanism for the input-parameter value, as follows:

- For data types that are passed by value, the **MI_DATUM** structure must contain the *actual* input-parameter value.
- For data types that are passed by reference, the **MI_DATUM** structure must contain *a pointer to* the input-parameter value.

Within your C UDR, you must use the column data type to determine how to assign the input-parameter value in the **MI_DATUM** structure. To assign the input-parameter values, you send an array of **MI_DATUM** structures to the **mi_exec_prepared_statement( )** or **mi_open_prepared_statement( )** function, which sends the prepared statement to the database server for execution. For more information, see "Assigning Values to Input Parameters" on page 8-27.

Retrieving Column Values

When you execute a query (SELECT or EXECUTE FUNCTION statement) in a C UDR, you choose a control mode for the retrieved data. If the query data is in binary representation, the column value that **mi_value( )** or **mi_value_by_name( )** passes back is in an **MI_DATUM** structure. Therefore, the size of the data type associated with the column determines the passing mechanism for the column value, as follows:

- For data types that are passed by value, the **MI_DATUM** structure contains the *actual* column value.
- For data types that are passed by reference, the **MI_DATUM** structure contains *a pointer to* this column value.

Within your C UDR, you must use the column data type to determine how to obtain the column value in the **MI_DATUM** structure. For more information, see "Obtaining Column Values" on page 8-42.

Routine-State Information

When the routine manager executes a C UDR, it puts information about the routine sequence for the UDR in an **MI_FPARAM** structure and passes this **MI_FPARAM** structure as the last argument to the UDR. From the **MI_FPARAM** structure, the UDR can obtain the following information:

- The number and data types of its arguments
- The assumed data type of its return value (for a user-defined function only)

- Any user data associated with the UDR

For information about how the routine manager executes a UDR, see "Executing a UDR" on page 12-18. For information about how to access the routine-state information in the **MI_FPARAM** structure, see "Accessing MI_FPARAM Routine-State Information" on page 9-2.

Event Handling

Your C UDR must perform event handling to ensure recovery from unexpected results, usually a warning or runtime error from the database server. To handle warnings and errors, the C UDR can define callback functions that the DataBlade API invokes when a particular event occurs. A C UDR can receive the following events.

Event	Description	Event Type
Database server exception	Raised when the database server generates an exception (a warning or an error)	MI_Exception
End of statement	Raised when the database server completes the execution of the current SQL statement	MI_EVENT_END_STMT
End a transaction (commit or rollback)	Raised when the database server reaches the end of the current transaction, whether the transaction contains one or many SQL statements	MI_EVENT_END_XACT
End of session	Raised when the database server reaches the end of the current session	MI_EVENT_END_SESSION

The UDR can register callback functions for any of these events. For more information about how to handle database server exceptions, see "Database Server Exceptions" on page 10-20. For information on how to handle state-transition events (such as end of statement, end of transaction, and end of session), see "State-Transition Events" on page 10-49.

Well-Behaved Routines

The most efficient way for a C UDR to execute is in the CPU virtual-process (CPU VP) class. However, to execute in the CPU VP, the UDR must be well-behaved. A well-behaved UDR adheres to a set of safe-code requirements that prevent the UDR from interfering with the efficient operation of the CPU VP. Table 13-1 on page 13-18 summarizes the safe-coding guidelines for a well-behaved UDR. You must ensure that your C UDR follows these guidelines for it to safely execute in the CPU VP class. Otherwise, the UDR must execute in a user-defined VP class. For more information, see "Using Virtual Processors" on page 13-16.

Compiling a C UDR

To compile a C UDR, use a C compiler to compile the source file (**.c** file extension) into an object file (**.o** file extension) and create a shared-object file that contains the object file.

Tip: The IBM Informix BladeSmith development tool, which is part of the Informix DataBlade Developers Kit, automatically generates makefiles to compile the DataBlade module code that it generates. It creates a makefile with a **.mak** extension for compilation on UNIX or Linux or with a **.dsw** extension for compilation on Windows. A makefile automates compilation of C UDRs. To compile a C UDR into a shared-object file (with a **.bld** extension), you only

have to run the appropriate makefile. For more information, see the *IBM Informix DataBlade Developers Kit User's Guide*.

Compiling Options

Use the C compiler to compile a C UDR. Include the following compiler options:

- Specify the necessary paths for any header files that the file needs, such as an **mi.h** header file, which includes the declarations of the DataBlade API data type structures and functions.

  These paths include the following subdirectories of the main Informix installation directory (which the **INFORMIXDIR** environment variable specifies):

  - The **incl/public** subdirectory contains public header files, such as **mi.h**.
  - The **incl/esql** subdirectory contains IBM Informix ESQL/C header files, such as **decimal.h**.

- Indicate that the DataBlade API module is a C UDR with the following compiler flag:

  MI_SERVBUILD

UNIX/Linux Only

On UNIX or Linux, the following sample command compiles the C UDR in the **abs.c** source file:

```
/compilers/bin/cc -I $INFORMIXDIR/incl
   -I $INFORMIXDIR/incl/esql -c abs.c

cc -KPIC -DMI_SERVBUILD -I$INFORMIXDIR/incl/public \
-I$INFORMIXDIR/incl -L$INFORMIXDIR/esql/lib -c abs.c
```

End of UNIX/Linux Only

At runtime:

```
LB_LIBRARY_PATH=/opt/informix/lib:/opt/informix/lib/esql
```

Windows Only

The following command is a sample of how to compile a C UDR named **abs.c** for Windows:

```
cl /DNT_MI_SAPI /DMI_SERVBUILD
-Id:\msdev\include -Id:\informix\incl\public
-Id:\informix\incl -c abs.c
```

End of Windows Only

Creating a Shared-Object File

You create a shared-object file to hold the compiled UDRs. This file resides in a directory on the server computer. Each UDR must have a unique name within the shared-object file.

UNIX/Linux Only

On UNIX or Linux a shared-object file is often called a *shared library*. On Solaris systems, shared-object files have the **.so** file extension.

Windows Only

On Windows a shared-object file is called a *dynamic link library* (DLL). DLLs usually have the **.dll** file extension.

--- **End of Windows Only** ---

When the database server executes an SQL statement that contains a UDR, it loads in memory the shared-object file in which the UDR executable code resides. It determines which shared-object file to load from the **externalname** column of the row in the **sysprocedures** system catalog table that describes the UDR. The CREATE FUNCTION or CREATE PROCEDURE statement creates a row for a UDR in **sysprocedures** when it registers the UDR.

To create a shared-object file for a C UDR:

1. Create a shared-object file and put the UDR object (**.o** file) file into this shared-object file.

   You can put C functions for related UDRs into the same shared-object file. However, the name of each C function must be unique within the shared-object file.

2. Put the shared-object file in a directory on which the user **informix** has read permission and the shared-object owner has write permission.

   The shared-object file must *not* have permissions that allow any user other than user **informix** to have write permission.

3. Specify the path of the shared-object file in the EXTERNAL NAME clause of the CREATE FUNCTION (or CREATE PROCEDURE) statement when you register the C UDR.

   The shared-object file does *not* have to exist before you register its path with CREATE FUNCTION or CREATE PROCEDURE. However, at UDR runtime, the paths of the shared-object file and the registered UDR must match for the database server to locate the UDR.

Important: If a shared-object file has write permission set to `all`, the database server issues error -9793 and writes a message in the log file when someone tries to execute any UDR in the shared object.

Note: For more information, see "Executing a UDR" on page 12-18. For information on how to create a shared-object file, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

To create a shared-object file on UNIX or Linux:

1. Load the **abs.o** object file into the **abs.so** shared library, as the following example shows:

   ```
   /compilers/bin/cc -K abs.o -o abs.so

   ld -G abs.o -o abs.so
   ```

2. Put the shared library in a directory on which user **informix** has read permission and set the permissions to 755 or 775 so that only the owner can write to the shared libraries.

```
# ls -ld /usr/code
drwxr-xr-x 12 informix devel 2560 Feb 25 05:27 /usr/code
# chmod 775 /usr/code/abs.so
drwxrwxr-x 12 informix devel 2560 Feb 25 05:27
   /usr/code
```

To create a shared-object file on Windows®:

1. Load the **abs.o** object file into a DLL named **abs.dll,** as the following example shows:

   ```
   link /DLL /OUT:abs.dll /DEF:abs.def abs.obj d:\informix\lib\SAPI.LIB
   ```

   The preceding command uses the IBM Informix software installed on the **d:** drive in a directory named **informix**.

2. If you are using Visual Studio 2005, embed the manifest file in the DLL with this command:

   ```
   mt -manifest abs.dll.manifest -outputresource:abs.dll;2
   ```

   The manifest file **abs.dll.manifest** was generated by the link command in step 1. The **mt** command embeds this manifest file in **abs.dll**. For more information on manifests, see the Microsoft Web site.

3. Put the DLL in a directory on which user **informix** has read permission and set the READONLY attribute with the **attrib +r** command:

   ```
   attrib +r abs.dll
   ```

Registering a C UDR

The CREATE FUNCTION and CREATE PROCEDURE statements register user-defined functions and user-defined procedures, respectively, in the database. These functions store information about the UDR in the **sysprocedures** system catalog table.

Registration for a C UDR requires the following special clauses of the CREATE FUNCTION and CREATE PROCEDURE statements to help the database server identify the routine:

- The required EXTERNAL NAME clause specifies the path to the shared-object file that contains the compiled C code for the UDR.
- The required LANGUAGE clause specifies the language in which the body of the UDR is written.
- The optional WITH clause specifies the routine modifiers for the UDR.

For example, Figure 12-1 shows a CREATE FUNCTION statement that registers a user-defined function named **abs_eq( )** whose corresponding C function is in a shared-object file named **abs.so**.

Routine name   Routine parameter list

```
CREATE FUNCTION abs_eq(arg1 INTEGER, arg2 INTEGER)
    RETURNS BOOLEAN
    WITH (NOT VARIANT)                              Return value
    EXTERNAL NAME                                   (Functions only)
        '/usr/code/abs.so'
    LANGUAGE C;                      Routine modifiers
                                     (optional)
```

Language name   Path to shared-object file
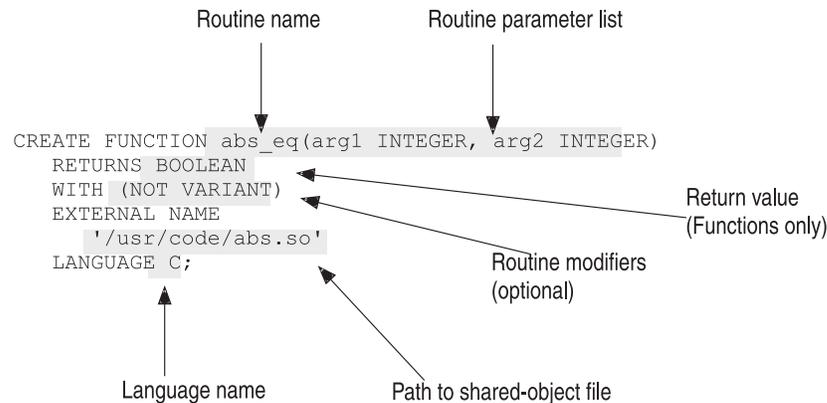
Figure 12-1. Registering a C UDR

Tip: The BladeSmith development tool, which is part of the Informix DataBlade
Developers Kit, automatically generates a file with the SQL statements needed
to register a DataBlade. The BladeManager development tool can install and
register a DataBlade module. For more information, see the *IBM Informix
DataBlade Developers Kit User's Guide*.

EXTEND Role Required to Register a C UDR

For security reasons, when the IFX_EXTEND_ROLE configuration parameter is set
to 1 or to 0n, only users who have been granted the EXTEND role by the database
server administrator (DBSA) can register external UDRs.

When a user creates or drops a procedure or a function, the database server checks
if the statement has the EXTERNAL clause, and then it checks if the user has been
granted the required permission. If the user does not have the EXTEND role, the
statement will fail. The default setting for the IFX_EXTEND_ROLE configuration
parameter is 0n. For more information about the IFX_EXTEND_ROLE
configuration parameter, see the *IBM Informix Dynamic Server Administrator's
Reference*.

The External Name

The EXTERNAL NAME clause of the CREATE FUNCTION or CREATE
PROCEDURE statement tells the database server where to find the object code for
the UDR. These statements store this location in the **externalname** column of the
sysprocedures system catalog table. When the database server executes an SQL
statement that contains a UDR, it loads into memory the shared-object file that
contains its executable code. The database server examines the **externalname**
column to determine which shared-object file to load.

In Figure 12-1, the EXTERNAL NAME clause of this CREATE FUNCTION
statement tells the database server that the object code for the **abs_eq( )**
user-defined function is in a Solaris shared-object file named **abs.so**, which resides
in the **/usr/code** directory.

By default, the database server uses the same name for the entry point into the
shared-object file for the UDR object code as the name of the UDR. For example,
the CREATE FUNCTION statement in Figure 12-1 on page 12-15 specifies that the
entry point in the **abs.so** shared-object file for the **abs_eq( )** user-defined function
is a C function named **abs_eq( )**.

The EXTERNAL NAME clause provides the following features to allow flexibility in the UDR external-name specification:

- Specifying a different entry point
- Including environment variables in the pathname

Each of these features is described in more detail below. For more information about the EXTERNAL NAME clause, see the External Reference segment of the *IBM Informix Guide to SQL: Syntax*.

Specifying the Entry Point

To specify an entry point whose name is *different* from the name of the UDR, put the name of the actual entry point in parentheses after the name of the shared-object file. You *must* specify an entry point when your UDR has a different name from the UDR that it implements. The following CREATE FUNCTION statement specifies that the entry point for the object code of the **abs_eq( )** UDR is a C function named **abs_equal( )**:

```
CREATE FUNCTION abs_eq(INTEGER, INTEGER)
   RETURNS boolean
   EXTERNAL NAME '/usr/code/abs.so(abs_equal)'
   LANGUAGE C;
```

The database server invokes the C function **abs_equal( )** whenever an SQL statement calls the **abs_eq( )** function with two arguments of INTEGER data type.

Using Environment Variables

You can include environment variables in the external-name specification of the EXTERNAL NAME clause. These environment variables must be set in the database server environment; that is, they must be set *before* the database server starts. For example, the following function registration specifies to evaluate the **USERFUNCDIR** environment variable when determining the location of the **my_func( )** user-defined function:

```
CREATE FUNCTION my_func(arg INTEGER)
RETURNING FLOAT
EXTERNAL NAME "$USERFUNCDIR/funcs.udr"
LANGUAGE C;
```

The UDR Language

The LANGUAGE clause of the CREATE FUNCTION or CREATE PROCEDURE statement tells the database server in which language the UDR is written. For C UDRs, the LANGUAGE clause must be as follows:

```
LANGUAGE C
```

The database server stores valid UDR languages in the **sysroutinelangs** system catalog table. These statements store the UDR language as an integer, called a language identifier, in the **langid** column of the **sysprocedures** system catalog table.

By default, only users with DBA privilege have the Usage privilege on the C language for UDRs. These users include user **informix** and the user who created the database. If you attempt to execute the CREATE FUNCTION or CREATE PROCEDURE statement with the LANGUAGE C clause as some other user, the database server generates an error.

To allow other users to register C UDRs in the database, a user with the DBA privilege can grant the Usage privilege on the C language with the GRANT statement. The following GRANT statement allows any user to register C UDRs:

```
GRANT USAGE ON LANGUAGE C TO public;
```

This statement stores the UDR-language privileges in the **syslangauth** system catalog table. By default, Usage privilege on C is only granted to the DBA. For more information on the syntax of the GRANT statement, see the *IBM Informix Guide to SQL: Syntax*.

Routine Modifiers

The routine modifiers tell the database server about attributes of the UDR. You specify routine modifiers in the WITH clause of the CREATE FUNCTION or CREATE PROCEDURE statement. The database server supports routine modifiers for C UDRs to perform the following tasks.

Type of UDR	Routine Modifier	More Information
Iterator function	ITERATOR	"Writing an Iterator Function" on page 15-3
Negator function	NEGATOR	"Creating Negator Functions" on page 15-60
Selectivity function	SELFUNC, SELCONST	"Writing Selectivity and Cost Functions" on page 15-54
Cost function	COSTFUNC, PERCALL_COST	"Writing Selectivity and Cost Functions" on page 15-54
Parallelizable UDR	PARALLELIZABLE	"Creating Parallelizable UDRs" on page 15-61
Recursive UDR	STACK	"Managing Stack Usage" on page 14-35
Ill-behaved UDR	CLASS	"Defining a User-Defined VP" on page 13-34
UDR that handles SQL NULL values as arguments	HANDLESNULLS	"Handling NULL Arguments" on page 13-8
UDR that is *not* valid within an SQL statement	INTERNAL	None

Parameters and Return Values

The CREATE FUNCTION and CREATE PROCEDURE statements specify any parameters and return value for a C UDR. For user-defined functions, the RETURN clause of the CREATE FUNCTION statement specifies the return value. Use SQL data types for parameters and the return value. These SQL data types must be compatible with the DataBlade API data types in the routine declaration. Table 1-1 on page 1-8 lists the SQL data types that correspond to the different DataBlade API data types.

For example, suppose you have a C UDR with the following C declaration:

```
mi_double_precision *func1(parm1, parm2)
   mi_integer parm1;
   mi_double_precision *parm2;
```

The following CREATE FUNCTION statement registers the **func1( )** user-defined function:

```
CREATE FUNCTION func1(INTEGER, FLOAT)
RETURNS FLOAT;
```

Use the opaque SQL data type, POINTER, to specify a data type for a C UDR whose parameter or return type has no SQL data type equivalent. For example, suppose you have a C UDR that has the following C declaration:

```
my_private_struc *func2(parm1, parm2)
   mi_integer parm1, parm2;
```

The following CREATE FUNCTION statement registers the **func2( )** user-defined function:

```
CREATE FUNCTION func2(INTEGER, INTEGER)
RETURNS POINTER;
```

This CREATE FUNCTION statement uses the POINTER data type because the data structure to which **func2( )** returns a pointer is a private data type, not one that is surfaced to users by registering it in the database.

Tip: If the C implementation of your UDR requires an **MI_FPARAM** structure in its declaration, omit this structure from the parameter list of the CREATE FUNCTION or CREATE PROCEDURE statement. For more information about when a C UDR requires an **MI_FPARAM** structure, see "MI_FPARAM Argument" on page 13-4.

For more information about how to declare a C UDR, see "Coding a C UDR" on page 13-2.

Privileges for the UDR

The CREATE FUNCTION and CREATE PROCEDURE statements assign the Execute privilege to the user who registers the UDR. Routine privileges for UDRs are stored in the **sysprocauth** system catalog table. By default, Execute privilege is granted to public. Whether you need to explicitly grant the Execute privilege for a UDR to other users depends on the whether or not the database is ANSI-compliant and on the setting of the **NODEFDAC** environment variable. For more information, see the description of the GRANT statement in the *IBM Informix Guide to SQL: Syntax*.

Executing a UDR

After you register a UDR as an external routine in the database, it can be called in one of the following ways:

- In a client application or SPL routine, through SQL statements:
  - In the select list of a SELECT statement
  - In the WHERE clause of a SELECT, UPDATE, or DELETE statement
  - In the VALUES clause of an INSERT statement
  - In the SET clause of an UPDATE statement
  - With the EXECUTE PROCEDURE or EXECUTE FUNCTION statement
- In a C UDR, as an SQL statement that one of the following DataBlade API statement-execution functions sends to the database server:
  - **mi_exec( )**
  - **mi_exec_prepared_statement( )**
  - **mi_open_prepared_statement( )**

  For more information on how to use statement-execution functions, see Chapter 8, "Executing SQL Statements," on page 8-1.
- Through an implicit UDR call

An implicit UDR is a UDR that the database server calls automatically in response to some SQL task. For example, in the following SELECT statement, the database server calls the **a_to_int( )** cast function when it executes the SELECT statement:

```
CREATE IMPLICIT CAST (a AS INTEGER WITH a_to_int);
...
SELECT a:int FROM tab1 WHERE b > 6;
```

- Through the Fastpath interface

  The Fastpath interface of the DataBlade API allows you to call a UDR directly from within another UDR. For more information, see "Calling UDRs with the Fastpath Interface" on page 9-14.

Tip: Within a C UDR, you can obtain the name of the SQL statement that invoked the UDR with the **mi_current_command_name( )** function.

Each occurrence of a UDR, implicit or explicit, in an SQL or SPL statement is a *routine instance*. One routine instance might involve several routine invocations. A *routine invocation* is one execution of the UDR. For example, if the following query selects five matching rows, the query has one routine instance of the **a_func( )** user-defined function and five routine invocations for this function:

```
SELECT a_func(x) FROM table1 WHERE y > 7;
```

Similarly an iterator function might contain many invocations in a single routine instance.

To execute a UDR instance in an SQL statement, the database server takes the following steps:

1. The query parser breaks the SQL statement into its syntactic parts and performs any *routine resolution* required.

   The query optimizer develops a query plan, which efficiently organizes the execution of the SQL-statement parts.

2. The query executer calls the *routine manager*, which handles execution of the UDR instance and any invocations.

The following sections provide information about how the steps of UDR execution can affect the way that you write the UDR. For more general information, see the chapter on how a UDR runs in the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Routine Resolution

If more than one registered UDR has the same routine name, the routine is overloaded. *Routine overloading* enables several routines to share a name and each of the routines to handle arguments of different data types. When an SQL statement includes a call to an overloaded routine, the query parser uses *routine resolution* to determine which of the overloaded routines best handles the data type of the arguments in the routine call of the SQL statement.

To perform routine resolution, the query parser looks up information in the system catalogs based on the *routine signature*. The routine signature contains the following information:

- The routine name
- The number and data types of the arguments
- Whether the routine is a function or a procedure

The database server combines this information to create an identifier that uniquely identifies the UDR. This routine identifier is in the **procid** column of the **sysprocedures** system catalog.

Tip: The DataBlade API provides the **mi_funcid** data type to hold routine identifiers. The **mi_funcid** data type has the same structure as the **mi_integer** data type. For backward compatibility, some DataBlade API functions (such as **mi_routine_id_get( )**) continue to store an **mi_integer** for a routine identifier.

For a detailed description of the steps involved in routine resolution, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

The Routine Manager

After the query parser has used routine resolution to determine which UDR to invoke, the query executor calls the routine manager to handle the UDR execution. The routine manager performs the following steps to execute the C UDR:

1. For each UDR instance:
   a. Load the shared-object file that contains the object code for the UDR into shared memory.
   b. Allocate and initialize the routine sequence for the UDR.
2. For each invocation of the UDR:
   a. Push the UDR argument values onto the thread stack.
   b. Dispatch the UDR to the appropriate virtual-processor class for execution.
   c. Save the return value of a user-defined function on the thread stack.
3. At the end of the UDR instance, release the routine sequence.

The following sections briefly describe each of these steps. For a general discussion of the routine manager, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Loading a Shared-Object File

When you compile a C UDR, you store its object code in a shared-object file. (For more information, see "Compiling a C UDR" on page 12-11.) For a UDR to execute, its object code must reside in memory so that a *virtual processor* (VP) can execute it. The database server uses virtual processors to service client-application SQL requests. A *thread* is a database server task that a VP schedules for processing.

Tip: For a detailed discussion of virtual processors and threads, see "Using Virtual Processors" on page 13-16.

When the routine manager reaches the first occurrence of the UDR in the SQL statement, the routine manager determines whether its shared-object file is currently loaded into the memory space of the appropriate VP class. If the file is *not* yet loaded, the routine manager dynamically loads its code and data sections into the data segment for *all* virtual processors of the VP class. The routine manager obtains the pathname of the shared-object file from the **externalname** column of the row in the **sysprocedures** system catalog for the UDR. This loading occurs for both explicit UDR calls and implicit calls (such as operator functions and opaque-type support functions).

Figure 12-2 shows a schematic representation of what VPs look like after the routine manager loads a shared-object file.
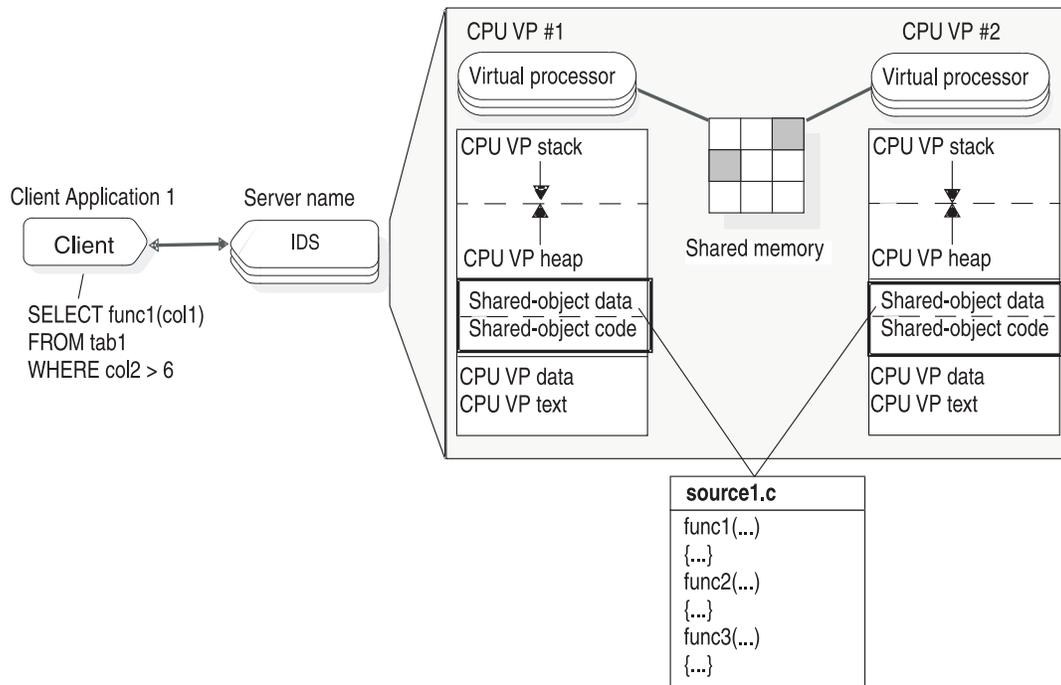
Figure 12-2. Loading a Shared-Object File

In Figure 12-2, assume that the **func1( )**, **func2( )** and **func3( )** functions are registered as user-defined functions with the CREATE FUNCTION statement and linked into the **source1.so** UNIX or Linux shared-object file. The client application calls the **func1( )** user-defined function within a SELECT statement. The routine manager loads the **source1.so** file into memory, if this file is not yet loaded. For subsequent references to these UDRs, the routine manager can skip the shared-object load.

The routine manager sends an entry to the message log file about the status of the shared-object load, as follows:

- When it successfully loads the shared-object file
- When it is not able to load the shared-object file for any of the following reasons:
  - The routine manager cannot find the shared-object file.
  - The shared-object file does not have read permission.
  - One of the symbols in the shared-object file cannot be resolved.
- When it unloads a shared-object file

For example, when the routine manager loads the **source1.so** shared-object file, the message log file would contain messages of the form:

```
12:28:45 Loading Module </usr/udrs/source1.so>
12:28:45 The C Language Module </usr/udrs/source1.so> loaded
```

Check the message log file for these messages to ensure that the correct shared-object file is loaded into the virtual processors.

You can monitor the loaded shared-object files with the **-g dll** option of **onstat**. This option lists the shared-object files that are currently loaded into the database server.

For information on when the shared-object file is unloaded, see "Unloading a Shared-Object File" on page 12-36. For information on how to create a shared-object file, see "Creating a Shared-Object File" on page 12-12. For general information about loading a shared-object file, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Creating the Routine Sequence

A *routine sequence* is the context in which the UDR executes. Generally, each routine instance (whether implicit or explicit) creates a single, independent routine sequence. For example, suppose you have the following query:

```
SELECT a_func(x) FROM table1 WHERE a_func(y) > 7;
```

When this query executes in serial, it contains two routine instances of **a_func( )**: one in the select list and the second in the WHERE clause. Therefore, this query has two routine sequences.

However, when a query with a parallelizable UDR (one that is registered with the PARALLELIZABLE routine modifier) executes in parallel, each routine instance might have more than one routine sequence. For more information, see "Executing the Parallelizable UDR" on page 15-64.

For each routine sequence, the routine manager creates a routine-state space, called an **MI_FPARAM** structure, that contains routine-state information from the routine sequence, including the following information:

- The routine identifier
- The number of arguments passed to the UDR
- Information about the UDR arguments
- The user state (optional)

The **MI_FPARAM** structure does *not* contain the actual argument values.

The routine manager allocates an **MI_FPARAM** structure when it initializes the routine sequence. This structure persists across *all* routine invocations in that routine sequence because the **MI_FPARAM** structure has a memory duration of PER_COMMAND. The routine manager passes an **MI_FPARAM** structure as the last argument to a UDR. (For more information, see "MI_FPARAM Argument" on page 13-4.) To obtain routine-state information, a C UDR invocation can access its **MI_FPARAM** structure. (For more information, see "Accessing MI_FPARAM Routine-State Information" on page 9-2.)

Pushing Arguments Onto the Stack

When the routine manager pushes arguments onto the thread stack, it pushes them as **MI_DATUM** values. The routine manager takes the following factors into account:

- Whether the argument is passed by value or by reference
- Whether the argument needs to be promoted

Passing Mechanism for MI_DATUM Values: The routine manager pushes **MI_DATUM** values onto the thread stack before it invokes the routine. The **MI_DATUM** structures contain the data in its internal database format. The size of the **MI_DATUM** data type determines whether the routine manager passes a particular argument by value or by reference, as follows:

- The routine manager passes most argument values *by reference*; that is, it passes a pointer to the actual argument value.

If the argument value has a data type whose size is greater than the size of the **MI_DATUM** data type, the routine manager passes the argument by reference because it cannot fit the actual value onto the stack. Instead, the **MI_DATUM** structure that the routine manager pushes onto the stack contains a pointer to the value. The routine manager allocates the memory for these pass-by-reference arguments with a PER_ROUTINE duration.

- The routine manager passes a few special types of argument *by value*; that is, the **MI_DATUM** structure contains the actual argument value.

  If the argument value is a data type whose size is less than or equal to the size of the **MI_DATUM** data type, the routine manager passes the argument by value because it can fit the actual value onto the stack.

Table 2-5 on page 2-33 lists the data types that the routine manager passes by value. All arguments whose data type is listed in this figure are passed by value *unless* the argument is an OUT parameter. OUT parameters are *never* passed by value; they are always passed by reference. The routine manager passes by reference any argument whose data type is *not* listed in Table 2-5 on page 2-33.

Tip: For a particular argument data type, you can determine from its type descriptor whether it is passed by reference or passed by value with the **mi_type_byvalue( )** function.

For information on how to code routine parameters, see "Defining Routine Parameters" on page 13-2. For information on how the routine manager passes return values out of a UDR, see "Returning the Value" on page 12-24.

Argument Promotion: C compilers that accept Kernighan-&-Ritchie (K&R) syntax promote all arguments to the **int** data type when they are passed to a routine. The size of this **int** data type is native for the computer architecture. ANSI C compilers permit arguments to be shorter than the native computer architecture size of an **int**. However, the routine manager uses K&R calling conventions when it pushes an **MI_DATUM** value onto the thread stack.

Tip: Many ANSI C compilers can use K&R calling conventions so code does work correctly across all platforms.

The routine manager cast promotes arguments with passed-by-value data types whose sizes are smaller than the size of the **MI_DATUM** data type to the size of **MI_DATUM**. When you obtain the smaller passed-by-value data type from the **MI_DATUM** structure, you should reverse the cast promotion to assure that your value is correct. For more information, see "MI_DATUM in a C UDR (Server)" on page 2-33.

Tip: To avoid this cast-promotion situation, the BladeSmith product generates C source code for BOOLEAN arguments as **mi_integer** instead of **mi_boolean**.

If you pass an argument smaller than an **MI_DATUM** structure, it is recommended that you pass a small "by-value" SQL type as an **mi_integer** value.

Managing UDR Execution

After the routine manager creates a routine sequence and pushes the arguments onto the stack, it invokes the UDR. It then manages the execution of the UDR associated with this routine sequence. The number of times that the UDR is invoked depends on the following factors:

- Does the UDR handle SQL NULL values?

If an argument to the UDR is the SQL NULL value and the UDR does *not* handle NULL values (it was not registered with the HANDLESNULLS routine modifier), the routine manager does *not* invoke the UDR.

- Is the UDR an iterator function?

An iterator function has several iterations. It executes once to initialize the iterations, once for each iteration, and once to release iteration resources. For more information, see "Writing an Iterator Function" on page 15-3.

- Where is the UDR invoked within the SQL statement?
  - If the UDR is in the select list, it executes once per row that the WHERE clause qualifies.
  - If the UDR is in the WHERE clause, the exact number of times that it executes cannot be predicted. It might be less than or equal to the number of rows or it might not be executed at all. The query optimizer makes this determination.
  - If the UDR is in an EXECUTE FUNCTION or EXECUTE PROCEDURE statement, it executes once (unless it is an iterator function).

A C UDR executes in one or more virtual processors (VPs). VPs are grouped by the kind of task they perform into VP classes. The presence of the CLASS routine modifier in the UDR registration determines in which VP class the UDR executes, as follows:

- If the UDR registration did *not* have a CLASS routine modifier or this CLASS routine modifier specified the CPU VP, the routine manager dispatches the UDR to a CPU VP for execution.
- If the UDR registration has a CLASS routine modifier that specifies a user-defined VP class, the routine manager dispatches the UDR to a VP in the specified VP class for execution.

For more information on how VPs execute C UDRs, see "Using Virtual Processors" on page 13-16.

Tip: The DataBlade API does provide some functions to change the VP environment once the UDR begins execution; however, these are advanced functions. You should use them only under special circumstances. For more information, see "Controlling the VP Environment" on page 13-38.

Returning the Value

For execution of a user-defined function, the routine manager returns any resulting value to the query executor when execution is complete. When the routine manager returns the value from a user-defined function, it passes this value as an **MI_DATUM** value. As with routine arguments, the passing mechanism that the routine manager uses depends on the size of the return-value data type, as follows:

- The routine manager passes most return values *by reference*; that is, it passes a pointer to the actual return value.

If the return value has a data type whose size is greater than the size of the **MI_DATUM** data type, the routine manager passes the return value by reference because it cannot fit the actual value onto the stack. The routine manager allocates the memory for these pass-by-reference return values with a PER_ROUTINE duration.

- The routine manager passes a few special types of return values *by value*; that is, the **MI_DATUM** structure contains the actual return value.

If the return value is a data type whose size is less than or equal to the size of the **MI_DATUM** data type, the routine manager passes the return value back by

value because it can fit the actual value onto the stack. Table 2-5 on page 2-33 lists the data types that the routine manager passes by value.

Tip: For a particular return-value data type, you can determine from its type descriptor whether it is passed by reference or passed by value with the **mi_type_byvalue( )** function.

The routine manager determines information about the return value (such as whether it is an SQL NULL value) from the **MI_FPARAM** structure of the UDR. For information on how to code routine return values, see "Defining a Return Value" on page 13-11.

Releasing the Routine Sequence

At the end of the routine instance, the routine manager releases the associated routine sequence. At this time, it also deallocates the **MI_FPARAM** structure.

Debugging a UDR

Because a UDR runs as part of the database server process, the routine must not do anything that might negatively affect the running of the database server, such as exiting. (For information on how to minimize the likelihood of interfering the database server, see "Creating a Well-Behaved Routine" on page 13-17.)

This section provides information on how to use a C debugger to attach to a virtual processor, handle the debugging session of a C UDR, and use DataBlade API tracing.

Using a Debugger

To debug your DataBlade module, use a debugger that can attach to the active server process and access the symbol tables of dynamically loaded shared object files.

UNIX/Linux Only

On UNIX or Linux, the **debugger** and **dbx** utilities meet these criteria. To start a debugger, enter the following command at the shell prompt, in which *pid* is the process identifier of the CPU or virtual processor:

```
debugger - pid
```

This command starts the debugger on the server virtual-processor process without starting a new instance of the virtual processor. For more information about available **debugger** commands, see the **debugger** manual page.

End of UNIX/Linux Only

To attach to the database server process:
1. Create a debugging version of the shared-object file.
2. Connect to the database server from a client application, such as DB–Access.
3. Make sure that the shared-object file is loaded into the server address space.
4. Obtain the process identifier for the virtual processor you want to debug.
5. Start the debugger on the server process.

The following sections describe these steps.

Creating a Debugging Version

To debug a shared-object file, you must compile the shared-object file with an option that makes additional symbol-table information available to the debugger. Many C compilers use the **-g** compiler option to create a debugging version of a shared-object file. For more information on how to compile, see "Compiling a C UDR" on page 12-11.

Connecting to the Database Server from a Client

To connect to the database server, choose a client tool that allows you to submit ad-hoc queries.

UNIX/Linux Only

On UNIX or Linux, you can use the DB-Access utility. For example, execute the following command, where *database* is a database in which you registered the shared-object file that you want to debug:

```
dbaccess database
```

End of UNIX/Linux Only

Windows Only

On Windows, you can use the SQL Editor.

End of Windows Only

Loading the Shared-Object File for Debugging

To load the shared-object file, you must execute one of the UDRs within the file. One technique is to execute the UDR itself within DB–Access. For example, if a user-defined function named **my_udf( )** resides within the shared-object file, you can use the following SQL statement to execute **my_udf( )**, which causes the database server to load the shared-object file that contains **my_udf( )**:

```
EXECUTE FUNCTION my_udf( );
```

Another technique for loading the shared-object file is to define a dummy UDR in the shared-object file that you use to load the shared-object file, as follows:

1. Create the dummy UDR in the shared-object file.

   The routine can be as simple as the following example:

   ```
   mi_integer load_so( )
   {
      return 0;
   }
   ```

   To prevent name conflicts with other shared-object files (or DataBlade modules), you can put a prefix in the routine name.

2. Compile the shared-object file.

3. Register the dummy UDR with the CREATE FUNCTION statement.

   ```
   CREATE FUNCTION load_so( )
   RETURNS INTEGER
   WITH (NOT VARIANT)
   EXTERNAL NAME '/usr/lib/udrs/myudrs.so(load_so)'
   LANGUAGE C;
   ```

To load the shared-object file, execute the dummy UDR. The following SELECT statement in your client application (or DB–Access) loads the **myudrs** shared-object file, which contains **load_so( )**:

```
SELECT load_so( ) FROM informix.systables WHERE tabid=1;
```

For more information about loading a shared-object file, see "Loading a Shared-Object File" on page 12-20.

Identifying the VP Process

To find the virtual processor in which your shared-object file is loaded, execute the **onstat** utility with the **-g glo** or **-g sch** option. Locate the CPU or user-defined virtual processor that you want to debug and record its process identifier (**pid**) for the next step. For more information on the **-g glo** and **-g sch** options of **onstat**, see "Monitoring Virtual Processors" on page 13-37.

Running a Debugging Session

You can set breakpoints, examine the stack, resume execution, or carry out any other normal debugger commands.

Breakpoints

You can set breakpoints in any function with an entry point known to your debugger. Valid functions include internal functions and the UDRs in your shared-object file. The database server is compiled with debugging support turned off, so local storage and line number information is not available for UDRs. However, because you compiled the shared-object file for debugging, you can see line number information and local storage for your functions.

The database server routine that calls functions in your shared-object file is named **udr_execute( )**. When you enter a command in the client application that calls one of your UDRs, the debugger stops in the **udr_execute( )** routine. You can then step through your UDR. Because your shared-object file is compiled with debugging support, you can view the local variables and stack for your functions.

──────────── UNIX/Linux Only ────────────

On UNIX or Linux, you can set a breakpoint in the **debugger** utility on the **udr_execute( )** function as follows:

```
stop in udr_execute
cont
```

──────────── End of UNIX/Linux Only ────────────

Debugging Hints

When you need to debug a UDR, keep the following considerations in mind:

- During the development of a UDR, to avoid interfering with the operation of the database server, develop UDRs on the client computer even if they are eventually intended to run from the database server process.
- Examine the database server log file for messages that the database server generates when it executes the UDR.

  The database server writes status messages to the message log file. By default, this file is named **online.log**. You can change the name of this file with the MSGPATH configuration parameter.
- Use the UDR tracing facility to insert trace messages within the body of the function.

The database server puts these trace messages in a trace file when the UDR executes. For more information, see "Using Tracing" on page 12-28.

- Simplify the UDR to create a good test case.

  Good test cases are as simple as possible.

- Install and register any required DataBlade modules

  If your execution environment includes DataBlade modules, you might want to first attempt debugging without these DataBlade modules so that the environment is as simple as possible. However, if this is not possible, make sure that you install and register any DataBlade modules that affect the execution of the UDR you are debugging. To install and register other DataBlade modules, such as the DataBlade modules included with Dynamic Server, see the instructions that accompany them.

Possible Memory Errors

Memory errors are usually caused by overrunning memory in a UDR. To avoid common causes of memory errors in a UDR, make sure you meet the following memory-handling requirements.

Memory-Handling Requirement	More Information
Do *not* return the NULL-valued pointer from a UDR.	"Returning a NULL Value" on page 13-13
Do *not* use null-terminated strings as data in a varying-length structure such as **mi_lvarchar**.	"Varying-Length Data and Null Termination" on page 2-17
Do *not* return local variables from a UDR.	"Returning a Value" on page 13-12
Make sure that you handle data types for parameters and return values with the correct passing mechanism.	"MI_DATUM Arguments" on page 13-3 and "Returning a Value" on page 13-12
Make sure memory that a UDR allocates is of the appropriate memory duration for its use. Do not access memory after its duration has expired.	"Choosing the Memory Duration" on page 14-4

Symbols in Shared-Object Files

The database server resolves undefined symbols in a shared-object file when it loads the shared-object file. If a symbol is missing, the load fails on the first execution of the UDR, and the database server writes a message in the log file. Symbols defined in two different shared-object files are distinct entities and do not resolve against each other.

A symbol defined in both a shared-object file and the Dynamic Server main module behaves in one of two ways:

- If the symbol referenced in the shared-object file is in the *same source file* that references it, the debugger accesses the symbol in the shared object file, as expected.
- If the shared-object file includes *more than one source file* and a cross-file symbol reference exists, the symbol is resolved to the main module of the database server.

Using Tracing

A *tracepoint* is a point within the code that can send special information about the current executing state of the UDR. Each tracepoint has the following parts:

- A *trace class* groups related tracepoints together so that they can be turned on and off at the same time.

- A *trace message* is the text that the database server sends to the tracing-output file.
- A *tracepoint threshold* determines when the tracepoint executes; if a tracepoint threshold is *not* greater than the current trace level, the DataBlade API writes the associated trace message to the trace output file.

Tip: The IBM Informix BladeSmith development tool, which is part of the Informix DataBlade Developers Kit, automatically includes tracing statements in the C source code that it generates. For more information, see the *IBM Informix DataBlade Developers Kit User's Guide*.

───────────────────────── **Client Only** ─────────────────────────

The DataBlade API tracing support is available only in C UDRs. Do *not* use this feature within client LIBMI applications.

───────────────────────── **End of Client Only** ─────────────────────────

The **mitrace.h** header file defines the functions and data type structures of the tracing interface. The **mi.h** header file automatically includes the **mitrace.h** header file. You must include either **mi.h** or **mitrace.h** in any C UDR that uses a DataBlade API tracing function.

The DataBlade API provides the following tracing support.

Time of Use	Tracing Support
At UDR-development time	Adds tracepoints in the C UDR with associated trace-level thresholds
At UDR runtime	Turns on different trace classes at specified trace levels

Adding a Tracepoint in Code

A *user-defined tracepoint* is a point within the code of a C UDR that can send special information about the current executing state of that routine.

To use a user-defined tracepoint:

1. Choose the trace class for the tracepoint.
2. Put trace messages into the UDR code.

   A tracepoint contains a trace message whose text you want to output. You assign to each tracepoint a trace class and a threshold level. If a tracepoint threshold is *not* greater than the current trace level, the DataBlade API writes the associated trace message to the trace output file.

3. Turn tracing on with an appropriate trace level for the tracepoints that you want to execute.

   You assign the current trace level when you turn on tracing with the **mi_tracelevel_set( )** function.

Choosing a Trace Class: Trace messages are grouped into *trace classes*. A trace class enables you to set up categories of related tracepoints, which you can then turn on or turn off independently. Within your UDR, you can choose either type of trace class for your tracepoint:

- The built-in **__myErrors__** trace class
- A user-defined trace class

Using the Built-In Trace Class: The DataBlade API provides a built-in trace class named **__myErrors__**. The **__myErrors__** trace class writes out the full text of database server exceptions (errors and warnings) as they occur. You can set the trace level of **__myErrors__** with **mi_tracelevel_set( )**, just as with any other trace class.

The **__myErrors__** trace class provides the following trace levels:
* A level of 10 or above to trace *only* error messages
* A level of 20 or above to trace *both* error and warning messages

Tip: The **__myErrors__** trace class does not appear in the **systraceclasses** system catalog table.

Creating a New Trace Class: To create your own trace class, define an entry for the trace class in the **systraceclasses** system catalog table. By default, all users can view this table, but only users with the DBA privilege can modify it. You can create as many trace classes as you like. The database server prevents you from creating a trace class name that is already in use.

Tip: The BladeSmith of the Informix DataBlade Developers Kit (DBDK) can add trace messages to the **systracemsgs** system catalog table. For more information, see your BladeSmith documentation.

Figure 12-3 shows the INSERT statement that creates a trace class named **funcEntry**.

```
INSERT INTO informix.systraceclasses(name)
   VALUES ('funcEntry');
```

Figure 12-3. Creating the funcEntry Trace Class

When you insert a new trace class into **systraceclasses**, the database server assigns it a unique identifier, called a *trace-class identifier*. It stores this trace-class identifier in the **classid** column of **systraceclasses**.

Tip: For more information on the columns of the **systraceclasses** system catalog table, see the *IBM Informix Guide to SQL: Reference*.

The built-in tracing that the DataBlade Developers Kit (DBDK) provides assumes that you create a single trace class and that its name is the same as the name of your DataBlade module. For more information, see the *IBM Informix DataBlade Developers Kit User's Guide*.

Putting Trace Messages into Code: The DataBlade API supports the following types of tracepoints in a C UDR:
* Tracepoints whose trace message is in U.S. English

───── **Global Language Support** ─────

* Internationalized tracepoints

    For more information on internationalized tracepoints, see the *IBM Informix GLS User's Guide*.

───── **End of Global Language Support** ─────

The DataBlade API provides the following tracing functions to insert U.S. English tracepoints into UDR code:
- The DPRINTF macro
- The trace-block functions, **tf( )** and **tprintf( )**

Tip: The BladeSmith of the Informix DataBlade Developers Kit (DBDK) can create tracing routines and macros as part of the code it generates for a DataBlade module. For more information, see your BladeSmith documentation.

Using DPRINTF Macro: The DPRINTF macro directly marks a tracepoint in the code. It formats a trace message and specifies the threshold for the tracepoint. The syntax for DPRINTF is as follows:

```
DPRINTF(trace_class, threshold, (format [, arg]...));
```

These syntax elements have the following values:

trace_class	is either a trace-class name or the trace-class identifier (an integer value) expressed as a character string.
threshold	is a non-negative integer that sets the tracepoint threshold for execution.
format	is a **printf**-style output format that formats the trace message and can include print formatting directives.
arg	is an expression to be evaluated for output. It provides the value for a print formatting directive in the *format* argument.

─── **Global Language Support** ───

The DataBlade API also provides the GL_DPRINTF macro for formatting internationalized trace messages. For more information, see the *IBM Informix GLS User's Guide*.

─── **End of Global Language Support** ───

The following example uses DPRINTF to insert a tracepoint of the **funcEntry** *trace_class* (which Figure 12-3 on page 12-30 defines) after the **doAbigPiece( )** function executes:

```
result = doAbigPiece(x, "x location");
DPRINTF("funcEntry", 50,
   ("After calling doAbigPiece with x = %d and %s \
      result = %f", x, "x location", result));
```

The trace message consists of the literal text, the print formatting directives, and the expressions for the print formatting directives.

To determine the value of a trace-class identifier, you can query the **systraceclasses** system catalog table for the **classid** column. The following SELECT statement obtains the trace-class identifier for the **funcEntry** trace class:

```
SELECT classid FROM informix.systraceclasses
WHERE name = 'funcEntry';
```

If the trace-class identifier for the **funcEntry** trace class is 42, the following DPRINTF call performs the same task as the preceding DPRINTF call:

```
result = doAbigPiece(x, "x location");
DPRINTF("42", 50,
   ("After calling doAbigPiece with x = %d and %s \
      result = %f", x, "x location", result));
```

The tracepoint *threshold* determines which tracepoints generate output, based on the current trace levels of the trace class. If a tracepoint threshold is *not* greater than the current trace level, the database server writes the associated trace message to the trace-output file.

The simplest tracing scheme is to have only two trace levels:
- Trace level = 0

  No tracing occurs.
- Trace level > 0

  Any tracepoint with a threshold greater than zero (0) prints. All others do not.

A more complex scheme could have four states: no tracing, light tracing, medium tracing, and heavy tracing. As an example, suppose you want to define the following trace levels for the **funcEntry** trace class.

Tracing Level	Threshold Values	Sample DPRINTF Call
None	0	None
Light	1 to 10	DPRINTF("funcEntry", 1, ("Entering doTheJob: the main function"));
Medium	11 to 20	DPRINTF("funcEntry", 11, ("Entering doAbigPiece: a top-level \help function"));
Heavy	>= 20	DPRINTF("funcEntry", 21, ("Entering doAlittlePiece: an often-called \helper"));

The maximum number of trace levels is the largest non-negative integer representable on the platform.

Trace Blocks: In some cases, you might need to perform some computations before you decide whether to output certain trace data. To perform computations for a tracepoint, define a *trace block*. A trace block initially compares a specified threshold with the current trace level to determine whether to continue with the trace computations. Within the trace block, it can output a result in a trace message.

The DataBlade API provides the following tracing functions for use in a trace block:
- The **tf( )** function acts as a threshold check that determines whether to execute a particular trace block.

  The syntax of the **tf( )** function is as follows:

  `tf(trace_class, threshold)`

  This function returns a Boolean result; it is TRUE if the current trace level of the *trace_class* class is greater than or equal to the *threshold*.
- The **tflev( )** function returns the current trace level of the specified trace class.

  The syntax of the **tflev( )** function is as follows:

  `tflev(trace_class)`

This function returns an integer result that is the current trace level of the *trace_class* class.

- The **tprintf( )** function outputs a trace message but does not require the threshold argument.

  The syntax of the **tprintf( )** function is as follows:

  ```
  tprintf(format [,arg]...)
  ```

  The *format* and *arg* arguments are the same as those in the DPRINTF macro.

The DataBlade API also provides the **gl_tprintf( )** function for formatting internationalized trace messages within a trace block. For more information on **gl_tprintf( )**, see the *IBM Informix GLS User's Guide*.

You can combine these trace-block functions with conventional C structures. The following example is typical:

```
/* Compare current trace level of "chck_consist" class and
 * with a tracepoint threshold of 20. Continue execution of
 * trace block if trace level >= 20
 */
if( tf("chck_consist", 20) )
   {
   x = fastScan(aList, y);
   consFactor = checkRconsit(x, bList);

   /* Generate trace message (in U.S. English) */
   tprintf("...in doWork: x = %f and consFactor = %f",
      x, consFactor);
   }
```

Defining Internationalized Trace Messages (GLS): The DataBlade API also supports internationalized trace messages, which are trace messages that correspond to a particular non-English locale. The current database locale determines which code set the trace message uses. Based on the current database locale, a given tracepoint can produce an internationalized trace message. Internationalized tracing enables you to develop and test the same code in many different locales. For more information on how to use internationalized trace messages, see the *IBM Informix GLS User's Guide*.

Using Tracing at Runtime

When your C UDR executes, you can use DataBlade API functions to perform the following tracing tasks.

Tracing Task	DataBlade API Function
Turn on tracing for one or more trace classes	**mi_tracelevel_set( )**
Set the trace-output file	**mi_tracefile_set( )**

After the trace-output file is created, you can examine its content for information about the runtime behavior of your UDR.

Turning Tracing On: The **mi_tracelevel_set( )** function sets the current trace level for one or more trace classes. You can use this function to perform the following tasks:

- Turn tracing for specified trace classes on or off.

By default, tracing for a particular trace class is off; that is, the current trace level of trace class is set to zero (0) for all trace classes. Any nonzero value for a trace level turns tracing on for the specified trace class.

• Change the current trace level for a trace class.

You can reset the trace level as often as necessary during testing. Once set, tracing persists throughout the session.

You pass to the **mi_tracelevel_set( )** function a series of *set commands*, one set command for each trace class that you want set. A set command has the following format:

traceclass_name trace_level

The following example sets the trace class **funcEntry** (which Figure 12-3 on page 12-30 defines) to a medium level (trace level of 14) and enables consistency checking with a trace class named **chk_consist**:

```
mi_integer ret;
...
ret = mi_tracelevel_set("chk_consist 1000 funcEntry 14");
```

You can also change the current trace level of a trace class with the **mi_tracelevel_set( )** function. The following example changes the trace level of the **funcEntry** trace class from 14 (from the previous example) to a lower level of 5:

```
ret = mi_tracelevel_set("funcEntry 5");
```

Specifying the Trace-Output File: By default, the database server puts all trace messages in a system-defined trace-output file with a **.trc** file extension. For the name of this system-defined trace-output file, see the description of **mi_tracefile_set( )** in the *IBM Informix DataBlade API Function Reference*.

You can change the destination of trace messages with the **mi_tracefile_set( )** function. With this function, you can specify the name of the trace-output file. For example, the following call to **mi_tracefile_set( )** sets the trace file to a UNIX file named **test14_may2.trc** in the **/d2/blades/tests** directory:

```
mi_integer status;
...
status =
   mi_tracefile_set("/d2/blades/tests/test14_may2.trc");
```

For more information, see the description of the **mi_tracefile_set( )** function.

Creating a UDR to Turn Tracing On: As a shortcut for debugging a UDR, you can create a UDR that automatically turns on tracing for your UDR. The registration of a sample user-defined procedure to perform such a task follows:

```
CREATE PROCEDURE trace_on(LVARCHAR, LVARCHAR)
EXTERNAL NAME '/usr/lib/udrs/my_tools.so(trace_on)'
LANGUAGE C;
```

The code for this user-defined procedure could be something like the following example:

```
void trace_on(trace_path, trace_level)
   mi_lvarchar *trace_path;
   mi_lvarchar *trace_level;
{
   mi_tracefile_set(mi_lvarchar_to_string(trace_path));
   mi_tracelevel_set(mi_lvarchar_to_string(trace_level));
};
```

After you register the trace-on procedure, you can turn on tracing for an SQL session with the following SQL statement:

```
EXECUTE PROCEDURE trace_on('trace_log_path',
   'my_trace_class 20');
```

In the preceding statement, *trace_log_path* is the path to your trace log.

Alternatively, you could create a user-defined procedure to turn on a particular trace class. The following CREATE PROCEDURE statement registers a user-defined procedure to turn on the MyBlade trace class:

```
CREATE PROCEDURE traceset_myblade(LVARCHAR, INTEGER)
EXTERNAL NAME '/usr/lib/udrs/myblade.bld(db_trace_on)'
LANGUAGE C;
```

The following code implements such a user-defined procedure:

```
void db_trace_on(trace_path, trace_level)
   mi_lvarchar *trace_path;
   mi_integer trace_level;
{
   char[16] trace_cmd;

   mi_tracefile_set(mi_lvarchar_to_string(trace_path));
   sprintf(trace_cmd, "%s %d", "MyBlade", trace_level);
   mi_tracelevel_set(trace_cmd);
}
```

Now the following SQL statement turns on tracing for trace class MyBlade with a trace level of 20 whose tracing output goes in the UNIX file **/u/dexter/udrs/myblade.trc**:

```
EXECUTE PROCEDURE trace_on('/u/dexter/udrs/myblade.trc', 20);
```

Understanding Tracing Output

The DataBlade API tracing functions prepend each trace message with a time stamp to show the time that the trace message is written to the trace-output file. The time stamp enables you to associate trace output with other information, such as entries in the database server log file.

Suppose you use the DPRINTF macro to create the following tracepoints:

```
mi_string *udr_name = "myUDR";
...
DPRINTF("funcEntry", 15, ("%s: entering UDR", udr_name));
x = 9;
result = doSomething(x);
DPRINTF("funcEntry", 15,
   ("%s: after calling doSomething\(%d\), result = %f",
      udr_name, x, result));
```

If you set a trace level of 15 or greater and run the UDR at 8:56 A.M., the tracepoints generate the following lines in the trace-output file:

```
08:56:03  myUDR: entering UDR
08:56:03  myUDR: after calling doSomething(9), result = value
```

In the previous trace output, *value* would be the value that the function call of **doSomething(9)** returned.

When trace messages in the source of the UDR appear in English and the UDR uses the default locale as its server-processing locale, messages appear in English in the trace-output file. If the code set of the trace-message characters in the UDR

source is different from (but compatible with) the code set of the server-processing locale, the database server performs the appropriate code-set conversion on these trace messages.

Global Language Support

To write an internationalized trace message to your trace-output file, the database server must locate a row in the **systracemsgs** system catalog table whose **locale** column matches (or is compatible with) the server-processing locale for your UDR. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Changing a UDR

This section provides information about how to alter a C UDR and how to unload a shared-object file.

Altering a Routine

The SQL statements ALTER FUNCTION, ALTER PROCEDURE, and ALTER ROUTINE allow you to alter some of the routine modifiers of a UDR after the UDR has been registered.

For information on how to alter a UDR, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide* and the syntax of the ALTER FUNCTION, ALTER PROCEDURE, and ALTER ROUTINE statements in the *IBM Informix Guide to SQL: Syntax*.

Unloading a Shared-Object File

In an attempt to keep memory usage to a minimum, the routine manager attempts to unload a shared-object file when it finds that no one is using any of its UDRs. That is, it unloads a shared-object file when *all* references to the UDRs within the shared-object file are removed from the internal cache of the database server and any one of them is marked as being dropped.

The database server cleans up its cache entries when you take any of the following actions:

- Explicitly drop a UDR

  When you use the DROP FUNCTION, DROP PROCEDURE, or DROP ROUTINE to drop all UDRs in a shared-object file, the routine manager unloads the shared-object file.

- Explicitly drop a database

  The routine manager unloads all shared-object files when DROP DATABASE executes.

- Create and reference UDRs in a transaction

For example, in the following SQL fragment, the shared-object file is loaded in and unloaded out of memory because the transaction has private cache entries until committed and the management mechanism treats them as being dropped:

```
BEGIN WORK;
CREATE FUNCTION c_func( ) ...;
CREATE FUNCTION spl_func( ) RETURNING c_func( )...;
COMMIT WORK;
```

To unload a shared-object file, you can take any of the following actions:

- For each routine that references the shared object (external file), execute the following SQL statement

```
ALTER ROUTINE routine-name (args, ...)
    WITH (
         MODIFY EXTERNAL NAME = 'shared-obj'
         );
```

  The new pathname for the shared object must be different from the existing one for the shared object to be unloaded. Instead of ROUTINE, you can specify FUNCTION for a function or PROCEDURE for a procedure.

  After the last routine is altered, nothing in the database server should refer to the old shared object, and a message appears in the online log to report that the shared object has been unloaded.

- Drop all UDRs in the shared-object file

  When you execute DROP FUNCTION, DROP PROCEDURE, or DROP ROUTINE to drop all UDRs in a shared-object file, the routine manager unloads the shared-object file. Similarly, the routine manager unloads all shared-object files when DROP DATABASE executes. Execution of one of these SQL statements is the safest way to unload a shared-object file.

- Execute the SQL procedure, **ifx_unload_module( )**, to request that an unused shared-object file be unloaded

  The **ifx_unload_module( )** function can *only* unload an unused shared-object file; that is when no executing SQL statements (in *any* database) are using any UDRs in the specified shared-object file. If any UDR in the shared-object file is currently in use, **ifx_unload_module( )** raises an error.

- Load a new module of a different name with the SQL function **ifx_replace_module( )**

  With the **ifx_replace_module( )** function, you do not have to change all the routine definitions. This action should eventually cause the old shared-object file to unload.

For the syntax of the **ifx_unload_module( )** and **ifx_replace_module( )** functions, see the *IBM Informix Guide to SQL: Syntax*.

Important: Do not use the **ifx_replace_module( )** function to reload a module of the same name. If the full names of the old and new modules that you send to **ifx_replace_module( )** are the same, unpredictable results can occur.

DataBlade modules can be shared across databases. Therefore, you might have more than one database using the same DataBlade module.

In Figure 12-2 on page 12-21, the routine manager has loaded the shared-object file named **source1.so**. This shared-object file contains definitions for the user-defined functions **func1( )**, **func2( )**, and **func3( )**.

The routine manager sends an entry in the message log file when it loads and unloads a shared-object file. When the routine manager unloads the **source1.so** shared-object file, the message-log file would contain messages of the form:

```
19:14:44  Unloading Module </usr/udrs/source1.so>
19:14:44  The C Language Module </usr/udrs/source1.so> unloaded
```

The message-log file is a useful place to check that the shared-object file is unloaded from the virtual processors. Alternatively, you can use the **onstat -g dll** command to monitor the results of an shared-object-file unload.

For information about when the shared-object file is loaded, see "Loading a Shared-Object File" on page 12-20. For information on how to prevent a shared-object file from being unloaded, see "Locking a Shared-Object File in Memory" on page 13-42.

Chapter 13. Writing a User-Defined Routine

In This Chapter

This chapter outlines some implementation issues for C user-defined routines (UDRs):

- Coding a C UDR
- Choosing the type of virtual processor in which to run your C UDR
- Controlling the virtual-processor environment
- Accessing operating-system files
- Accessing information in the server environment from within a C UDR

For information on how to manage memory within a C UDR, see Chapter 14, "Managing Memory," on page 14-1.

Client Only

This chapter covers topics specific to the creation of a C UDR. This material does not necessarily apply to the creation of client LIBMI applications. For information specific to the creation of client LIBMI applications, see Appendix A, "Writing a Client LIBMI Application," on page A-1.

End of Client Only

Coding a C UDR

When you code a C UDR, you perform the following tasks to write a C function:

- Define routine parameters
- Obtain argument values
- Define a return value
- Code the routine body

Defining Routine Parameters

When the routine manager invokes a C UDR, the routine manager passes to the UDR any argument values that the calling SQL statement provided. When you write a C UDR, you can define routine parameters that indicate the data types of the arguments that you expect the UDR to handle.

This section provides information about how to define routine parameters for the following UDR arguments:

- **MI_DATUM** argument
- **MI_FPARAM** argument, which the routine manager passes in to every UDR

Tip: Routine arguments are optional; however, if your UDR does not require arguments, you must still declare an **MI_FPARAM** parameter in the C-function declaration. For more information, see "MI_FPARAM Argument" on page 13-4.

The routine manager uses the parameter data types in routine resolution. Therefore, you can have multiple UDRs with the same name, provided that their parameter lists uniquely identify the UDRs. For more information, see "Routine Resolution" on page 12-19.

Routines with No Arguments

Routine parameters are optional. If your UDR does not need parameters, follow your C-compiler conventions for the syntax to use when declaring the C function.

MI_DATUM Arguments

When an SQL statement invokes a UDR, the statement can specify column values or expressions to pass to the UDR. The routine manager passes these argument values to a UDR as **MI_DATUM** values. The data type of each argument determines the passing mechanism that the routine manager uses for the argument value, as follows:

- Values of most data types cannot fit into an **MI_DATUM** structure. The routine manager passes these argument values *by reference*.
- Values of a few data types can fit into an **MI_DATUM** structure. The routine manager passes these argument values *by value*.

The passing mechanism that the routine manager uses for a particular argument determines how you must declare the corresponding parameter of the UDR. For more information about how the routine manager passes argument values to a C UDR, see "The MI_DATUM Data Type" on page 2-32 and "Pushing Arguments Onto the Stack" on page 12-22.

Pass-by-Reference Arguments: When an argument has a value that cannot fit into an **MI_DATUM** structure, the routine manager passes the argument by reference. For each of these pass-by-reference arguments, you declare a parameter that is a *pointer to* a value of the parameter data type, in the C-function declaration.

Figure 13-1 shows the **bigger_double( )** user-defined function, which compares two **mi_double_precision** values. Because the routine manager passes **mi_double_precision** values by reference, **bigger_double( )** declares the two parameters as pointers to values of the **mi_double_precision** data type.

```
mi_double_precision *bigger_double(left, right)
   mi_double_precision *left, *right;
{
   mi_double_precision *dpp;

   dpp = mi_alloc(sizeof(mi_double_precision));
   if ( *left > *right )
      {
      *dpp = *left;
      return(dpp);
      }
   else
      {
      *dpp = *right;
      return(dpp);
      }
}
```

Figure 13-1. Passing Arguments by Reference

Important: Memory that the routine manager allocates to pass an argument by reference has a PER_ROUTINE memory duration. Therefore, it is guaranteed to be valid only for the duration of the UDR execution. The database server automatically frees this memory when the UDR completes.

Any C-language code that calls **bigger_double( )** must pass the
mi_double_precision values by reference, as in the following sample call:

```
mi_double_precision double1, double2, *result;

double1 = 13497.931669;
double2 = 235521832.00484;
result = bigger_double(&double1, &double2);
```

Tip: For varying-length data, the routine manager does not pass a pointer to the
actual data itself. Instead, it stores the varying-length data inside a
varying-length structure. Therefore, your C UDR must declare parameters that
expect varying-length data as a pointer to the appropriate varying-length
structure. Varying-length data includes text arguments (see "Handling
Character Arguments" on page 13-6) and varying-length opaque data types
(see "Handling Varying-Length Opaque-Type Arguments" on page 13-10).

Values passed into a UDR are often also used in other places in the SQL statement.
If your UDR modifies a pass-by-reference value, successive routines in the SQL
statement might use the modified value. When your UDR is run within the context
of an SQL statement, a routine that runs before it can see (and possibly modify)
any pass-by-reference values.

Tip: Avoid the modification of a pass-by-reference argument within a C UDR. For
more information, see "Modifying Argument Values" on page 13-11.

Pass-by-Value Parameters: When an argument has a data type that can fit into an
MI_DATUM structure, the routine manager passes the argument by value.
Table 2-5 on page 2-33 lists data types for arguments that you can pass by value.
For these pass-by-value arguments, you declare a parameter as the *actual parameter
data type* in the C-function declaration.

Figure 13-2 shows the **bigger_int( )** UDR, which compares two **mi_integer** values.
Because the routine manager passes **mi_integer** values by value, the UDR declares
the two parameters with the **mi_integer** data type, not as pointers to **mi_integer**.

```
mi_integer bigger_int(left, right)
   mi_integer left, right;
{
   if ( left > right )
      return(left);
   else
      return(right);
}
```

Figure 13-2. Passing Arguments by Value

Any C-language code that calls **bigger_int( )** must also pass the **mi_integer** values
by value, as in the following sample call:

```
mi_integer int1, int2, result;
...
int1 = 6;
in2 = 8;
result = bigger_int(int1, int2);
```

MI_FPARAM Argument

The routine manager passes an **MI_FPARAM** structure into *every* UDR that it
executes. This structure contains routine-state information about the UDR, such as
information about arguments and return values. Because the routine manager

automatically passes an **MI_FPARAM** structure to a UDR, you do *not* need to explicitly declare this structure in most C-function declarations.

You should include an **MI_FPARAM** declaration in the C-function declaration in the following cases:

- You need to access routine-state information within the UDR.

  When you declare an **MI_FPARAM** parameter, this declaration must be the *last* parameter in the C declaration of your UDR. For more information about the DataBlade API functions that access the routine-state information from **MI_FPARAM**, see "Accessing MI_FPARAM Routine-State Information" on page 9-2.

- You declare a UDR that does not take *any* arguments.

  A C UDR *always* gets at least one argument: a pointer to the **MI_FPARAM** structure. When the parameter list of your SQL UDR is *empty*, you must still include a declaration for the **MI_FPARAM** structure, even if the UDR does not access routine-state information.

For example, the **bigger_double( )** user-defined function in Figure 13-1 on page 13-3 does not include a declaration for the **MI_FPARAM** structure because it does not need to access routine-state information and it has other parameters. However, suppose you register a user-defined function named **func_noargs( )** that does not require any arguments:

```
CREATE FUNCTION func_noargs( ) RETURNS INTEGER
   EXTERNAL NAME '/usr/lib/udrs/udrs.so' LANGUAGE C;
```

In the C UDR, you can declare the **func_noargs( )** function with a single parameter, a pointer to the **MI_FPARAM** structure:

```
mi_integer func_noargs(MI_FPARAM *fparam)
{
...
}
```

The declaration of the **MI_FPARAM** structure allows the routine manager to pass this structure into the UDR.

Tip: If your C UDR does not declare an **MI_FPARAM** structure but determines dynamically that it needs information in this structure, it can use the **mi_fparam_get_current( )** function to obtain a pointer to its **MI_FPARAM** structure. This function, however, is an advanced feature. Make sure you need it before you use it in a C UDR.

Obtaining Argument Values

To obtain the argument value with a C UDR, access the parameter that you have specified in the C declaration of the function. The parameter declaration indicates the appropriate passing mechanism for the UDR parameters. You can access the argument values through these declarations, as you would any other C-function parameter, as follows:

- For pass-by-reference parameters, access the argument value through its pointer. Do *not* modify this pointer within the body of the UDR.

  Most data types are passed by reference. The sample UDR **bigger_double( )**, in Figure 13-1 on page 13-3, shows how to access pass-by-reference arguments within a UDR.

- For pass-by-value parameters, you can access the argument value directly through its parameter variable.

For a list of data types that can be returned by value, see Table 2-5 on page 2-33. The sample UDR **bigger_int( )**, in Figure 13-2 on page 13-4, shows how to access pass-by-value arguments within a UDR.

Tip: You can obtain information about an argument, such as its type and length, from the **MI_FPARAM** structure. For more information, see "Checking Routine Arguments" on page 9-3.

Handling Character Arguments

When the routine manager receives text data for a C UDR, it puts this text data into an **mi_lvarchar** varying-length structure. It then passes a pointer to this **mi_lvarchar** structure as the **MI_DATUM** structure for the UDR argument. Therefore, a C UDR must have its text parameter declared as a pointer to an **mi_lvarchar** structure when the parameter accepts data from the following SQL character data types:

- **CHAR**
- **IDSSECURITYLABEL**
- **LVARCHAR**
- **NCHAR**
- **NVARCHAR**
- **VARCHAR**

Note: Use of the SQL TEXT data type in a C UDR is not supported.

Global Language Support

For more information on the NCHAR and NVARCHAR data types, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Important: These SQL data types cannot be represented as null-terminated strings. A C UDR never receives a null-terminated string as an argument. Do *not* code a C UDR to receive null-terminated strings as arguments. For more information on how to access **mi_lvarchar**, see "Varying-Length Data Type Structures" on page 2-13.

For example, suppose you want to define a user-defined function named **initial_cap( )** that accepts a VARCHAR string, ensures that the string begins with an uppercase letter, and ensures that the rest of the string consists of lowercase letters. This UDR would be useful in the following query to retrieve a customer's last name:

```
SELECT customer_num
FROM customer
WHERE initial_cap(lname) = "Sadler";
```

In the preceding query, use of the **initial_cap( )** function means that you do not have to ensure that the customer last names (in the **lname** column) were entered with an initial uppercase letter. The preceding query would locate the customer number for either *Sadler* or *sadler*.

The following CREATE FUNCTION statement registers the **initial_cap( )** function in the database:

```
CREATE FUNCTION initial_cap(str VARCHAR(50))
RETURNS VARCHAR(50)
EXTERNAL NAME '/usr/udrs/text/checkcaps.so'
LANGUAGE C;
```

The following declaration of **initial_cap( )** specifies an **mi_lvarchar** pointer as the parameter data type even though the function is registered to accept a VARCHAR column value:

```
/* Valid C UDR declaration for string parameter */
mi_lvarchar *initial_cap(str)
   mi_lvarchar *str;
```

The following declaration of **initial_cap( )** is *invalid* because it specifies an **mi_string** pointer as the parameter data type:

```
/* INVALID declaration for string parameter */
mi_string *initial_cap(string)
   mi_string *string;
```

The **initial_cap( )** function in the preceding declaration would *not* execute correctly because it interprets its argument as an **mi_string** value when the routine manager actually sends this argument as an **mi_lvarchar** value.

Figure 13-3 shows the implementation of the **initial_cap( )** function.

```
#include <mi.h>
#include <ctype.h>

mi_lvarchar *initial_cap(str)
   mi_lvarchar *str;
{
   char *var_ptr, one_char;
   mi_lvarchar *lvarch_out;
   mi_integer i, var_len;

   /* Create copy of input data */
   lvarch_out = mi_var_copy(str);

   /* Obtain data pointer for varying-length data */
   var_ptr = mi_get_vardata(lvarch_out);

   /* Obtain data length */
   var_len = mi_get_varlen(lvarch_out);

   /* Check string for proper letter case */
   for ( i=0; i < var_len; i++ )
      {
      one_char = var_ptr[i];

      if ( i == 0 )
         /* Change lowercase first letter to uppercase */
         {
         if ( islower(one_char) ) /* is lowercase */
            var_ptr[i] = toupper(one_char);
         }
      else
         /* Change uppercase other letters to lowercase */
         if ( isupper(one_char) ) /* is uppercase */
            var_ptr[i] = tolower(one_char);
      }

   return (lvarch_out);
}
```

Figure 13-3. Handling Character Data in a UDR

Tip: A C UDR that returns data for one of the SQL character data types must return a pointer to an **mi_lvarchar**. The **initial_cap( )** function returns a varying-length structure to hold the initial-capital string. For more information, see "Returning Character Values" on page 13-13.

Handling NULL Arguments

By default, a C UDR does *not* handle SQL NULL values. When you call a UDR with an SQL NULL as the argument, the routine manager does not invoke the UDR. It returns a value of SQL NULL for the UDR. To have the UDR invoked when it is called with SQL NULL arguments, register the UDR with the HANDLESNULLS routine modifier and code the UDR to take special steps when it receives a NULL argument.

To determine whether an argument is SQL NULL, declare the **MI_FPARAM** structure as the last argument in the UDR and use the **mi_fp_argisnull( )** function to check for NULL argument values. Do *not* just compare the argument with a NULL-valued pointer. For more information, see "Handling NULL Arguments with MI_FPARAM" on page 9-5.

Handling Opaque-Type Arguments

When the routine manager receives opaque-type data for a C UDR, the way the routine manager passes this data to the UDR depends on the kind of opaque data type, as follows:

- For fixed-length opaque types, the routine manager usually passes a pointer to the internal format of the opaque type.
- For varying-length opaque types, the routine manager passes a pointer to an **mi_bitvarying** varying-length structure.

A C UDR must declare its opaque-type parameter appropriately.

Handling Fixed-Length Opaque-Type Arguments: For UDR arguments that are fixed-length opaque types, the routine manager passes a pointer to the internal format of the opaque type to the C UDR. If the fixed-length opaque type is defined as passed by value, however, the routine manager passes the actual internal format. For more information, see "Determining the Passing Mechanism for an Opaque Type" on page 16-7.

For example, suppose you want to define a user-defined function named **circle_area( )** that accepts a fixed-length opaque type named **circle** (which is defined in Figure 16-2 on page 16-3) and computes its area. The following CREATE FUNCTION statement registers the **circle_area( )** function in the database:

```
CREATE FUNCTION circle_area(arg1 circle)
RETURNS FLOAT
EXTERNAL NAME '/usr/udrs/circle/circle.so'
LANGUAGE C;
```

Because **circle** is a *fixed*-length data type that cannot fit into an **MI_DATUM** structure, the following declaration of **circle_area( )** specifies a pointer to the internal format of **circle**:

```
/* Valid C UDR declaration for fixed-length opaque-type
 * parameter
 */
mi_double_precision *circle_area(circle_ptr)
   circle_t *circle_ptr;
```

Figure 13-4 shows the implementation of the **circle_area( )** function.

```
#include <mi.h>
#include <ctype.h>
#include <circle.h>

mi_double_precision *circle_area(circle)
    circle_t *circle;
{
    mi_double_precision *area;

    /* Allocate memory for mi_double_precision return
     * value
     */
    area = mi_alloc(sizeof(mi_double_precision));

    /* Calculate circle area using radius from circle_t
     * structure and constant PI_CONSTANT (defined in
     * circle.h).
     */
    *area = (circle->radius * circle->radius) * PI_CONSTANT;

    return ( area )
}
```

Figure 13-4. Handling Fixed-Length Opaque-Type Data in a UDR

Tip: A C UDR that returns fixed-length opaque data must return a pointer to the internal format (unless the internal format can fit into an **MI_DATUM** structure and is declared to be passed by value). For more information, see "Returning Opaque-Type Values" on page 13-14.

Handling Varying-Length Opaque-Type Arguments: For UDR arguments that are varying-length opaque types, the routine manager puts the data into an **mi_bitvarying** varying-length structure. It then passes a pointer to this **mi_bitvarying** structure as the **MI_DATUM** structure for the UDR argument. Your UDR must extract the actual opaque-type data from the data portion of the **mi_bitvarying** varying-length structure. For more information on how to access varying-length structures, see "Using a Varying-Length Structure" on page 2-13.

Suppose that you want to create a user-defined function named **image_id( )** that accepts a varying-length opaque type named **image** (which is defined in Figure 16-3 on page 16-5) and returns its integer image identifier (**img_id**). The following CREATE FUNCTION statement registers the **image_id( )** function in the database:

```
CREATE FUNCTION image_id(arg1 image)
RETURNS INTEGER
EXTERNAL NAME '/usr/udrs/image/image.so'
LANGUAGE C;
```

Because **image** is a *varying*-length data type, the following declaration of **image_id( )** specifies an **mi_bitvarying** pointer as the parameter data type even though the function is registered to accept a value of type **image**:

```
/* Valid C UDR declaration for varying-length opaque-type
 * parameter
 */
mi_integer image_id(image)
    mi_bitvarying *image;
```

The following declaration of **image_id( )** is *invalid* because it specifies an **image** pointer as the parameter data type:

```
/* INVALID declaration for varying-length opaque-type
 * parameter
 */
mi_integer image_id(image)
   image_t *image;
```

The **image_id( )** function in the preceding declaration would *not* execute correctly because it interprets its argument as the internal structure for **image** (**image_t**) when the routine manager actually sends this argument as an **mi_bitvarying** value.

Figure 13-5 shows the implementation of the **image_id( )** function.

```
#include <mi.h>
#include <ctype.h>
#include <image.h>

mi_integer image_id(image)
   mi_bitvarying *image;
{
   image_t *image_ptr;

   /* Obtain pointer to image_t structure, contained
    * within the data portion of the mi_bitvarying
    * structure.
    */
   image_ptr = (image_t *)mi_get_vardata((mi_lvarchar *)image);

   return (image_ptr->img_id);
}
```

Figure 13-5. Handling Varying-Length Opaque-Type Data in a UDR

Tip: A C UDR that returns varying-length opaque-type data must return a pointer to an **mi_bitvarying** structure. For more information, see "Returning Opaque-Type Values" on page 13-14.

Modifying Argument Values

Do *not* modify a UDR argument unless it is an OUT parameter. The routine manager does *not* make routine-specific copies of the arguments that it passes to UDRs because it is more efficient not to do so. Keep in mind that values passed into a UDR are often used on other places in the SQL statement. If you modify a pass-by-reference value within the UDR, you also modify it for all other parts of the SQL statement (including other UDRs) that operate on the value after the UDR executes. When you modify a pass-by-reference argument within the UDR, you might create an order-dependent result of the SQL statement. That is, it now might make a difference when your UDR is run within the SQL statement.

Defining a Return Value

When you declare a C UDR, you specify the routine return value, as follows:

- For a user-defined function, the C declaration specifies the data type that the UDR returns.
- For a user-defined procedure, the C declaration specifies the **void** data type as a return value.

Important: A C user-defined function can only return one value.

Returning a Value

When a user-defined function completes, the routine manager returns its value as an **MI_DATUM** value. The data type of the return value determines the passing mechanism that the routine manager uses for the value, as follows:

- Most data types cannot fit into an **MI_DATUM** structure and are passed *by reference*.
- A few data types can fit into an **MI_DATUM** structure and are passed *by value* (see Table 2-5 on page 2-33).

The passing mechanism that the routine manager uses for a particular return value determines how you must declare it in the user-defined function, as follows.

Return-Value Data Type	Tasks to Return the Value
Data types that *cannot* fit into an **MI_DATUM** structure	Return the value *by reference*: • Declare a local variable that is a pointer to the actual return value • Allocate the memory for the return value with the PER_ROUTINE memory duration. Use a DataBlade API memory-management function. For more information, see "Managing User Memory" on page 14-20. • Assign the address of this memory to a local variable • Store the return value in this memory • Return the pointer to this memory as the return value
Data types that *can* fit into the **MI_DATUM** structure	Can return the value *by value*: • Declare a local variable to hold the actual return value • Store the return value in this local variable • Return the local variable as the return value

Important: A user-defined function cannot return an automatic or local variable if its data type cannot be returned by value. That is, any automatic or local variables with data types that cannot fit into an **MI_DATUM** structure cannot be returned by value from the UDR.

To return a value, use the automatic or local variable that you declared in the user-defined function, like you would any other C-function variable, as follows:

- For a pass-by-reference return value, use a pointer to allocated memory.

  Most data types are passed by reference. The sample UDR **bigger_double( )**, in Figure 13-1 on page 13-3, shows how to return an **mi_double_precision** value by reference. It allocates PER_ROUTINE memory for the return value, which the database server frees when the user-defined function completes.

- For a pass-by-value return value, you can return a variable directly as the value.

  For a list of data types that can be returned by value, see Table 2-5 on page 2-33. The sample UDR **bigger_int( )**, in Figure 13-2 on page 13-4, shows how to return an **mi_integer** value by value.

Tip: You can obtain information about a return value, such as its type or maximum length, from the **MI_FPARAM** structure. For more information, see "Accessing Return-Value Information" on page 9-6.

Returning a NULL Value: To return an SQL NULL value from a user-defined function, pass the **MI_FPARAM** structure as the last argument in the UDR and use the **mi_fp_setreturnisnull( )** function to set the NULL value in this **MI_FPARAM** structure. You must call the **mi_fp_setreturnisnull( )** function with MI_TRUE before your UDR completes. If you do not, you might receive an incorrect result from the UDR. Do *not* just return a NULL-valued pointer. For more information, see "Returning a NULL Value" on page 9-8.

Returning Character Values: The routine manager handles all character return values from a C UDR as **mi_lvarchar** values. Therefore, a C UDR must declare its return value as a pointer to an **mi_lvarchar** when it returns data for any of the following SQL character data types:

- **CHAR**
- **IDSSECURITYLABEL**
- **LVARCHAR**
- **NCHAR**
- **NVARCHAR**
- **VARCHAR**

Note: Use of the SQL TEXT data type in a C UDR is not supported.

Global Language Support

For more information on the NCHAR and NVARCHAR data types, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Important: SQL data types are not represented as null-terminated strings, so do *not* code a C UDR to return a null-terminated string. For more information on how to access an **mi_lvarchar** structure, see "Varying-Length Data Type Structures" on page 2-13.

For example, the **initial_cap( )** function in Figure 13-3 on page 13-8 can ensure that names are entered with an initial uppercase letter followed by lowercase letters. This UDR would be useful in the following query to ensure consistent capitalization of the customer last name:

```
INSERT INTO customer(customer_num, lname, fname)
VALUES (0, initial_cap("ANDERSON"), initial_cap("TASHI"));
```

The calls to **initial_cap( )** in this INSERT statement convert the last and first names of this customer as *Anderson* and *Tashi*, respectively.

Figure 13-3 on page 13-8 shows the following declaration for **initial_cap( )**:

```
/* Valid C UDR declaration for string return value */
mi_lvarchar *initial_cap(str)
   mi_lvarchar *str;
```

This declaration correctly specifies an **mi_lvarchar** pointer as the return type so that the function can return the VARCHAR value. The following declaration of **initial_cap( )** is *invalid* because it specifies an **mi_string** pointer as the return type:

```
/* INVALID declaration for string return value */
mi_string *initial_cap(string)
   mi_lvarchar *string;
```

The **initial_cap( )** function in the preceding declaration would *not* return the expected value because the routine manager interprets the **mi_string** that the UDR returns as an **mi_lvarchar**.

Tip: A C UDR that accepts data for one of these SQL character data types must also declare its parameters as **mi_lvarchar** pointers. For more information, see "Handling Character Arguments" on page 13-6.

Returning Opaque-Type Values: When the routine manager returns opaque-type data from a C UDR, the way it handles the return value depends on the kind of opaque data type, as follows:

- For fixed-length opaque types, the routine manager expects a pointer to the internal format of the opaque type, unless it was declared as pass by value.

  Therefore, a C UDR must declare its return value as a pointer to the internal format of the fixed-length opaque type. Only if the internal format can fit into an **MI_DATUM** structure can the C UDR pass the internal format by value.

- For varying-length opaque types, the routine manager expects a pointer to an **mi_bitvarying** varying-length structure.

  Therefore, a C UDR must declare its return value as a pointer to an **mi_bitvarying**. To return a varying-length opaque type, the UDR must put the varying-length structure into the data portion of the **mi_bitvarying** structure and return a pointer to this **mi_bitvarying** structure.

Tip: A C UDR that accepts opaque-type data must also declare its parameters based on whether the opaque type is fixed-length or varying-length. For more information, see "Handling Opaque-Type Arguments" on page 13-9.

Returning Multiple Values

Unlike an SPL routine, a C user-defined function can directly return *at most* one value. However, a user-defined function can return multiple values when you use the following features together:

- An OUT parameter in the user-defined function
- A statement local variable (SLV) in the SQL statement that calls the user-defined function

OUT parameters and SLVs enable a user-defined function to return a second value to the calling SQL statement.

Tip: This section discusses the use of SLVs and OUT parameters in the context of a C user-defined function. You cannot use SLVs and OUT parameters in SPL functions. A user-defined procedure with an OUT parameter must be called in the WHERE clause of an SQL statement. For general information on how to use an OUT parameter, see the discussion of how to return multiple values from external functions in the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

An alternative to using an OUT parameter is an iterator function. This special-purpose user-defined function can return multiple values, one value per iteration of the function. For more information, see "Writing an Iterator Function" on page 15-3.

Using an OUT Parameter: An *OUT parameter* is a routine argument that is always passed by reference to the C user-defined function. A C user-defined function can use an OUT parameter to return a value indirectly. For the OUT parameter, the database server allocates storage for an opaque data type or for a data type that

you could pass by value (but not for a varying-length data type) and passes a pointer to that storage to the UDR. Multiple OUT parameters are supported and they are allowed anywhere in the argument list, not just at the end.

For a C user-defined function to receive an OUT parameter, it must perform the following actions:

- Declare the OUT parameter as a pointer to the appropriate data type

  The size of the OUT parameter must be the size of an **MI_DATUM** structure. The passing mechanism for the parameter *must* be pass by reference regardless of the data type that you pass back.

- Set the argument-value array of the **MI_FPARAM** structure to NULL.

  The UDR should update the argument-value array.

DataBlade API modules often use OUT parameters for Boolean functions to return rank or scoring information (which can indicate how closely the return result matched the query criteria). For example, Figure 13-6 shows a C UDR, named **out_test( )**, that does not actually search a particular title for a string, but returns a 100 percent relative weight of match success as an OUT parameter.

```
mi_integer out_test(
   mi_lvarchar *doc,
   mi_lvarchar *query,
   mi_integer *weight,     /* OUT parameter */
   MI_FPARAM *fp)
{
   /* Set the value of the OUT parameter */
   *weight = 100;

   /* Set the value of the OUT parameter to "not null" */
   mi_fp_setargisnull(fp, 2, MI_FALSE);

   return MI_TRUE;
}
```

Figure 13-6. The out_test( ) User-Defined Function

In Figure 13-6, the call to the **mi_fp_setargisnull( )** function sets the third argument, which is the OUT parameter, to MI_FALSE, which indicates that the argument does not contain an SQL NULL value. The **MI_FPARAM** structure stores routine arguments in zero-based arrays. Therefore, the **mi_fp_setargisnull( )** function specifies a position of 2 to access the third argument.

Tip: For more information on how to access the **MI_FPARAM** structure, see "Accessing MI_FPARAM Routine-State Information" on page 9-2.

When you register the user-defined function, precede the OUT parameter with the OUT keyword. For example, the following CREATE FUNCTION statement registers the **out_test( )** function (which is defined in Figure 13-6):

```
CREATE FUNCTION out_test(doc LVARCHAR,
                  query VARCHAR(120),
                  OUT weight INTEGER)
   RETURNING BOOLEAN
   EXTERNAL NAME '/usr/udrs/udrs.so'
   LANGUAGE C;
```

Using the Statement-Local Variable: When you call a user-defined function that has an OUT parameter, you *must* declare a statement-local variable (SLV) in the

WHERE clause of the SQL statement. The SLV holds the value that the OUT parameter returns. Other parts of the SQL statement can then access the OUT parameter value through the SLV.

For example, the following SELECT statement calls the **out_test( )** function (which Figure 13-6 on page 13-15 defines) and saves the result of the OUT parameter in a statement-local variable named **weight**:

```
SELECT title, weight FROM mytab
   WHERE out_test(title, 'aaa', weight # INTEGER);
```

The SELECT statement specifies the statement-local variable in its select list so it can return the value of the OUT parameter.

For more information on the syntax and use of SLVs, see the description of how to return multiple values from a function in the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Coding the Routine Body

The actual work of the C UDR is done with C-language statements in the routine body. You can use the following statements and calls in the routine body:

- C-language statements
- Calls to functions in libraries that the DataBlade API supports

  For a description of these function libraries, see "Regular Public Functions" on page 1-14.
- Calls to other C functions

  You can call any of the following kinds of C functions:

  – Any other C UDR that is linked into the same shared-object file

    For more information, see "Calling UDRs Within a DataBlade API Module" on page 9-12.

  – A C UDR that does *not* reside in the same shared-object file

    For more information, see "Calling UDRs with the Fastpath Interface" on page 9-14.

You *cannot* directly call any other functions within a C UDR.

Using Virtual Processors

To service multiple client-application SQL requests, the database server uses *virtual processors* (VPs). The database server breaks the SQL request into distinct tasks, based on the resource that the task requires. Different VP types, called *virtual-processor classes* (VP classes), service the different kinds of tasks. The following table lists some of the types of VP classes that the database server supports.

Virtual-Processor Class	Description
System VP classes:	
CPU	Central processing (the primary VP class, which controls client-application requests)
AIO	Asynchronous disk I/O
SHM	Shared-memory network communications
User-defined VP class	Special VP class for additional types of processing

Tip: User-defined VPs are also referred to as Extension VPs, EXP VPs, EVPs, or Named VPs. This manual uses only the term "user-defined VP" to refer to a VP class that you define.

The database server preserves the state of each request in a *thread*. The database server assigns the thread to a VP class that manages the task or resource that the request requires. The VPs in the VP class service multiple requests for their resource by scheduling the threads on the resource.

The CPU virtual processor (CPU VP) is the main VP for the database server. The CPU VP acts as the central processor for client-application SQL requests. When a client application establishes a connection, the CPU VP creates the *session thread* for that client application. A CPU VP runs multiple session threads to service multiple SQL client applications.

Tip: This section describes VPs in the context of C UDRs. For a general description of VPs and UDRs, see the description of VPs in the *IBM Informix User-Defined Routines and Data Types Developer's Guide*. For a general description of VPs, see the chapter on database server architecture in your *IBM Informix Administrator's Guide*.

When an SQL request includes a C UDR, execution of this UDR becomes one of the tasks that the thread performs. Because a session thread is the primary thread for the processing of SQL requests, any C UDRs in an SQL request normally execute in the CPU VP. However, the tasks that your C UDR needs to perform might limit its ability to execute in the CPU VP, as follows:

- A well-behaved UDR *can* execute in the CPU VP.

  A well-behaved UDR adheres to a set of safe-code requirements that prevent the UDR from interfering with the efficient operation of the CPU VP.

- An ill-behaved UDR *cannot* execute in the CPU VP.

  If a C UDR does not follow all the safe-code requirements for a well-behaved routine, it must execute in a user-defined VP class. Some of the safe-code requirements can be relaxed for a C UDR that runs in a user-defined VP.

Important: The success of your C UDRs and your DataBlade API project depends in large degree on how well you implement the features related to the safety and interoperability of your C UDR.

Creating a Well-Behaved Routine

Because the CPU VP is used to execute all client requests, it is important that the code it executes be *well-behaved*; that is, all code should have the following attributes:

- Preserve *availability* of the CPU VP

  The CPU VP performs system services and related tasks and executes code for UDRs. If a UDR issues a standard blocking I/O call in a CPU VP, then the VP must wait for the I/O to complete and cannot attend to other threads and administrative tasks. The time spent waiting adversely affects the overall performance of the system. DataBlade API I/O functions enable the CPU VP to process the I/O asynchronously and do not block the CPU VP.

  The benefit of releasing the CPU VP so that it can execute other threads outweighs the overhead involved in saving the current thread state and switching to another thread. Each thread should explicitly yield the CPU VP in a timely manner (at least every 1/10 of a second).

- Be *process safe*

  Well-behaved code must be able to migrate among processes without loss of essential information or changing the global VP state. C UDR code is process safe when all state information is entirely encapsulated within the arguments to each C function and within the scope of the function itself. UDRs should not use global variables or system calls that change the process state.

  Code that is provided to execute within SQL statements (such as built-in SQL functions) is well-behaved. However, IBM does *not* have control over the code you write in your C UDR. A C UDR must be well-behaved to execute in the CPU VP. As a UDR developer, you must ensure that your C UDR adheres to the safe-code requirements in Table 13-1.

Table 13-1. Safe-Code Requirements for a Well-Behaved UDR

Safe-Code Requirement	Coding Rule	Possible Workarounds
Preserve availability.	Yield the CPU VP in a timely manner (at least every 1/10 of a second).	To execute in the CPU VP, use **mi_yield( )** to explicitly yield the CPU VP during resource-intensive processing.
		Otherwise, execute in a user-defined VP class.
	Do not use blocking I/O calls.	Execute in a yielding user-defined VP class.
	Never change the working directory.	None
Be process safe.	No heap-memory allocation	To execute in the CPU VP, use the DataBlade API memory-management functions.
	No modification of global or static data	To execute in the CPU VP, use the **MI_FPARAM** structure if you need to preserve state information. If necessary, global or static data can be read, as long as it is not updated.
		Otherwise, execute in a nonyielding user-defined VP class or a single-instance user-defined VP.
	No modification of the global state of the virtual processor	A C UDR that modifies the global VP state cannot execute safely in any VP.
		If modification of this data is essential to the application, execute the C UDR in a nonyielding user-defined VP class or a user-defined VP class that has only one VP defined.
Avoid unsafe operating-system calls.	Do not use any system calls that might impair availability or allocate local resources.	If use of such system calls is essential to the application, execute the C UDR in a nonyielding user-defined VP class and a single-instance VP and then change back.

If a UDR does not follow the safe-code requirements in Table 13-1, it is called an *ill-behaved* routine. An ill-behaved routine *cannot* safely execute in the CPU VP.

Warning: Execution of an ill-behaved routine in the CPU VP can cause serious interference with the operation of the database server. In addition, the UDR itself might not produce correct results.

If your C UDR has one of the ill-behaved traits in Table 13-1, follow the suggestions in the **Possible Workarounds** column. The following sections describe more fully the safe-code requirements for a well-behaved C UDR.

Preserving Availability of the CPU VP

A well-behaved C UDR must preserve the availability of the CPU virtual processor (CPU VP). The CPU virtual processor appears to execute multiple threads

simultaneously because it switches between threads. The database server tries to keep a thread running on the same CPU VP that begins the thread execution. However, if the current thread is waiting for some other type of resource to be accessed or some other task to be performed, the CPU virtual processor is needlessly held up. To avoid this situation, the database server can *migrate* the current thread to another VP.

For example, a query request starts as a session thread in the CPU VP. Suppose this query contains a C UDR that accesses a smart large object. While the thread waits for the smart-large-object data to be fetched from disk, the database server migrates the thread to an AIO VP, releasing control of the CPU VP so that other threads can execute.

At a given time, a VP can run only one thread. To maintain availability for session threads, the CPU VP swaps out one thread to allow another to execute. This process of swapping threads is sometimes called *thread yielding*. This continual thread yielding keeps the CPU VP available to process many threads. The speed at which CPU-VP processing occurs produces the appearance that the database server processes multiple tasks simultaneously.

Unlike an operating system, which assigns time slices to processes for their CPU access, the database server does not preempt a running thread when a fixed amount of time expires. Instead, it runs a thread until the thread *yields* the CPU VP. Thread yielding can occur at either of the following events:
- When the thread explicitly calls **mi_yield( )**
- When the thread requires some external resource to continue execution (such as file or data I/O)

When a thread yields, the VP switches to the next thread that is ready to run. The VP continues execution and migration of threads until it eventually returns to the original thread.

For a C UDR to preserve availability of the CPU VP, the UDR must ensure that it does not monopolize the CPU VP. When a C UDR keeps exclusive control of the CPU VP, the UDR *blocks* other threads from accessing this VP. A C UDR can impair concurrency of client requests if it behaves in either of the following ways:
- It does not regularly yield the CPU.

  You must ensure that the C UDR yields the CPU VP at appropriate intervals.
- It calls a blocking-I/O function.

  You must ensure that the C UDR does not call any blocking I/O functions because they can monopolize the CPU VP and possibly hang the database server.

Denying other threads access to the CPU VP can affect every user on the system, not just the users whose queries contain the same C UDR. If you cannot code a C UDR to explicitly yield during resource-intensive processing and to avoid blocking-I/O functions, the UDR is an ill-behaved routine and must execute in a user-defined VP class.

Yielding the CPU VP: To preserve the availability of the CPU VP, a well-behaved C UDR must ensure that it regularly yields the CPU VP to other threads. A C UDR might yield when it calls a DataBlade API function because DataBlade API functions automatically yield the VP when appropriate. For example, the UDR thread might migrate to the AIO VP to perform any of the following kinds of I/O:

- Smart-large-object I/O with a DataBlade API function such as **mi_lo_open( )**, **mi_lo_read( )**, or **mi_lo_write( )**
- External-file I/O with a DataBlade API file-access function such as **mi_file_open( )**, **mi_file_read( )**, or **mi_file_write( )**

Therefore, you can assume that thread migration might occur during execution of any DataBlade API function.

However, if your C UDR performs any of the following types of resource-intensive tasks (which do not involve calls to DataBlade API functions), your UDR does not automatically yield the VP:
- A task that is CPU- or I/O-bound
- A task that causes other threads to wait for an undue length of time (usually longer than 0.1 seconds)

For such a C UDR to be well-behaved, it must explicitly yield the CPU VP with the DataBlade API function **mi_yield( )**. The **mi_yield( )** function causes the thread that is executing the UDR to voluntarily yield the CPU VP so that other threads get a chance to execute in the VP. When the original thread is ready to continue execution, execution resumes at the point immediately after the call to the **mi_yield( )** function.

Write your C UDR so that it yields the VP at strategic points in its processing. Possible points include the beginning or end of lengthy loops and before and/or after expensive computations. Judicious use of **mi_yield( )** generally leads to an improved response time overall.

If you cannot code the C UDR to explicitly yield during resource-intensive passages of code, the UDR is considered an ill-behaved routine and must *not* execute in the CPU VP. To isolate a resource-intensive UDR from the CPU VP, you can assign the routine to a user-defined VP class. To determine which kind of user-defined VP to define, you must also consider whether you need to preserve availability of the user-defined VP. Keep in mind that all VPs of a class share a thread queue. If there are multiple users of your UDR, multiple threads can accumulate in the same thread queue. If your UDR does *not* yield, it blocks other UDRs that execute in the same VP class. Therefore, the VP might not effectively share between users. One user might have to wait while the UDR in the query of some other user completes.

You can use a user-defined VP to execute a resource-intensive routine:
- To preserve availability of a user-defined VP, execute the routine in a *yielding* user-defined VP.

  Within your UDR, you can use the **mi_yield( )** function to yield the user-defined VP to other threads that execute in the same VP class. To increase availability, you can define multiple instances of the yielding user-defined VP.
- If you cannot rewrite the routine to yield, add more user-defined VPs.

  A *nonyielding* user-defined VP is used for code that *must* maintain ownership of the process until it completes. A nonyielding VP might modify a global variable or use a command resource that cannot be shared.

Avoiding Blocking I/O Calls: To preserve concurrency, a well-behaved C UDR must avoid system calls that perform blocking input and output operations (I/O). Some of these operating-system calls follow:

accept( )	msgget( )	putmsg( )	semop( )
bind( )	open( )	read( )	wait( )
fopen( )	pause( )	select( )	write( )
getmsg( )	poll( )		

When a C UDR executes any of these system calls, the CPU VP must wait for the I/O to complete. In the meantime, the CPU VP cannot process any other requests. The database server can appear to stall because the concurrency of the CPU VP is impaired.

If your C UDR needs to perform file I/O, do *not* use operating-system calls to perform this task. Instead, use the DataBlade API file-access functions. These file-access functions allow the CPU VP to process the I/O asynchronously. Therefore, they do not block the CPU VP. For more information, see "Access to Operating-System Files" on page 13-52.

If your UDR must issue blocking I/O calls, assign the routine to execute in a user-defined VP class. When a UDR blocks a user-defined VP, only those UDRs that are assigned to that VP are affected. You might need to use a single instance of a user-defined VP, which would affect client response. Your UDR must also handle any problems that could occur if the thread yielded; for example, operating-system file descriptors do not migrate with a thread if it moves to a different VP.

Writing Threadsafe Code

A well-behaved C UDR must be threadsafe. During execution, an SQL request might travel around the different VP classes. For example, a query starts in the CPU VP, but it might migrate to a user-defined VP to execute a UDR that was registered for that VP class. In turn, the UDR might fetch a smart large object, which would cause the thread to migrate to the AIO VP.

Migrating a thread to a different VP means that the database server must preserve the state of the thread before it migrates the thread. When a client application connects to the database server, the database server creates a thread-control block (TCB) to store thread-state information needed when a thread switches VPs. The TCB includes the following thread-state information:

- Contents of the VP system registers
- Program counter, which contains the address of the next instruction to execute.
- Stack pointer, which points to private memory, called a *thread stack*

  For more information on use of the thread stack by a UDR, see "Managing Stack Space" on page 14-35.

Tip: For more information on the structure and use of the thread-control block, see your *IBM Informix Administrator's Guide*.

When a thread migrates from one VP to another, it releases its original VP so this VP can execute other threads. The benefit of releasing the CPU VP outweighs the overhead involved in saving the thread state. Therefore, a C UDR must be able to continue execution without loss of information when it migrates to a different VP.

For a C UDR to successfully migrate among VPs, its code must be *threadsafe*; that is, it must have the following attributes:

- Does *not* perform any dynamic memory allocation with operating-system calls
- Does *not* modify global or static data

- Does *not* modify other global process-state information

Tip: A parallelizable UDR has additional coding restrictions. For more information, see "Creating Parallelizable UDRs" on page 15-61.

Restricting Memory Allocation: To be threadsafe, a well-behaved C UDR must *not* use system memory-management routines to allocate memory dynamically including the following operating-system calls:

calloc( )	mmap( )	shmat( )
free( )	realloc( )	valloc( )
malloc( )		

Many other system calls allocate memory as well.

These operating-system calls allocate memory from the program heap space. The location of this heap space on only one VP creates the following problems:

- Heap memory available to one VP is not visible after a thread migrates to another VP.

  Once the thread migrates, the UDR can no longer access any data that was stored in heap memory. Even if the UDR allocates heap memory at the beginning of execution and frees this memory before it completes, the thread might still migrate to a different VP during execution of the UDR.

- Other VPs are not prevented from using the same address space for the shared-memory pool.

  When a VP needs to extend the virtual memory pool, it negotiates the addition of new shared-memory segments to the existing pool. The VP then updates the resident portion of shared memory and sends a signal to other VPs so that they can become aware of changes to shared memory.

  A VP that extends the memory pool is not aware of any portion of memory that **malloc( )** (or any other system memory-management routine) is using. Therefore, the VP might try to use the same address space that a system memory-management call has reserved.

- Heap memory that system memory-management calls allocate is not automatically freed.

  If a C UDR does not explicitly free this heap memory, memory leaks can occur.

For a C UDR to be well-behaved, it must handle dynamic memory allocation with the DataBlade API memory-management functions. These DataBlade API functions provide the following benefits:

- They allocate user memory from the database server shared memory.

  All VPs can access database server shared memory. Figure 14-2 on page 14-3 shows the areas of memory from which DataBlade API and operating-system memory-management functions allocate. For more information, see "Managing User Memory" on page 14-20.

- They allocate user memory with a specified lifetime called a *memory duration*.

  If a C UDR does not explicitly free memory that these DataBlade API functions allocate, the database server automatically deallocates it when its memory duration has expired. This automatic reclamation reduces memory leaks. For more information, see "Choosing the Memory Duration" on page 14-4.

Important: Do not call operating-system memory-management functions from within a C UDR. Use these DataBlade API memory-management

functions instead. The DataBlade API memory-management functions are safer in a C UDR than their operating-system equivalents.

If you are porting legacy code to a C UDR, you might want to write simple C programs to implement system memory-management calls and link these functions into your code before you make the UDR shared-object module. The following code fragment shows a simple implementation of **malloc( )** and **free( )** functions:

```
/* mallocfix.c: This file contains "fixed" versions of the
 *              malloc( ) and free( ) system memory-management
 *              calls for use in legacy code that currently
 *              uses malloc( ) and free( ).
 * Use mi_alloc( ) and mi_free( ) in new code.
 */
#include <mi.h>
void *malloc(size_t size)
{
   return (mi_alloc((mi_integer)size));
}

void free(void *ptr)
{
   mi_free(ptr);
}
```

This code fragment uses **mi_alloc( )**, which allocates user memory in the current memory duration. Therefore, the fragment allocates the memory with the default memory duration of PER_ROUTINE. For more information, see "Managing the Memory Duration" on page 14-21.

If you cannot avoid using system memory-management functions, your C UDR is ill-behaved. You can use system memory-management functions in your UDR *only* if you can guarantee that the thread will *not* migrate. A thread could migrate during *any* DataBlade API call. To guarantee that the thread *never* migrates, you can either allocate and free the memory inside a code block that does not execute any DataBlade API functions or use a single-instance VP.

This restriction means that if you must use a system memory-management function, you must segment the UDR into sections that use DataBlade API functions and sections that are *not* safe in the CPU VP. All files must be closed and memory deallocated before you leave the sections that are not safe in the CPU VP. For more information, see "External-Library Routines" on page 13-28.

Avoiding Modification of Global and Static Variables: To be threadsafe, a well-behaved C UDR must avoid use of global and static variables. Global and static variables are stored in the address space of a virtual processor, in the data segment of a shared-object file. These variables belong to the address space of the VP, not of the thread itself. Modification of or taking pointers to global or static variables is not safe across VP migration boundaries.

When an SQL statement contains a C UDR, the routine manager loads the shared-object file that contains the UDR object code into each VP. Therefore, each VP receives its own copy of the data and text segments of a shared-object file and all VPs have the same initial data in their shared-object data segments. Figure 13-7 shows a schematic representation of a virtual processor and indicates the location of global and static variables.
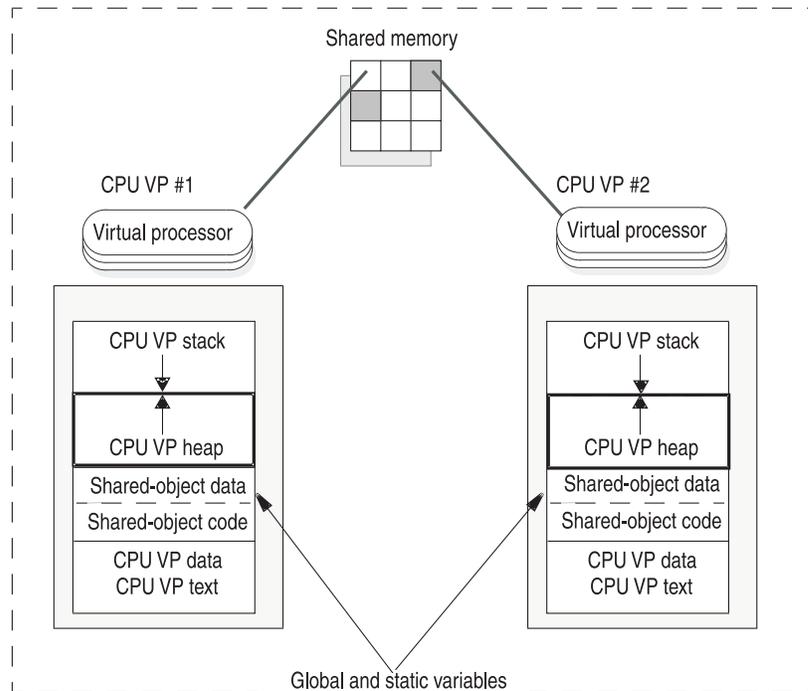
Figure 13-7. Location of Global and Static Variables in a VP

As Figure 13-7 shows, global and static variables are *not* stored in database server shared memory, but in the data and text segments of a VP. These segments in one VP are *not* visible after a thread migrates to another VP. Therefore, if a C UDR modifies global or static data in the data segment of one VP, the same data is not available if the thread migrates.

Figure 13-8 shows an implementation of a C UDR named **bad_rowcount( )** that creates an incremented row count for the results of a query.

```
/* bad_rowcount( )
 *      Increments a counter for each row in a query result.
 *      This is the WRONG WAY to implement the function
 *      because it updates a static variable.
 */
mi_integer
bad_rowcount(Gen_fparam)
   MI_FPARAM *Gen_fparam;
{
   static mi_integer bad_count = 0;
   bad_count++;
   return bad_count;
}
```

Figure 13-8. Incorrect Use of Static Variable in a C UDR

Suppose the following SELECT statement executes:

```
SELECT bad_rowcount( ), customer_id FROM customer;
```

The CPU VP that is processing this query (for example, CPU-VP 1) executes the **bad_rowcount( )** function. The **bad_rowcount( )** function is not well-behaved because it uses a static variable to hold the row count. Use of this static **bad_count** variable creates the following problems:

- The updated **bad_count** value is not visible when the thread migrates to another VP.

  When **bad_rowcount( )** increments the **bad_count** variable to 1, it updates the static variable in the shared-object data segment of CPU-VP 1. If the thread now migrates to a different CPU VP (for example, CPU-VP 2), this incremented value of **bad_count** is *not* available to the **bad_rowcount( )** function. This next invocation of **bad_rowcount( )** gets an initialized value of zero (0), instead of 1.

- Concurrent activity of the **bad_rowcount( )** function is not interleaved.

  For example, suppose CPU-VP 1 and CPU-VP 2 are processing session threads for three client applications, each of which execute the **bad_rowcount( )** function. Now two copies of the **bad_count** static variable are being incremented among the three client applications.

A well-behaved C UDR can avoid use of global and static data with the following workarounds.

Workaround	Description
Use only local (stack) variables and user memory (which the DataBlade API memory-management functions allocate).	Both of these types of memory remain accessible when a thread migrates to another VP:
	- Because the stack is maintained as part of the thread, reads and writes of local variables are maintained when the thread migrates among VPs. Write reentrant code that keeps variables on the stack.
	- User memory resides in database server shared memory and therefore is accessible by all VPs.
	For more information, see "Managing User Memory" on page 14-20.
Use a *function-parameter structure*, named **MI_FPARAM**, to track private state information for a C UDR.	The **MI_FPARAM** structure is available to all invocations of a UDR within a routine sequence. Figure 9-4 on page 9-11 shows the implementation of the **rowcount( )** function, which uses the **MI_FPARAM** structure to correctly implement the row counter that **bad_rowcount( )** attempts to implement. For more information, see "Saving a User State" on page 9-8.
If necessary, you can use *read-only* static or global variables because the values of these variables remain the same in each CPU VP.	Keep in mind, however, that addresses of global and static variables as well as addresses of functions are *not* stable when the UDR migrates across VPs.

If your C UDR cannot avoid using global or static variables, it is an ill-behaved routine. You can execute the ill-behaved routine in a nonyielding user-defined VP class but *not* in the CPU VP. A nonyielding user-defined VP prevents the UDR from yielding and thus from migrating to another VP. Because the nonyielding VP executes the UDR to completion, any global (or static) value is valid for the duration of a *single invocation* of the UDR. The nonyielding VP prevents other invocations of the same UDR from migrating into the VP and updating the global or static variables. However, it does not guarantee that the UDR will return to the same VP for the next invocation.

For the global (or static) value to be valid across a single UDR instance (*all invocations* of the UDR), define a single-instance user-defined VP. This VP class contains one nonyielding VP. It ensures that all instances of the same UDR execute on the same VP and update the same global variables. A single-instance user-defined VP is useful if your UDR must access a global or static variable by its address.

For more information, see "Choosing the User-Defined VP Class" on page 13-30.

Modifying the Global Process State: To be VP safe, a well-behaved C UDR must avoid modification of the global process state. All virtual processors that belong to the same VP class share access to both data and processing queues in memory. However, the global process state is *not* shared. The database server assumes that the global process state of each VP is the same. This consistency ensures that VPs can exchange work on threads.

For a C UDR to be well-behaved, it must avoid any programming tasks that modify the global process state of the virtual processor. Update of global and static data ("Avoiding Modification of Global and Static Variables" on page 13-23) involves modification of the global process. A well-behaved UDR must not use operating-system calls that can alter the process state, such as **chdir( )**, **fork( )**, **signal( )**, or **unmask( )**. Such operating-system calls can interfere with thread migration because the global process state does not migrate with the thread. In addition, you need to be careful with tasks such as opening file descriptors and using operating-system threads.

Avoiding Restricted System Calls

A well-behaved C UDR must avoid the use of restricted system calls, which can have the following adverse effects:

- They might block I/O, which causes the operating system to suspend the process that calls them.

  This suspension slows down both the C UDR that contains the calls and any other threads that share the same CPU virtual processor.

- Many system calls allocate resources local to the process and are not re-entrant.

IBM cannot provide a definitive list of unsafe system calls because system calls that are unsafe vary among versions of operating systems and different types of operating systems. Additionally, the implementation of the VPs is different between UNIX or Linux and Windows:

--- **UNIX Only** ---

- On UNIX or Linux, the VPs are implemented as separate processes.

--- **End of UNIX Only** ---

--- **Windows Only** ---

- On Windows, each VP is a thread of a common process.

--- **End of Windows Only** ---

The difference in VP implementation means that some system calls are acceptable when the C UDR runs on Windows but not when this same UDR runs on UNIX or Linux. There are also differences in how UNIX or Linux handles shared libraries

and how Windows handles dynamic link libraries (DLLs) that can affect the platform on which operating-system calls are valid. Therefore, UDRs might not be portable from one operating system to another.

Unsafe Operating-System Calls: An unsafe system call is one that blocks, causing the virtual processor to stall the CPU until the call returns, or one that allocates resources local to the virtual processor instead of in shared memory. A system call within a transaction is not terminated by a rollback, so a suspended transaction can wait indefinitely for the call to return. For instructions on recovery from a deadlock during a long transaction rollback, see the *IBM Informix Dynamic Server Administrator's Guide*.

A well-behaved C UDR must not include any of the categories of system calls in Table 13-2. The system calls listed in the Sample Operating-System Calls column are listed *only* as possible examples. The operating-system calls that are unsafe in your C UDR can depend on your operating system. Consult your operating-system documentation for information on system calls that perform the categories of tasks in Table 13-2.

Table 13-2. Unsafe Operating-System Calls

Type of Operating-System Call	Sample Operating-System Calls
Calls that manipulate signals to processes	**signal( ), alarm( ), sleep( )**
Calls that modify the system security	**setuid( ), seteuid( ), setruid( ), setgid( ), setegid( ), setrgid( )**
Calls that initiate or halt system processes	**fork( ), exec( ), exit( ), system( ), popen( )**
Calls that modify the shared-memory segments	**shmat( )**
Calls that modify the runtime environment of the dynamic linker	**dlopen( ), dlsym( ), dlerror( ), dlclose( )**Windows: **LoadLibrary( )**

Warning: The database server reserves all operating-system signals for its own use. The virtual processors use signals to communicate with one another. If a UDR were to use signals, these signals would conflict with those that the virtual processors use. Therefore, do *not* raise, handle, or mask signals within a C UDR.

You can use system utilities to check if undesired system calls were included in your shared-object file:

─── **UNIX Only** ───

- On UNIX or Linux, you can use the **nm** and **ldd** commands to obtain this information. The **ldd** command lists the dynamic dependencies from a shared object.

─── **End of UNIX Only** ───

─── **Windows Only** ───

- On Windows, you can use the **DUMPBIN** command with its **/IMPORTS** option to obtain this information.

─── **End of Windows Only** ───

Tip: Given a DataBlade build (**.bld**) file, check for unresolved references in the file and all its dependencies. You can compare this list for system calls that violate the rules of the VP you have chosen to execute your C UDR.

For a list of operating-system calls that are generally safe in a C UDR, see "Safe Operating-System Calls" on page 13-28.

External-Library Routines: It is recommended that a C UDR avoid the use of routines from existing external libraries. Some of these external routines might contain system calls that are restricted in your VP. If your C UDR must use an external routine, it might be *ill behaved*. Avoid calling the following kinds of external library routines, which are not safe in the CPU VP:

- Routines that do blocking I/O, such as routines that open files
- Routines that dynamically allocate memory, such as **malloc( )**
- Routines that allocate static memory

To execute one of these routines safely in a UDR, the following steps are possible:

1. Divide the UDR into critical-code sections and DataBlade-API-code sections.
2. Execute the UDR in a user-defined VP.

The following text explains these steps.

Important: Any external-library routine that uses signals cannot be used in a C UDR. Do not use this suggested workaround for any external library call that uses signals.

For an external routine to execute safely, the thread that executes the UDR must *not* migrate out of the VP as long as the UDR uses the unsafe resources (open files, memory allocated with **malloc( )**, or static-memory data). However, DataBlade API functions might automatically yield the VP when they execute. This yielding causes the thread to migrate to another VP.

Therefore, you cannot interleave DataBlade API calls and external routines in your UDR. Instead, you must segment your C UDR into the following distinct sections:

- Critical-code sections

  These sections contain *only* the external-library calls that are not safe in the CPU VP. Before execution leaves the critical-code section, any unsafe resources must be released: open files must be closed and memory allocated with **malloc( )** must be deallocated.

- DataBlade-API code sections

  These sections contain *only* DataBlade API functions. No external-library functions that are not safe in the CPU VP exist in these sections because any DataBlade API function might cause the thread to migrate.

Safe Operating-System Calls: The following table lists operating-system calls that are considered safe within a well-behaved C UDR on *all* supported platforms. Be sure to use threadsafe (**_r**) versions where applicable.

Category	System Calls	Notes
Character classification	**isalnum( )**, **isalpha( )**, **isascii( )**, **isastream( )**, **isatty( )**, **iscntrl( )**, **isdigit( )**, **isgraph( )**, **islower( )**, **isspace( )**, **isprint( )**, **ispunct( )**, **isupper( )**, **isxdigit( )**	None

Category	System Calls	Notes
String manipulation	**tolower( ), toupper( ), toascii( )**	None
String parsing	**getopt( ), getsubopt( )**	None
Multibyte strings	**mbtowc( ), wctomb( ), mblen( ), mbstowcs( ), wcstombs( )**	None.
String processing	**strcasecmp( ), strcat( ), strchr( ), strcmp( ), strcoll( ), strcpy( ), strcspn( ), strdup( ), strerror( ), strlen( ), strncasecmp( ), strncat( ), strncmp( ), strncpy( ), strpbrk( ), strrchr( ), strsignal( ), strspn( ), strstr( ), strtod( ), strtok( ), strtok_r( ), strtol( ), strtoll( ), strtoul( ), strtoull( ), strxfrm( )**	None.
String formatting	**sprintf( ), sscanf( )**	None
Numeric processing	**a641( ), l64a( ), abs( ), labs( ), llabs( ), atof( ), atoi( ), atol( ), atoll( ), div( ), ldiv( ), lldiv( ), lltostr( ), strtoll( )**	None
Random-number generation	**srand( ), rand( ), srandom( ), random( ), srand48( ), drand48( ), erand48( ), lrand48( ), nrand48( ), mrand48( )**	The random-number generator must be reseeded whenever a thread switch might have occurred.
Numeric conversion	**econvert( ), fconvert( ), gconvert( ), seconverty( ), sfconvert( ), sgconvert( ), qeconvert( ), qfconvert( ), ecvt( ), fcvt( ), gcvt( )**	**ifx_dececvt( ), ifx_decfcvt( )**
Time functions	**ascftime( ), strftime( ), cftime( ), ctime( ), ctime_r( ), asctime( ), asctime_r( ), gmtime( ), gmtime_r( ), difftime( ), localtime( ), localtime_r( )clock( ), gettimeofday( ), mktime( )**	No time-zone changes are permitted.
Date functions	**getdate( )**	None
Sorting and searching	**bsearch( ), qsort( ), lfind( ), lsearch( )**	None
Encryption	**crypt( ), setkey( ), encrypt( )**	None
Memory management	**memccpy( ), memchr( ), memcmp( ), memcpy( ), memmove( ), memset( )**	Use **memmove( )** and **memset( )** *only* for memory that was allocated with **mi_alloc( )**.
Environment information	**getenv( )**	None
Bit manipulation	**ffs( )**	None
Byte manipulation	**swab( )**	None
Structure-member manipulation	**offsetof( )**	None
Trigonometric functions	**acos( ), acosh( ), asin( ), asinh( ), atan( ), atan2( ), atanh( )cos( ), cosh( ), sin( ), sinh( ), tan( ), tanh( )**	None
Bessel functions	**j0( ), j1( ), jn( ), y0( ), y1( ), yn( )**	None
Root extraction	**cbrt( ), sqrt( )**	None

Category	System Calls	Notes
Rounding	**ceil( )**, **floor( )**, **rint( )**	None
IEEE functions	**copysign( )**, **isnan( )**, **fabs( )**, **fmod( )**, **nextafter( )**, **remainder( )**	None
Error functions	**erf( )**, **erfc( )**	None
Exponentials and logarithms	**exp( )**, **expm1( )**, **log( )**, **log10( )**, **log1p( )**, **pow( )**	None
Gamma functions	**lgamma( )**, **lgamma_r( )**	The contents of **signgam** are unreliable after a thread switch.
Euclidean distance	**hypot( )**	None

Tip: The system calls in the preceding table follow the Portable Operating System Interface for Computing Environments (POSIX) specification.

For a list of categories of operating-system calls that are generally unsafe in a UDR, see "Unsafe Operating-System Calls" on page 13-27.

Windows Only

The following actions are valid only in C UDRs that run on Windows and only if they do *not* interfere with the shared-memory model that the database server uses:
- C UDRs can create additional threads or processes.
- C UDRs can use shared memory for interprocess communication.

End of Windows Only

Important: Use of user-defined VPs can result in slightly lower performance because the thread must migrate from the CPU VP to the user-defined VP on which the C UDR executes. Use a user-defined VP only when necessary.

Choosing the User-Defined VP Class

When you run your C UDR in a user-defined VP, you can relax some, but not all, of the CPU VP safe-code requirements (Table 13-1 on page 13-18). You must choose a user-defined VP that is appropriate for the ill-behaved traits of your UDR. The following types of user-defined VPs allow a C UDR to contain the ill-behaved traits.

Type of User-Defined VP	Purpose
Yielding user-defined VP	Prevents a UDR from blocking the CPU VP because it blocks a user-defined VP thread
Nonyielding user-defined VP	Preserves global state of the VP across one UDR invocation
Single-instance user-defined VP	Preserves global state of the VP across all UDR invocations and instances

Warning: The user-defined VP class frees the CPU VPs from effects of some ill-behaved traits of a UDR. However, this VP class provides little protection from process failures. Even when the UDR runs in a

user-defined VP class, programming errors that cause process failures can severely affect the database server.

The Yielding User-Defined VP: By default, a user-defined virtual processor is a *yielding* VP. That is, it expects the thread to yield execution whenever the thread waits for other resources. Once a thread yields a user-defined VP, the VP can run other threads that execute UDRs assigned to this VP class. The most common use of a yielding user-defined VP class is for execution of code that cannot be rewritten to use the DataBlade API file-access functions to perform file-system activity.

The following table summarizes the programming requirements for C UDRs that apply to execution in a yielding user-defined VP.

CPU VP Safe-Code Requirement Rule	Required for Yielding User-Defined VP?
Yields the VP on a regular basis	Recommended
Does not use blocking operating-system calls	Not required
Does not allocate local resources, including heap memory	Yes
Does not modify global or static data	Yes
Does not modify other global process-state information	Yes
Does not use restricted operating-system calls	Yes

The main advantages of a yielding user-defined VP class are as follows:

- You can use the **mi_yield( )** function in your UDR to explicitly yield the user-defined VP.

  Failure to use **mi_yield( )** in a UDR creates the same loss of concurrency that it would in a CPU VP. However, loss of concurrency is not as critical in user-defined VPs because these VPs do not handle all query processing, as the CPU VPs do. For more information, see "Yielding the CPU VP" on page 13-19.

- You are no longer restricted from use of blocking I/O calls in the UDR.

  The C UDR can issue direct file-system calls that block further VP processing until the I/O is complete. Because user-defined VPs are not in the same VP class as CPU VPs, this blocking does not affect concurrency of the CPU VP or threads on other VPs. The most common use of a yielding user-defined VP is to run a UDR in which it is not practical to rewrite file-system activity with the DataBlade API file-access functions. For more information, see "Avoiding Blocking I/O Calls" on page 13-20.

Important: A yielding user-defined VP relaxes the restriction on use of blocking I/O calls. However, they do not remove the restrictions on other types of unsafe system calls. For more information, see "Avoiding Restricted System Calls" on page 13-26.

The main disadvantage of a yielding user-defined VP is that it can reduce performance of UDR execution. Execution in the CPU VP maximizes performance of a well-behaved UDR.

For more information, see "Defining a Yielding User-Defined VP Class" on page 13-35.

The Nonyielding User-Defined VP: A *nonyielding* user-defined virtual-processor class runs a C UDR in a way that gives the routine exclusive use of the VP. It executes the UDR serially. That is, each UDR runs to completion before the next

UDR begins. The C UDR does not yield. The most common use of a nonyielding user-defined VP class is for porting of legacy code that is not designed to handle concurrency issues (non-reentrant code) or that uses global memory.

The following table summarizes the programming rules that apply to execution in a nonyielding user-defined VP.

CPU VP Safe-Code Requirement	Required for Nonyielding User-Defined VP?
Yields the CPU on a regular basis	Not required
Does not use blocking operating-system calls	Not required
Does not allocate local resources, including heap memory	Yes
Does not modify global or static data	Not required (for global changes accessed by a *single invocation* of the UDR)
Does not modify other global data	Not required (for global changes accessed by a *single invocation* of the UDR)
Does not use unsafe operating-system calls	Yes

The main advantages of a nonyielding user-defined VP class is that a single invocation of the UDR is guaranteed to run on the same VP. This restriction creates the following benefits for an ill-behaved routine.

Feature of a Nonyielding User-Defined VP	Benefit to an Ill-Behaved UDR
Provides the same support for blocking I/O as a yielding user-defined VP	A UDR can perform blocking I/O functions. For a list of some sample blocking I/O functions, see "Avoiding Blocking I/O Calls" on page 13-20.
Can execute a C UDR that was not designed or coded to handle the concurrency issues of multiprocessing	A UDR executes to completion. A nonyielding user-defined VP ignores requests for a yield within DataBlade API functions as well as explicit calls to **mi_yield( )**.
Allows your UDR to modify global information	A UDR can modify global information (such as global or static variables, or global process information) as long as the changes to this global information are only needed within a single invocation of the UDR.

For more information, see "Avoiding Modification of Global and Static Variables" on page 13-23 and "Modifying the Global Process State" on page 13-26. |

However, a nonyielding user-defined VP has the following disadvantages:

- It reduces concurrency of the UDR execution.

  If you have multiple VPs in the nonyielding VP class, multiple instances of the UDR can run concurrently, one per VP. However, each UDR invocation runs to completion. No migration occurs while one UDR invocation executes (or if the UDR performs blocking I/O).

- It does not guarantee that the state remains across multiple instances of the UDR.

  Two invocations of the UDR might not overlap on the same VP. Therefore, the global VP state remains stable. However, another instance of the UDR might migrate into the VP and change the global VP state.

Important: If your UDR needs to make changes to global information that is available across the UDR instance, you must use a single-instance user-defined VP to execute the UDR.

For more information, see "Defining a Nonyielding User-Defined VP Class" on page 13-35.

The Single-Instance User-Defined VP: A *single-instance* user-defined VP class is a VP class that has only one VP. Therefore, it runs a C UDR in a way that gives the routine exclusive use of the entire VP class. As with a nonyielding user-defined VP, a single-instance VP executes a C UDR serially. Therefore, the UDR does not need to yield. Because a single-instance VP class has only one VP, the thread that executes the UDR does not migrate to another VP.

Depending on your requirements for yielding, a single-instance user-defined VP can be regular or nonyielding. A regular single-instance user-defined VP can handle the use of **malloc( )** and other local memory access. If it is nonyielding, the VP can deal with problems like modification of global variables.

CPU VP Safe-Code Requirement	Required for Single-Instance User-Defined VP?
Yields the CPU on a regular basis	Not required
Does not use blocking operating-system calls	Not required
Does not allocate local resources, including heap memory	Yes
Does not modify global or static data	Not required (for global changes accessed by a *single instance* of the UDR)
Does not modify other global process-state information	Not required (for global changes accessed by a *single instance* of the UDR)
Does not use restricted operating-system calls	Required for some calls

The main advantage of a single-instance user-defined VP class is that *all* instances of the UDR are guaranteed to run on the same VP (that is, on the same system process). Therefore, changes the UDR makes to the global information (global or static variables, or the global process state) are accessible across *all* instances of the UDR. A UDR might execute many times for a query, once for each row processed. With multiple VPs in a class, you cannot guarantee that all instances of a UDR execute on the same VP. Though execution for the first invocation might be on one VP, the execution for the next invocation might be on some other VP.

The only way to guarantee that all instances execute on one VP is to define a single-instance user-defined VP class. Therefore, a single-instance user-defined VP class is useful for a UDR that shares special information across multiple instances. Examples might be a special iterator function or a user-defined aggregate.

Tip: The DataBlade API supports the **mi_udr_lock( )** function to explicitly lock a UDR to a VP. For more information, see "Locking a Routine Instance to a VP" on page 13-41.

For example, suppose you have a UDR that contains the following code fragment:

```
{
    static stat_var;
    static file_desc;
    mi_integer num_bytes_read;
    ...
    file_desc = mi_file_open(....);
    num_bytes_read = mi_file_read(file_desc ....);
    ...
}
```

If this UDR ran on a yielding user-defined VP, the thread might yield at the
mi_file_read( ) call. Another thread might then execute this same code and change
the value of **file_desc**. When the original thread returned, it would no longer be
reading from the file it had opened. Instead, if you can assign this UDR to a
nonyielding user-defined VP, the thread never yields and the value of **file_desc**
cannot be changed by other threads.

The main disadvantage of a single-instance user-defined VP is that it removes
concurrency of UDR execution. This loss of concurrency brings the following
restrictions:

- A single-instance user-defined VP is probably not a scalable solution.

    All instances of the UDR that execute on a single-instance VP must compete for
    the same VP. You cannot increase the number of VPs in the single-instance class
    to improve performance.

- A single-instance user-defined VP does *not* support execution of parallel UDRs.

Important: If your UDR needs to make changes to global information that is
available across only a single invocation of the UDR, use a nonyielding
user-defined VP to execute the UDR. For more information, see "The
Nonyielding User-Defined VP" on page 13-31.

You must weigh these advantages and disadvantages carefully when choosing
whether to use a single-instance user-defined VP class to execute your ill-behaved
UDR. For more information, see "Defining a Single-Instance User-Defined VP
Class" on page 13-36.

Defining a User-Defined VP

You define a new virtual-processor class in the ONCONFIG file with the VPCLASS
configuration parameter. The **num** option specifies the number of virtual
processors in a user-defined VP class that the database server starts during its
initialization. The class name is not case sensitive, but it must have fewer than 128
characters. If your DataBlade uses a prefix, such as **USR**, begin the names of any
user-defined VPs with this prefix.

Dynamic Server supports the following types of user-defined VP classes for
execution of an ill-behaved C UDR.

Type of User-Defined VP Class	VPCLASS Option
Yielding user-defined VP	None (default type of user-defined VP class)
Nonyielding user-defined VP	**noyield**
Single-instance user-defined VP (yielding or nonyielding)	**num**=1

Important: When you edit the ONCONFIG file to create a new virtual-processor class, you must add a VPCLASS parameter and remove the SINGLE_CPU_VP parameter. For more information on the ONCONFIG file, see the *IBM Informix Administrator's Reference*.

After you add or modify the VPCLASS configuration parameter, restart the database server with the **oninit** utility (or its equivalent). For more information about how to restart the database server, see your *IBM Informix Administrator's Guide*. You can add or drop user-defined virtual processors while the database server is online. For more information, see "Adding and Dropping VPs" on page 13-37.

When you use a class of user-defined virtual processors to run a C UDR, you must ensure that the name of the VP is the same in both of the following locations:
- In the VPCLASS parameter in the ONCONFIG file, which defines the VP class
- In the CLASS routine modifier of the CREATE FUNCTION or CREATE PROCEDURE statement, which registers the C UDR in the database

For more information, see "Assigning a C UDR to a User-Defined VP Class" on page 13-36.

Defining a Yielding User-Defined VP Class: The VPCLASS configuration parameter creates a yielding user-defined VP by default. You can also use the **num** option to specify the number of VPs in the yielding user-defined VP class.

Figure 13-9 defines a yielding user-defined VP class named **newvp** with three virtual processors.

```
VPCLASS newvp,num=3                    # Yielding VP class with 3 instances
```

Figure 13-9. Defining a Yielding User-Defined VP Class

The C user-defined function, **GreaterThanEqual( )**, in Figure 13-12 on page 13-36, executes in the **newvp** VP class.

Defining a Nonyielding User-Defined VP Class: To create a nonyielding user-defined VP, include the **noyield** option of the VPCLASS configuration parameter. You can also use the **num** option to specify the number of VPs in the nonyielding user-defined VP class.

Tip: The **noyield** option is ignored for predefined virtual-processor classes such as CPU and AIO. For more information on the VPCLASS configuration parameter, see the *IBM Informix Administrator's Reference*.

Figure 13-10 defines the nonyielding user-defined VP class named **nonyield_vp** with two VPs in the class.

```
VPCLASS nonyield_vp, num=2, noyield               # Nonyielding VP class
```

Figure 13-10. Defining a Nonyielding User-Defined VP Class

At runtime you can determine whether the VP on which a UDR is running is part of a nonyielding user-defined VP class with the **mi_vpinfo_isnoyield( )** function. For more information, see "Obtaining VP-Environment Information" on page 13-39.

Defining a Single-Instance User-Defined VP Class: To define a single-instance user-defined VP, specify a value of one (1) for the **num** option of the VPCLASS configuration parameter. Figure 13-11 creates a yielding single-instance user-defined VP class, **single_vp**.

```
VPCLASS single_vp, num=1                    # Single-instance VP class
```

Figure 13-11. Defining a Single-Instance User-Defined VP Class

At runtime you can determine whether the VP on which a UDR is running is part of a single-instance user-defined VP class with the **mi_vpinfo_vpid( )** and **mi_class_numvp( )** functions. For more information, see "Obtaining VP-Environment Information" on page 13-39.

Assigning a C UDR to a User-Defined VP Class

When you register an ill-behaved C UDR, you assign it to a class of user-defined virtual processors with the CLASS routine modifier of the CREATE FUNCTION or CREATE PROCEDURE statement.

Tip: By default, all C UDRs execute in any VP. To have your C UDR run only in the CPU VP, you can specify the string "cpu vp" with the CLASS modifier. If your C UDR can run anywhere, you should omit the CLASS modifier.

For example, Figure 13-12 shows a CREATE FUNCTION statement that registers the C user-defined function, **GreaterThanEqual( )** and specifies that the user-defined VP class named **newvp** executes this function.

```
CREATE FUNCTION GreaterThanEqual(ScottishName, ScottishName)
    RETURNS BOOLEAN
    WITH (CLASS = 'newvp')
    EXTERNAL NAME '/usr/lib/objects/udrs.so(grtrthan_equal'
    LANGUAGE C;
```

Figure 13-12. Specifying a User-Defined VP Class for a C UDR

Figure 13-9 on page 13-35 shows the definition of the **newvp** user-defined VP class. All UDRs that specify the **newvp** VP class with the CLASS routine modifier share the three VPs in the **newvp** VP class.

When you register user-defined functions or user-defined procedures with the CREATE FUNCTION or CREATE PROCEDURE statement, you can reference any user-defined VP class that you like. The CREATE FUNCTION and CREATE PROCEDURE statements do not verify that the VP class you specify exists when they register the UDR.

Important: When you try to run a UDR that was registered to execute in a user-defined VP class, that VP class must exist and it must have virtual processors assigned to it. If the class does not have any virtual processors, you receive an SQL error. For information on how to define a user-defined VP, see "Defining a User-Defined VP" on page 13-34.

For more information on the syntax of CREATE FUNCTION or CREATE PROCEDURE to assign a C UDR to a VP class, see the description of the CLASS routine modifier in the Routine Modifier segment of the *IBM Informix Guide to SQL: Syntax*.

Managing Virtual Processors

To manage virtual processors, you need to perform the following tasks:

- Initialize VP classes
- Add and drop VPs
- Monitor VPs

Initializing a VP Class

Check your *IBM Informix Administrator's Guide* and the *IBM Informix Administrator's Reference* for information on VP-class initialization.

Adding and Dropping VPs

You can add or drop virtual processors in a user-defined VP class or in the CPU VP class while the database server is online. Use **onmode -p** to add a VP to a class or to drop a VP from a class.

The following command adds one virtual processor to the **newvp** class (which Figure 13-9 on page 13-35 defines):

```
onmode -p +1 newvp
```

To remove a virtual processor, specify a negative value in the **-p** option. For more information on the **onmode** utility, see the *IBM Informix Administrator's Reference*.

Monitoring Virtual Processors

You can use the following options on the **onstat** utility to monitor VPs:

- The **-g glo** option generates information about global multithreading such as CPU use of virtual processors and total number of sessions.
- The **-g rea** option generates information about the number of threads in the ready queue of the VP class.
- The **-g sch** option generates information about the number of semaphore operations, spins, and busy waits for each virtual processor.

A user-defined VP class appears in the **onstat -g glo** output as a new process. You can use the **-g glo** option to find the virtual process in which your DataBlade API module is loaded. Figure 13-13 shows the last section of the output of this **onstat** command.

```
Individual virtual processors:
vp    pid     class     usercpu   syscpu   total
1     11440   cpu       31.66     1.41     33.07
2     11441   adm       0.07      0.24     0.31
3     11442   jvp       0.04      0.05     0.09
4     11443   lio       0.25      1.57     1.82
5     11444   pio       0.03      0.25     0.28
6     11445   aio       0.37      1.77     2.14
7     11446   msc       0.00      0.04     0.04
8     11447   aio       0.25      1.47     1.72
9     11448   aio       0.08      0.68     0.76
10    11449   aio       0.19      0.68     0.87
11    11450   aio       0.15      0.46     0.61
12    11451   aio       0.06      0.35     0.41
13    11615   newvp     0.00      0.02     0.02
              tot       33.15     8.99     42.14
```

Figure 13-13. onstat -g glo Command Output

In Figure 13-13, the **onstat** utility displays CPU usage for the CPU VP as the first line of output. It displays the processor and CPU usage for the user-defined VP **newvp**, which Figure 13-9 on page 13-35 defines, as the thirteenth line of output. For more information on the **onstat** utility, see the *IBM Informix Administrator's Reference*.

In addition, you can select information from the **sysvpprof** SMI table about the virtual processors that are currently running. The **sysvpprof** SMI table exists only in the **sysmaster** database.

Controlling the VP Environment

The routine manager executes your C UDR in a *virtual-processor* (VP) *environment*. The VP environment consists of a VP and VP class, as follows:

- The current VP

  When a C UDR executes, it runs on a particular virtual processor called the *current VP*, which has an ID number from 1 to MAXVPS. A current VP is an *active VP*; that is, it is currently performing some task. The task that the active VP performs depends on the VP class to which it belongs. For example, a CPU VP can execute SQL statements and well-behaved UDRs. A user-defined VP executes those UDRs that are assigned to it (with the CLASS routine modifier of the CREATE FUNCTION or CREATE PROCEDURE statement).

- The VP class to which the current VP belongs

  The UDR specifies its VP class with the CLASS routine modifier when it is registered. If the CREATE FUNCTION or CREATE PROCEDURE statement omits the CLASS modifier, the UDR executes in the current active VP class.

The following traits of C UDRs are common reasons for needing to control the VP environment:

- The code uses advanced operating-system calls.

  For more information, see "Avoiding Restricted System Calls" on page 13-26.

- The code performs some other task that is ill-behaved.

  For more information, see "Preserving Availability of the CPU VP" on page 13-18 and "Writing Threadsafe Code" on page 13-21.

- The code is written in C++.

  All C++ code has the potential to not follow the memory management rules for well-behaved code. The most serious violation of these rules is the use of static virtual function pointers in C++ classes.

Warning: The ability of the database server to support some C++ features should not be taken as an open invitation to freely use C++ in your UDR code. Many C++ features implicitly violate the Safe-Coding Requirements for a well-behaved routine (see Table 13-1 on page 13-18). Problems can arise if some C++ features are used in a UDR.

If the source code is not available to change the UDR so that it is well-behaved, the only solution is to isolate the code execution from the CPU VP class. Possible execution scenarios include executing:

- In a user-defined VP class
- Locked to one VP or VP class
- As a separate process

The DataBlade API provides the following functions to enable UDRs and DataBlade modules to examine their VP environment and to control portions thereof.

VP-Environment Information	DataBlade API Function
Obtain information about the current VP environment from within a UDR	**mi_vpinfo_classid( )**, **mi_vpinfo_isnoyield( )**, **mi_vpinfo_vpid( )** **mi_class_id( )**, **mi_class_maxvps( )**, **mi_class_name( )**, **mi_class_numvp( )**
Lock the UDR to a VP environment	**mi_module_lock( )**, **mi_udr_lock( )**
Change the VP environment in which a UDR executes	**mi_call_on_vp( )**, **mi_process_exec( )**

Warning: These advanced functions can adversely affect your UDR if you use them incorrectly. Use them only when no regular DataBlade API functions can perform the tasks you need done.

Obtaining VP-Environment Information

By default, the routine manager executes a C UDR in a CPU VP class, which is a yielding VP class. However, execution on the CPU VP implies that the UDR is well-behaved. (For more information, see "Creating a Well-Behaved Routine" on page 13-17.) If your UDR is *not* well-behaved, you can specify that the routine manager execute the UDR in a user-defined VP class. However, a user-defined VP class imposes limitations on the tasks that the UDR can perform. If these limitations are too restrictive for your UDR, the UDR can dynamically obtain information about its VP environment and make decisions about whether to change it.

Warning: The need to examine and possibly change the VP environment should only be done in special cases. For the most efficient execution, a C UDR should be well-behaved and thereby execute safely in the CPU VP. Ill-behaved routines can usually execute in a user-defined VP class without changing the VP environment.

From within a C UDR, you can obtain the following kinds of information about the VP environment:

- Information about the current VP
- Information about the VP class to which the current VP belongs

If a UDR can identify its VP environment, it can sometimes take care of its own migratory needs.

Identifying the Current VP

A VP that is currently performing some task is called an *active VP*. The database server assigns a unique integer, called the VP *identifier*, to each active VP. The **onstat -g glo** command displays the VP identifier in the first column of the output it generates (column with the heading "**vp**"). For example, the **onstat** output in Figure 13-13 on page 13-37 shows information for VPs whose VP identifiers range from 1 to 13.

The VP identifier uniquely identifies the running **oninit** process. You can use it as an identifier for named memory that stores information unique to that VP.

The active VP on which a UDR executes is the current VP for the UDR. To obtain the VP identifier of the current VP, use the **mi_vpinfo_vpid( )** function. Once you have the VP identifier of the current VP, you can use the following functions to obtain additional information about the VP environment of the UDR.

VP-Environment Information	DataBlade API Function
VP-class identifier	**mi_vpinfo_classid( )**
Whether the current VP is part of a nonyielding VP class	**mi_vpinfo_isnoyield( )**

Identifying a VP Class

The database server assigns a unique integer, called the VP-*class identifier*, to each VP class, including:

- System VP classes (such as CPU and AIO)
- User-defined VP classes (which the VPCLASS configuration parameter defines)

You can obtain a VP-class identifier with either of following DataBlade API functions.

DataBlade API Function	VP-Class Identifier Returned
mi_vpinfo_classid( )	The VP-class identifier for the VP class of the current VP (the VP on which the current UDR is running)
mi_class_id( )	The VP-class identifier for a specified VP class

Once you have a VP-class identifier for an active VP, you can obtain the following information about the associated VP class.

VP-Class Information	DataBlade API Function
VP-class name	**mi_class_name( )**
Maximum number of VPs in the VP class	**mi_class_maxvps( )**
Number of active VPs in the VP class	**mi_class_numvp( )**

Changing the VP Environment

If the UDR determines that its VP environment is not correct for its execution requirements, it can perform either of the following tasks to change it.

Change to VP Environment	DataBlade API Function
Execute a specified C function on another VP	**mi_call_on_vp( )**
Fork and execute a new process to perform some task	**mi_process_exec( )**

Executing on Another VP

If the VP environment is not useful for the execution of your C function, you can tell the routine manager to switch its execution to another VP with the **mi_call_on_vp( )** function. Pass the following arguments to this function:

- The VP identifier of the VP on which to execute the C function
- A pointer to the return value of the function
- The address of the C function to execute

- The number of arguments to the C function
- Any arguments that the C function needs to execute

The **mi_call_on_vp( )** function switches the current thread to the specified VP and executes the C function on this VP. When the C function completes, **mi_call_on_vp( )** stores as one of its arguments the C-function return value and returns control to the originating VP.

Forking and Executing a Process

If you need to run some program or script as a separate process, you can use the **mi_process_exec( )** function. The **mi_process_exec( )** function forks and executes a new process and returns immediately. The database server does not wait for completion, and the new process is allowed to run independently.

Warning: Never use the operating-system **fork( )** and **exec( )** calls from within a UDR. These system calls are unsafe within a UDR. (For more information, see "Unsafe Operating-System Calls" on page 13-27.) If you must execute a separate process, use the **mi_process_exec( )** function to create this new process.

The **mi_process_exec( )** function is similar to most operating-system **exec( )** system calls in that you pass the function an **argv** array. This array contains all the command strings that are to be passed after the new process is forked. For more information on the syntax of the **argv** array, see the description of the **mi_process_exec( )** function in the *IBM Informix DataBlade API Function Reference*.

Locking a UDR

If the UDR determines that its VP environment *is* correct and needs to remain as it is, the UDR can perform either of the following tasks.

Lock UDR	DataBlade API Function
Lock the UDR to the current VP	**mi_udr_lock( )**
Lock the shared-object file that contains the UDR into memory	**mi_module_lock( )**

Locking a Routine Instance to a VP

If your UDR allocates resources that are process specific, it needs to be locked onto the VP where it started execution. When you write a UDR that needs access to global process information, you must take either of the following actions:

- Restrict execution of the UDR to a single-instance VP class.

  By executing in a single-instance VP, a UDR can be guaranteed that all invocations and instances execute in the same VP. Therefore, all UDRs can access global information of the process. (For more information, see "Avoiding Modification of Global and Static Variables" on page 13-23.) However, a single-instance VP class does have significant impact on performance and parallel scalability.

- Lock the UDR to a VP with the **mi_udr_lock( )** function.

  When you call **mi_udr_lock( )** with an argument of MI_TRUE, you set the VP lock flag to prevent this instance of the UDR from migrating to another VP. Therefore, the UDR instance *always* executes on the VP where it is running. However, an MI_TRUE VP lock flag does *not* prevent another instance of the UDR from executing on a different VP.

Important: These solutions do not address resource allocations that last longer than the individual routine sequence in a statement or subquery. nor do they address the general issue of reclaiming resources for these sequences.

Locking a Shared-Object File in Memory

If the set of UDRs in a shared-object file requires a lot of initialization or uses external resources, it can be costly to have the routine manager continually load and unload this shared-object file. To prevent the routine manager from unloading a shared-object file, use the **mi_module_lock( )** function. When you call **mi_module_lock( )** with an argument of MI_TRUE, you set the module-lock flag, which locks the shared-object file in memory. Therefore, the routine manager does not allow the shared-object file to be unloaded for any reason.

This feature enables a DataBlade (or group of related UDRs) to prevent its shared-object file from being unloaded in any of the following cases:
- On execution of the DROP FUNCTION, DROP PROCEDURE, DROP ROUTINE, or DROP DATABASE statements
- In various transaction rollback scenarios

Performing Input and Output

Because a C UDR executes in the context of the database server, it should not use the standard input/output (I/O) calls such as **scanf( )** and **printf( )**. The DataBlade API provides the following support for I/O from a UDR:
- I/O on a generic stream
- I/O on an operating-system file

Access to a Stream (Server)

The DataBlade API provides a *stream I/O interface*, which enables you to use the same function calls to access different objects. *Stream* is a generic term for an object that can be written to or read from. A stream has the following information associated with it:
- The *stream data*, to which the stream provides access
- The *stream seek position*, which identifies where the next read or write operation starts in the stream

  When you first open a stream, its seek position is at byte zero (0).
- The *stream descriptor*, which contains information about the stream

To provide access to a stream from within a C UDR, the DataBlade API has the **MI_STREAM** data type structure for stream descriptors. An **MI_STREAM** structure contains information about a stream on a particular object. The following table summarizes the memory operations for a stream descriptor.

Memory Duration	Memory Operation	Function Name
Current memory duration	Constructor	**mi_stream_open_fio( )**, **mi_stream_open_mi_lvarchar( )**, **mi_stream_open_str( )**
		Other, user-defined stream-open functions
	Destructor	**mi_stream_close( )**

To access a stream in your UDR:

1. Open the stream with the appropriate type-specific stream-open function.

   The stream-open function is the stream I/O function that opens the stream, making the data available for a read or write operation. It returns a pointer to a stream descriptor, which the C UDR uses to access the stream. For more information, see "The Stream-Open Function" on page 13-47.

2. Access the opened stream with the appropriate generic stream I/O function.

   Once a particular stream is open, a UDR can use the generic functions of the stream I/O interface to access the associated I/O object. Each of the generic stream I/O functions requires a stream descriptor for the stream on which the function is to operate. The usual sequence of access is to seek to the desired location in the stream, read or write the desired number of bytes, and close the stream.

Table 13-3 shows the generic stream I/O functions of the DataBlade API. You can use these generic stream I/O functions on *any* stream (as long as the stream class implements them).

Table 13-3. Generic Stream I/O Functions

Stream-I/O Function	Description
mi_stream_close( )	Close the stream.
mi_stream_eof( )	Check the stream for the end-of-stream condition.
mi_stream_get_error( )	Obtain the last error that occurred on the specified stream.
mi_stream_getpos( )	Obtain the current stream seek position, returning it in a function parameter.
mi_stream_length( )	Obtain the length of the stream data.
mi_stream_read( )	Read a specified number of bytes from the stream.
mi_stream_seek( )	Move the stream seek position to the desired location.
mi_stream_set_error( )	Sets the last error status on the specified stream.
mi_stream_setpos( )	Set the stream seek position.
mi_stream_tell( )	Obtain the current stream seek position, returning it from the function.
mi_stream_write( )	Write a specified number of bytes to the stream.

The advantage of accessing data through a stream is that the call to the generic stream I/O function is the same, regardless of the format of the underlying data. With these generic stream-I/O functions, the DataBlade API provides a common interface for the transportation and access of data independent of the data type or destination.

For example, the following call to **mi_stream_read( )** reads 164 bytes of data from a stream into a user-defined buffer named **buf**:

```
nbytes_read = mi_stream_read(strm_desc, buf, 164);
```

The calling code does not need to concern itself about the format of the underlying data. Whether **mi_stream_read( )** reads the data from a file, character array, varying-length structure, or user-defined stream depends on which stream-open function has obtained the pointer to the stream descriptor (**MI_STREAM** structure).

In addition to the generic stream I/O functions in Table 13-3 on page 13-43, the stream I/O interface contains the following functions for different stream classes.

Classes of Stream I/O Function	Stream I/O Function	More Information
Stream-open functions for the predefined stream classes:		"Using Predefined Stream Classes" on page 13-44
• File stream	**mi_stream_open_fio( )**	
• String stream	**mi_stream_open_mi_lvarchar( )**	
• Varying-length-data stream	**mi_stream_open_str( )**	
Abstract stream I/O functions for user-defined streams	**mi_stream_init( )**	"Creating a User-Defined Stream Class" on page 13-47
	Type-specific stream-open function	

Using Predefined Stream Classes

The DataBlade API provides several predefined stream classes that you can access with the stream I/O interface.

To use a predefined stream class in your UDR:

1. Open a stream with the appropriate type-specific stream-open function.

   The following table shows the predefined stream classes that the DataBlade API provides and their associated stream-open functions.

Predefined Stream Class	Stream-Open Function
File stream	**mi_stream_open_fio( )**
String stream	**mi_stream_open_str( )**
Varying-length-data stream	**mi_stream_open_mi_lvarchar( )**

   The **mistrmtype.h** header file declares these predefined stream-open functions.

2. Access the open stream with the appropriate stream I/O function.

   Table 13-3 on page 13-43 lists the stream I/O functions that the DataBlade API provides.

For example, the following code fragment reads 26 bytes of data from a string stream into a user-defined buffer named **buf**:

```
#define STRING_SIZE = 80

MI_STREAM *strm_desc;
mi_integer nbytes;
char buf[200];
char string_txt[STRING_SIZE] =
   "A stream is a generic term for some object that can be\
 written to or read from."

strm_desc = mi_stream_open_str(NULL, string_txt, STRING_SIZE);
   if ( (nbytes = mi_stream_read(strm_desc, buf, 26)) != 26 )
      /* error in read */
mi_stream_close(strm_desc);
```

After this code fragment completes, the **buf** user-defined buffer contains the following character string:

```
A stream is a generic term
```

The following sections provide additional details on each of the predefined DataBlade API stream classes.

The File Stream: The *file stream* provides access to an operating-system file through the stream I/O interface. To support a data stream on an operating-system file, the DataBlade API provides the stream I/O functions in Table 13-4.

Table 13-4. Stream I/O Functions for a File Stream

Stream I/O Task	Stream I/O Function
Initialize and open a file stream.	**mi_stream_open_fio( )**
Move the file seek position to the desired location.	**mi_stream_seek( )**
Read a specified number of bytes from the file stream.	**mi_stream_read( )**
Write a specified number of bytes to the file stream.	**mi_stream_write( )**
Obtain the current file seek position, returning it from the function.	**mi_stream_tell( )**
Obtain the current file seek position, returning it in a function parameter.	**mi_stream_getpos( )**
Set the file seek position.	**mi_stream_setpos( )**
Obtain the length of the operating-system file.	**mi_stream_length( )**
Close the file stream.	**mi_stream_close( )**

Tip: You can also use the **mi_stream_get_error( )** and **mi_stream_eof( )** functions on a file stream.

As Table 13-4 shows, the stream I/O interface for a file stream consists of a type-specific stream-open function, **mi_stream_open_fio( )**, plus the generic stream I/O functions. The **mi_stream_open_fio( )** function opens the file and returns a new file stream.

The other stream I/O functions in Table 13-4 handle return status differently from DataBlade API file-access functions because the stream I/O functions do not allow you to obtain the **errno** status value directly. Instead, these functions handle their return status as follows:

- A file-access function returns MI_OK for success and sets **errno** to indicate an error, but a stream I/O function returns the MI_OK status for success and a negative number to indicate an error.

  The stream I/O function maps the values associated with **errno** to DataBlade API constants that have negative values. The **mistream.h** header file defines these constants.

- A file-access function returns the amount written to or read from a file, but a stream I/O function returns either of the following values:
  - On success, the amount written or read
  - On failure, a negative number (defined in **mistream.h**)

The String Stream: The *string stream* provides access to a character array through the stream I/O interface. The string stream does *not* handle character data as null-terminated strings. It does not evaluate the contents of the data stream in any way. To support a data stream on a character array, the DataBlade API provides the stream I/O functions in Table 13-5.

Table 13-5. Stream I/O Functions for a String Stream

Stream I/O Task	Stream I/O Function
Initialize and open a string stream.	**mi_stream_open_str( )**

Table 13-5. Stream I/O Functions for a String Stream (continued)

Stream I/O Task	Stream I/O Function
Move the string seek position to the desired location.	**mi_stream_seek( )**
Read a specified number of bytes from the string stream.	**mi_stream_read( )**
Write a specified number of bytes to the string stream.	**mi_stream_write( )**
Obtain the current string seek position, returning it from the function.	**mi_stream_tell( )**
Obtain the current string seek position, returning it in a function parameter.	**mi_stream_getpos( )**
Set the string seek position.	**mi_stream_setpos( )**
Obtain the length of the character array.	**mi_stream_length( )**
This is the *str_len* value to pass to **mi_stream_open_str( )** when you create the string stream.	
Close the string stream.	**mi_stream_close( )**

Tip: You can also use the **mi_stream_get_error( )** and **mi_stream_eof( )** functions on a string stream.

As Table 13-5 shows, the stream I/O interface for a string stream consists of the generic stream I/O functions plus a type-specific stream-open function, **mi_stream_open_str( )**.

The Varying-Length-Data Stream: The *varying-length-data stream* provides access to the data within a varying-length structure (**mi_lvarchar**) through the stream I/O interface. A varying-length-data stream does *not* handle varying-length data as null-terminated strings. It also does not evaluate the contents of the data stream in any way. To support a data stream on a varying-length structure, the DataBlade API provides the stream I/O functions in Table 13-6.

Table 13-6. Stream I/O Functions for a Varying-Length-Data Stream

Stream I/O Task	Stream I/O Function
Initialize and open a varying-length-data stream.	**mi_stream_open_mi_lvarchar( )**
Move the stream seek position to the desired location.	**mi_stream_seek( )**
Read a specified number of bytes from the varying-length-data stream.	**mi_stream_read( )**
Write a specified number of bytes to the varying-length-data stream.	**mi_stream_write( )**
Obtain the current stream seek position, returning it from the function.	**mi_stream_tell( )**
Obtain the current stream seek position, returning it in a function parameter.	**mi_stream_getpos( )**
Set the stream seek position.	**mi_stream_setpos( )**
Obtain the length of the varying-length data.	**mi_stream_length( )**
Close the varying-length-data stream.	**mi_stream_close( )**

Tip: You can also use the **mi_stream_get_error( )** and **mi_stream_eof( )** functions on a varying-length-data stream.

As Table 13-6 shows, the stream I/O interface for a varying-length-data stream consists of the generic stream I/O functions plus a type-specific stream-open function, **mi_stream_open_mi_lvarchar( )**. This function returns a new varying-length-data stream.

Creating a User-Defined Stream Class

You can provide a stream I/O interface to create your own protocol for reciprocal reading and writing of SQL data and other data streams. The DataBlade API stream I/O interface provides a consistent interface for accessing data; that is, each stream I/O function has a fixed function name and argument list, regardless of the actual kind of stream that it accesses. This fixed syntax provides the main benefits of stream access:

- The calling code can use the exact same syntax to access different kinds of data.
- The underlying data can be transparent to the calling code.

Important: Enterprise Replication does not support user-defined stream classes.

To create a user-defined stream class, you need to write the following stream I/O functions:

- A type-specific stream-open function

  Each type of data to which a stream provides access usually has a unique way of being opened. Its stream-open function must accept as arguments the information required to open the data so that the **mi_stream_init( )** function can initialize the stream.

- Type-specific implementations for the generic stream I/O functions

  You must implement the generic stream I/O functions that your stream supports so that they correctly handle the format of your stream data.

The **mi_stream_init( )** function initializes the stream descriptor with the arguments it receives. The following code fragment of a stream-open function calls **mi_stream_init( )** with the stream-operations structure in Figure 13-15, the internal structure for the **mytype** opaque type, and a NULL-valued pointer:

```
MI_STREAM *mi_stream_open_mytype(void *mydata)
{
   MI_STREAM *strm_desc; /* could be passed in as input to open( )
                          * also.
                          */
/* Code to process any stream-open arguments */
...
/* Call to mi_stream_init( ) to allocate and initialize
 * the stream descriptor
 */
   strm_desc = mi_stream_init(stream_ops_mytype, mydata, NULL);

/* Return pointer to newly allocated stream descriptor */
   return strm_desc;
}
```

Because **mi_stream_init( )** receives a NULL-valued pointer as its stream descriptor, it allocates the stream descriptor in the current memory duration. The **mi_stream_init( )** function then returns a pointer to this newly allocated structure, which the **mi_stream_open_mytype( )** function also returns.

The Stream-Open Function: Your stream-open function must take the following steps:

1. Accept as its arguments the type-specific initialization information and use them to open the data.

2. Call **mi_stream_init( )** with appropriate information to initialize an
   MI_STREAM structure (see Table 1-4 on page 1-12).

The stream-open function must prepare the arguments for the call to the
mi_stream_init( ) function, which initializes and optionally allocates an
MI_STREAM structure. The **mi_stream_init( )** function takes the following
arguments:

- The stream-operations structure
- The stream data
- A stream descriptor

The Stream-Operations Structure: The *stream-operations structure* contains pointers to
the C functions that implement the generic stream I/O functions for the particular
stream. A valid stream-operations structure must exist for the DataBlade API to
locate at runtime your type-specific implementations of these generic stream I/O
functions. Therefore, it must be initialized *before* the call to **mi_stream_init( )**.

Figure 13-14 shows the declaration of the stream-operations structure, **mi_st_ops**.
For the most current definition, see the **mistream.h** header file.

```
#define OPS_NAME_LENGTH 40

struct mi_stream_operations {
   /* the pointers to the functions */
   mi_integer (*close)(MI_STREAM *strm_desc);
   mi_integer (*read)(MI_STREAM *strm_desc, void *buf,
      mi_integer nbytes);
   mi_integer (*write)(MI_STREAM *strm_desc, void *buf,
      mi_integer nbytes);
   mi_integer (*seek)(MI_STREAM *strm_desc,
      mi_int8 *offset, mi_integer whence);
   mi_int8 *  (*tell)(MI_STREAM *strm_desc);
   mi_integer (*setpos)(MI_STREAM *strm_desc,
      mi_int8 *pos);
   mi_integer (*getpos)(MI_STREAM *strm_desc,
      mi_int8 *pos);
   mi_integer (*length)(MI_STREAM *strm_desc,
      mi_int8 *length);
   /* names of the functions above */
   char close_name [OPS_NAME_LENGTH];
   char read_name   [OPS_NAME_LENGTH];
   char write_name [OPS_NAME_LENGTH];
   char seek_name   [OPS_NAME_LENGTH];
   char tell_name   [OPS_NAME_LENGTH];
   char setpos_name[OPS_NAME_LENGTH];
   char getpos_name[OPS_NAME_LENGTH];
   char length_name[OPS_NAME_LENGTH];
   /* the function handles for the functions above */
   void *close_fhandle;
   void *read_fhandle;
   void *write_fhandle;
   void *seek_fhandle;
   void *tell_fhandle;
   void *setpos_fhandle;
   void *getpos_fhandle;
   void *length_fhandle;
} mi_st_ops;
```

Figure 13-14. The Stream-Operations Structure

As Figure 13-14 shows, the stream-operations structure consists of the following parts:

- The function pointers to the generic stream I/O functions
- The names of the generic stream I/O functions
- The function handles of the generic stream I/O functions

You should initialize the pointers and names and set the handles to NULL.

Figure 13-15 shows a sample stream-operations structure that provides function pointers for the type-specific implementations of the **mi_stream_close( )**, **mi_stream_read( )**, and **mi_stream_write( )** functions for a stream on a user-defined type named **newstream**.

```
static struct mi_st_ops stream_ops_newstream =
{
   stream_close_newstream,
   stream_read_newstream,
   stream_write_newstream,
   NULL,
   NULL,
   NULL,
   NULL,
   NULL,
   "stream_close_newstream",
   "stream_read_newstream",
   "stream_write_newstream",
   NULL,
   NULL,
   NULL,
   NULL,
   NULL,
   NULL,
   NULL,
   NULL,
   NULL,
   NULL,
   NULL,
   NULL
};
```

Figure 13-15. A Sample Stream-Operations Structure

The code fragment in Figure 13-15 statically initializes the stream-operations structure. If you initialize this structure dynamically, do so in the stream-open function.

The Stream Data: The second argument to **mi_stream_init( )** is an uninterpreted data pointer that is stored in the **MI_STREAM** structure initialized by the call to **mi_stream_init( )**. The stream interface does not interpret this pointer, which is for the benefit of the stream implementer. You can retrieve the value of this pointer through a call to **mi_stream_get_dataptr( )**.

The Stream Descriptor: A *stream descriptor* holds information about the stream that all stream I/O functions need to access. The **mi_stream_init( )** function accepts as its stream-descriptor argument either of the following values:

- A NULL-valued pointer
- A pointer to a valid, allocated **MI_STREAM** structure

When you pass the **mi_stream_init( )** function a NULL-valued pointer for its stream-descriptor argument, the function allocates a new stream descriptor in the current memory duration. If your application requires a specific memory duration for the stream descriptor, your stream-open function can perform one of the following tasks:

- *Before* the call to **mi_stream_init( )**, change the current memory duration to what is required.

  The **mi_switch_mem_duration( )** function changes the current memory duration. Its return value is the previous current duration so that you can return the duration to its original value. For more information, see "Changing the Memory Duration" on page 14-22.

  In this case, pass a NULL-valued pointer as the stream descriptor to **mi_stream_init( )** so that **mi_stream_init( )** allocates a new stream descriptor in the new current memory duration.

- Allocate a stream descriptor in the required memory duration.

  In this case, pass a pointer to the allocated stream descriptor as the stream descriptor for **mi_stream_init( )** so that this function does *not* allocate a new stream descriptor. The **mi_stream_close( )** function does not automatically free a stream descriptor that your stream-open function allocates. Your code must handle the deallocation.

Initialization of the Stream Descriptor: After your type-specific stream-open function has prepared the arguments for the **mi_stream_init( )** function, it must call **mi_stream_init( )** to initialize the stream descriptor.

Tip: Whether the **mi_stream_init( )** function actually allocates the stream descriptor depends on the value of its third argument. For more information, see "The Stream Descriptor" on page 13-49.

The *stream descriptor*, **MI_STREAM**, holds information about the data stream such as the data and its seek position. For the most current definition of the **MI_STREAM** structure, see the **mistream.h** header file.

Support for Stream Access: To provide access to the data in your user-defined stream, you must implement the appropriate generic stream I/O functions. The following table shows which stream I/O functions to implement for the stream characteristics that your stream supports.

Stream Characteristic	Description	Stream I/O Function
Stream seek position	The location within the data at which the next read or write operation begins	**mi_stream_seek( )**, **mi_stream_tell( )**, **mi_stream_getpos( )**, **mi_stream_setpos( )**
Stream length	The size of the data This length can be the size of the data when the stream is initialized or the current size of the data.	**mi_stream_length( )**
Stream mode	Which operations are valid: read-only, read/write, or write-only	**mi_stream_read( )**, **mi_stream_write( )**

Tip: You do not have to implement the stream I/O functions **mi_stream_get_error( )** and **mi_stream_eof( )** for your user-defined stream. The implementation of these functions is generic for any stream.

Consider the following information when deciding which stream I/O functions to implement:

- Implement *only* those stream I/O functions needed to support the selected stream mode.

  If your stream is to be read-only, you need to implement the **mi_stream_read( )** function but *not* the **mi_stream_write( )** function. For a write-only stream, implement only the **mi_stream_write( )** function. For a read/write stream, implement *both* **mi_stream_read( )** and **mi_stream_write( )**.

- Implement stream I/O functions that access the stream seek position *only* if your stream supports a seek position.

  If your stream supports a seek position, you must maintain the **st_pos** field of the stream descriptor. You can choose whether to support one or two methods of accessing the stream seek position:

  - The **mi_stream_seek( )** function provides specification of the stream seek position through an offset and a "whence" stream position.

  - The **mi_stream_getpos( )** function provides specification of the stream seek position through an absolute position.

  The **mi_stream_tell( )** function returns the current stream seek position as its return value. This function cannot return any negative error value to indicate the cause of an error.

  The **mi_stream_setpos( )** function returns the current stream seek position as one of its parameters. This function can return an integer status value.

  If your stream does *not* have a seek position, you do not need to write any of the following functions: **mi_stream_seek( )**, **mi_stream_tell( )**, **mi_stream_getpos( )**, or **mi_stream_setpos( )**.

- Implement an **mi_stream_close( )** function to deallocate stream resources.

  The type-specific implementation of **mi_stream_close( )** must explicitly free any memory that the associated stream-open function (or any other of the generic stream I/O functions) has allocated. For information, see "Releasing Stream Resources" on page 13-52.

The following general rules apply to values that the generic stream I/O functions return:

- All stream I/O functions *except* **mi_stream_tell( )** must return the following values:
  - On success, MI_OK
  - On failure, a negative integer defined in the **mistream.h** header file
- The **mi_stream_tell( )** function must return the following values:
  - On success, a valid pointer to the current stream seek position, an **mi_int8** value
  - On failure, a NULL-valued pointer

Registering a UDR That Accesses a Stream

To declare a stream as an argument or return value of a C UDR, use the **MI_STREAM** data type. When you register this UDR in the database, use the opaque data type **stream** to represent the stream descriptor.

The database server represents a stream with the **stream** opaque type. As for other opaque types, the database server stores information on **stream** in the **sysxtdtypes** system catalog table.

For example, suppose you have a C declaration for a UDR named **get_data( )**:

```
mi_lvarchar *get_data(strm_desc, nbytes)
   MI_STREAM *strm_desc;
   mi_integer nbytes;
```

The following CREATE FUNCTION statement registers the **get_data( )** UDR, using the **stream** data type as its first argument:

```
CREATE FUNCTION get_data(data_source stream, nbytes INTEGER)
RETURNS VARCHAR
EXTERNAL NAME '/usr/local/udrs/stream/stream.so(get_data)'
LANGUAGE C;
```

Releasing Stream Resources

When your DataBlade API module no longer needs a stream, you need to assess whether you can release resources that the stream is using. A stream descriptor that the **mi_stream_init( )** function allocated has the current memory duration, so it remains valid until one of the following events occurs:

- The **mi_stream_close( )** function closes the stream, freeing the stream descriptor.
- The current memory duration expires.

To conserve resources, use the **mi_stream_close( )** function to deallocate the stream descriptor explicitly when your DataBlade API module no longer needs it. The **mi_stream_close( )** function is the destructor function for a stream descriptor. This function frees a stream descriptor that **mi_stream_init( )** allocated and any associated resources, including the stream-data buffer.

The **mi_stream_close( )** function does not automatically free a stream descriptor allocated by your stream-open function. If the **mi_stream_init( )** function does not allocate a stream descriptor, your type-specific implementation of **mi_stream_close( )** must handle the deallocation.

Access to Operating-System Files

The DataBlade API provides file-access functions for access to operating-system files from within a C UDR. These functions provide file management that is similar to what operating-system file-access functions provide. The DataBlade API file-access functions call the corresponding operating-system functions to perform their tasks; however, the DataBlade API functions periodically yield the virtual processor to limit the effects of blocking I/O.

Important: Do not call operating-system file I/O functions from within a C UDR. Use these DataBlade API file-access functions instead because they are safer in a C UDR than their operating-system equivalents. For more information, see "Avoiding Blocking I/O Calls" on page 13-20.

Table 13-7 lists the DataBlade API functions for the basic file-access operations and the analogous operating-system calls for these operations.

Table 13-7. DataBlade API File-Access Functions

File-Access Operation	File-Access Function	Operating-System Call
Open an operating-system file and generate a file descriptor for the file	**mi_file_open( )**	**open( )**
Seek to a specified position to begin a read or write operation	**mi_file_seek( )**	**seek( )**
Obtain the current seek position	**mi_file_tell( )**	**tell( )**

Table 13-7. DataBlade API File-Access Functions (continued)

File-Access Operation	File-Access Function	Operating-System Call
Perform a read or write operation for a specified number of bytes	**mi_file_read( )**, **mi_file_write( )**	**read( )**, **write( )**
Obtain status information about a specified smart large object	**mi_file_sync( )**	**sync( )**
Close an operating-system file and deallocate the file descriptor	**mi_file_close( )**	**close( )**
Unlink (remove) an operating-system file	**mi_file_unlink( )**	**unlink( )**
Obtain an **errno** value for the file operation	**mi_file_errno( )**	GLOBAL INT ERRNO;

Tip: The DataBlade API file-access functions execute in client LIBMI applications as well as C UDRs. For DataBlade API modules that you design to run in both client LIBMI applications and UDRs, use these file-access functions. For information on the behavior of these functions in a client LIBMI application, see Appendix A, "Writing a Client LIBMI Application," on page A-1.

The DataBlade API accesses operating-system files through *file descriptors*. These file descriptors are similar in purpose to operating-system file descriptors. The following table summarizes the memory durations for a file descriptor.

Memory Duration	Memory Operation	Function Name
Duration of session (PER_SESSION)	Constructor	**mi_file_open( )**
	Destructor	**mi_file_close( )**, **mi_file_unlink( )**

Opening a File

The **mi_file_open( )** function is the constructor function for a file descriptor. Through the file descriptor, you access an operating-system file. This section provides the following information on how to open a file:

- How to specify the filename, including its path
- How to specify the open flags and open mode, which the underlying operating-system call supports
- How UDRs can share open files

Specifying a Filename: The filename argument of **mi_file_open( )** identifies the operating-system file to open. This filename is relative to the server computer. You can include an environment variable in the filename path for the **mi_file_open( )** and **mi_file_to_file( )** file-access functions. This environment variable must be set in the database server environment; that is, it must be set *before* the database server starts.

For example, Figure 13-16 shows an **mi_file_open( )** call that opens the operating-system file **data_file1**, which resides in the directory that the **DATA_FILES** environment variable specifies.

```
fd = mi_file_open("$DATA_FILES/data_file1",
    O_WRONLY | O_APPEND | O_CREAT, 0644);
```

Figure 13-16. Sample Call to Open an Operating-System File

Suppose the **DATA_FILES** environment variable is set to the following directory in the database server environment:

/usr/local/app/load_files

The call to **mi_file_open( )** in Figure 13-16 opens the following file:

/usr/local/app/load_files/data_file1

Calling the Operating-System Open Call: To open a file, the **mi_file_open( )** function calls the **open** system call that your operating system supports.

─────────────────────── **UNIX/Linux Only** ───────────────────────

On UNIX or Linux, the **open( )** system call opens an operating-system file.

─────────────────────── **End of UNIX/Linux Only** ───────────────────────

─────────────────────── **Windows Only** ───────────────────────

On Windows, the **_open** command opens an operating-system file.

─────────────────────── **End of Windows Only** ───────────────────────

The **mi_file_open( )** function provides the following information about the file to the appropriate system call.

Argument of mi_file_open( )	Information Provided
open_flags (second argument)	Access mode for the operating-system file
open_mode (third argument)	Open mode for the operating-system file.

The function takes this information and passes it directly to the underlying operating-system call. Therefore, **mi_file_open( )** supports the access modes and open modes that your operating-system **open** call supports.

Tip: For more information on the open flags and open mode, see the documentation for your operating-system **open** call.

Specifying Open Flags: The **mi_file_open( )** function takes as its second argument the open flags with which to open the operating-system file. The *open flags* value provides two pieces of information:

- A masked flag value that specifies information such as access mode (read/write, read-only, write-only)

  The **mi_file_open( )** function passes these open flags directly to the underlying operating-system call that opens a file, so you must use flag values that your operating system supports. Also, you must include the operating-system header file (such as **fcntl.h**) that defines the open-flag constants you use.

- A file-mode flag to indicate on which computer the file to open resides

  The DataBlade API file-access functions support access to a file on either the server or client computer. By default, the **mi_file_open( )** function opens a file

on the server computer. To open a server file, you can omit the file-mode flag or specify the MI_O_SERVER_FILE file-mode flag. To open a client file, you must include the MI_O_CLIENT_FILE file-mode flag as part of the open flags.

For example, the **mi_file_open( )** call in Figure 13-16 on page 13-54 masks the following open flags for the file to open.

Open-Flag Constant	Purpose
O_WRONLY	Open the file write-only.
O_APPEND	Append new data to the end of the file.
O_CREAT	If the file does not exist, create it.

The example in Figure 13-16 on page 13-54 is based on the following assumptions:
- The operating-system open call supports the O_WRONLY, O_APPEND, and O_CREAT flags.
- The code that executes this **mi_file_open( )** call includes the header file that defines O_WRONLY, O_APPEND, and O_CREAT.
- The file resides on the server computer.

Specifying the Open Mode: The **mi_file_open( )** function takes as its third argument the open mode in which to open the operating-system file. The *open mode* specifies the ownership of the file. The **mi_file_open( )** function passes this open mode directly to the underlying operating-system call. The semantics for *mode* must match those that the underlying operating-system call supports.

For example, the **mi_file_open( )** call in Figure 13-16 on page 13-54 specifies an open mode of 0644:
- Read/write for owner
- Read-only for group
- Read-only for general public

Sharing Open Files: All UDRs that execute under the same connection can share a file (because they have the same connection descriptor). For example, if UDR1 opens a file, UDR2 can read, write to, or close this file, as long as these two UDRs execute under the same connection. However, UDRs that do *not* execute under the same connection cannot share a file.

The DataBlade API generates an error if your UDR attempts any of the following file I/O tasks:
- To access a file that a UDR outside the session opened
- To access a file that was not opened at all
- To access a file that was opened and was closed

Closing a File
To close an operating-system file, free the associated file descriptor. A file descriptor remains active until either of the following events occurs:
- The **mi_file_close( )** function explicitly closes the file.
- The client application ends the session.

In a C UDR, a file descriptor has a memory duration of PER_SESSION. Files
remain open after the UDR closes the connection.

In a client LIBMI application, a connection descriptor has a scope of the session.
For more information, see "Accessing Operating-System Files in Client LIBMI
Applications" on page A-3.

Copying a File

The DataBlade API provides the **mi_file_to_file( )** function to enable you to copy
an operating-system file between the computer on which the database server runs
and the client computer. This function provides the *open-mode* flags for the new
operating-system file. Unlike the **mi_file_open( )** function, these open-mode flags
are *not* those of the underlying operating-system open function. Instead,
mi_file_to_file( ) supports the set of DataBlade API file-mode constants in
Table 6-21 on page 6-59.

Sample File-Access UDR

The following sample UDR, **logmsg( )**, uses the DataBlade API file-access
functions to output messages to an external file:

```
#include <mi.h>
#include <fcntl.h>
#include <errno.h>

void logmsg (filename, message, Gen_fparam)
   mi_lvarchar *filename,
   mi_lvarchar *message,
   MI_FPARAM *Gen_fparam
{
   mi_integer fd, /* file descriptor */
           ret, /* return status from file-access funcs
                 *functions
                 */
           error; /* mi_file_errno( ) errno return */

   mi_string pathname[256], /* mi_lvarchar_to_buffer( ) result */
           *msg_str, /* mi_lvarchar_to_string( ) result */
           *newline = "\n", /* output new line */
            msg_error[150], /* errno error message */
            tmp_error[150], /* temp error message */
           *p;

   if ( mi_get_varlen(filename) >= sizeof(pathname) )
      {
      mi_db_error_raise(NULL, MI_EXCEPTION,
         "Pathname exceeded 255 characters!");
      return;
      }

   mi_var_to_buffer(filename, pathname);
   msg_str = mi_lvarchar_to_string(message);

   fd = mi_file_open(pathname,
      O_WRONLY | O_APPEND | O_CREAT, 0644);
```

```
        if ( fd == MI_ERROR )
            {
            error = mi_file_errno( );
            switch( error )
                {
                /* Include your favorite errors from
                 * /usr/include/sys/errno.h.
                 */
                case ENOENT:
                    p = "No such file or directory";
                    break;
                case EACCES:
                    p = "Permission denied";
                    break;
                case EISDIR:
                    p = "Pathname is a directory instead of file";
                    break;
                default:
                    p = "Unhandled errno case";
                    break;
                }

            tmp_error = "logmsg: mi_file_open( ) failed for";
            sprintf(msg_error, "%s '%s' -- %s (errno=%d)",
                tmp_error, pathname, p, error);

            mi_db_error_raise(NULL, MI_EXCEPTION, msg_error);
            return; /* not reached */
            }

    ret = mi_file_write(fd, msg_str, strlen(msg_str));
    if( ret == MI_ERROR )
        {
            error=mi_file_errno( );
            switch( error )
                {
                case ENOSPC:
                    p = "No space left on device";
                    break;
                default:
                    p = "Unhandled errno case";
                    break;
                }

            tmp_err = "logmsg: mi_file_write( ) failed for"
            sprintf(msg_error, "%s '%s' -- %s (errno=%d)",
                tmp_err, pathname, p, error);
            mi_db_error_raise(NULL, MI_EXCEPTION, msg_error);
            return; /* not reached */
            }

    ret = mi_file_write(fd, newline, strlen(newline));
    if( ret == MI_ERROR )
        {
        mi_db_error_raise(NULL, MI_EXCEPTION,
            "mi_file_write( ) failed for newline!");
        return;
        }

    mi_file_close(fd);
    mi_free(msg_str); /* mi_lvarchar_to_string( ) allocated
                       * result
                       */

    return;
}
```

Accessing the UDR Execution Environment

When the UDR obtains a session, its execution environment is made up of the following environments:

- The session environment, which describes the current connection
- The server environment, which describes the environment in which the database server executes

Accessing the Session Environment

The *session environment* describes the current session, which includes the database server and open database that are in effect when the client application called the SQL statement that contains the UDR. The UDR obtains its session environment when it obtains a connection descriptor with the **mi_open( )** function.

The following DataBlade API functions provide information about the session environment of a UDR.

Session-Environment Information	DataBlade API Function
Connection parameters: • Name of the database server • Server port for a connection	**mi_get_connection_info( )**, **mi_get_default_connection_info( )**
Database parameters: • Name of the open database • Name of the account and password for the user that established the connection	**mi_get_database_info( )**, **mi_get_default_database_info( )**
Database options: • ANSI compliant • Transaction logging • Exclusive mode	**mi_get_connection_option( )**

─────────────── **Global Language Support** ───────────────

The session environment also includes the following locale information:

- The server-processing locale, which the database server creates when the client application establishes a connection)
- The server locale (which is the locale that the database server uses to read and write its own files)

You can obtain the name of the server locale from the connection-information descriptor (**MI_CONNECTION_INFO**) with the **mi_get_connection_info( )** or **mi_get_default_connection_info( )** function. For more information on these locales, see the *IBM Informix GLS User's Guide*.

─────────── **End of Global Language Support** ───────────

Tip: You can use the **mi_get_id( )** function to obtain the session identifier for the session. A session identifier uniquely identifies the session.

Accessing the Server Environment

The *server environment* describes the environment of the database server in which the UDR executes. The server environment is established when the database server

is initialized with the **oninit** utility (or its equivalent). The operating-system process that runs the **oninit** utility (or its equivalent) is called the *server-initialization process*. This process invokes the *database server instance*.

The server environment includes the following information.

Server-Environment Information	How It Is Established
Environment variables	The environment variables set for the server-initialization process
File-access permissions	The file-access permissions of the server-initialization process
Configuration parameters	The ONCONFIG file that is current for the database server
Working directory	The working directory of the server-initialization process

───────────────── **Global Language Support** ─────────────────

The server environment includes the value of the **SERVER_LOCALE** environment variable, which can specify a nondefault server locale. Values of the **DB_LOCALE** and **CLIENT_LOCALE** environment variables in the server locale do *not* necessarily apply to the UDR. While it executes, a UDR obtains the client and database locales from the server-processing locale (which the database server creates when the client application establishes a connection). You can obtain the current value of **DB_LOCALE** with the **mi_get_db_locale( )** function. For more information on these locales, see the *IBM Informix GLS User's Guide*.

───────────── **End of Global Language Support** ─────────────

The UDR obtains its server environment when it begins execution. You can obtain the values of the server-environment variables and configuration parameters with the **mi_get_serverenv( )** function.

Chapter 14. Managing Memory

In This Chapter

A C user-defined routine (UDR) has access to the following types of memory:

- Shared memory for dynamic allocations
- Stack memory for routine arguments (including the **MI_FPARAM** structure), local stack variables, return values

The DataBlade API provides functions to manage these types of memory.

	DataBlade API Function		
Memory-Allocation Task	**Allocation**	**Deallocation**	**Other**
Shared memory			
User memory	**mi_alloc( )**, **mi_dalloc( )**, **mi_realloc( ))**, **mi_zalloc( )**	**mi_free( )**	**mi_switch_mem_duration( )**
Named memory	**mi_named_alloc( )**, **mi_named_zalloc( )**	**mi_named_free( )**	**mi_named_get( )**, **mi_lock_memory( )**, **mi_try_lock_memory( )**, **mi_unlock_memory( )**
Stack memory for routine arguments	**mi_call( )**	None	None

This chapter describes each of these kinds of memory management in detail.

Understanding Shared Memory

When a C UDR executes in a virtual processor (VP), it allocates memory from the shared memory of the database server. To perform this allocation, the UDR takes the following steps:

1. Ensures that dynamic memory allocations come from the shared memory of the database server

   All virtual processors can access database server shared memory.

2. Chooses a memory duration to associate with this memory

   The database server automatically reclaims its shared memory through an associated memory duration.

Accessing Shared Memory

A C UDR executes in a virtual processor, which is associated with an operating-system process. While a C UDR executes on a VP (VP #1), it can access memory that is associated with that virtual processor. This memory space includes the stack, heap, and data segments of the VP. Figure 14-1 shows a schematic representation of what a virtual processor that has loaded a shared-object file looks like internally.
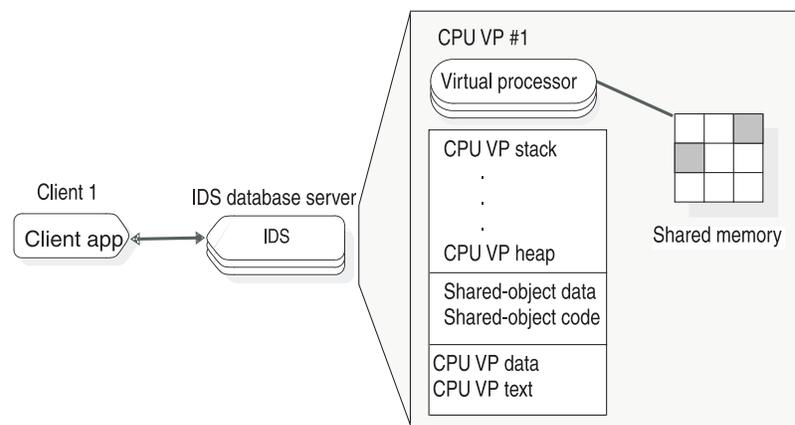
Figure 14-1. VP Memory Space for a C UDR

If the UDR needs to perform some noncomputational task (such as I/O), the database server migrates its thread to the appropriate VP class. When this noncomputational task is complete, the database server migrates the thread back to a computational VP (such as the CPU VP). Once the UDR migrates from VP #1 to another VP (VP #2), it no longer has access to any information in the memory space of VP #1. It can now only access the memory space of the new VP. The only memory that the UDR *can* access from both VP #1 and VP #2 is the database server shared memory. This restriction leads to the following guidelines for the dynamic memory allocation in a C UDR:

- The C UDR must be threadsafe.

  The UDR must *not* assume that it can always access information that is stored in the VP memory space. This guideline is part of the requirements for a well-behaved UDR. For more information, see "Creating a Well-Behaved Routine" on page 13-17.

- The C UDR must use the DataBlade API memory-management functions to allocate memory dynamically.

The DataBlade API memory-management functions in Table 14-7 on page 14-19 allocate memory from the shared memory of the database server, not from the memory space of a virtual processor, as Figure 14-2 shows.
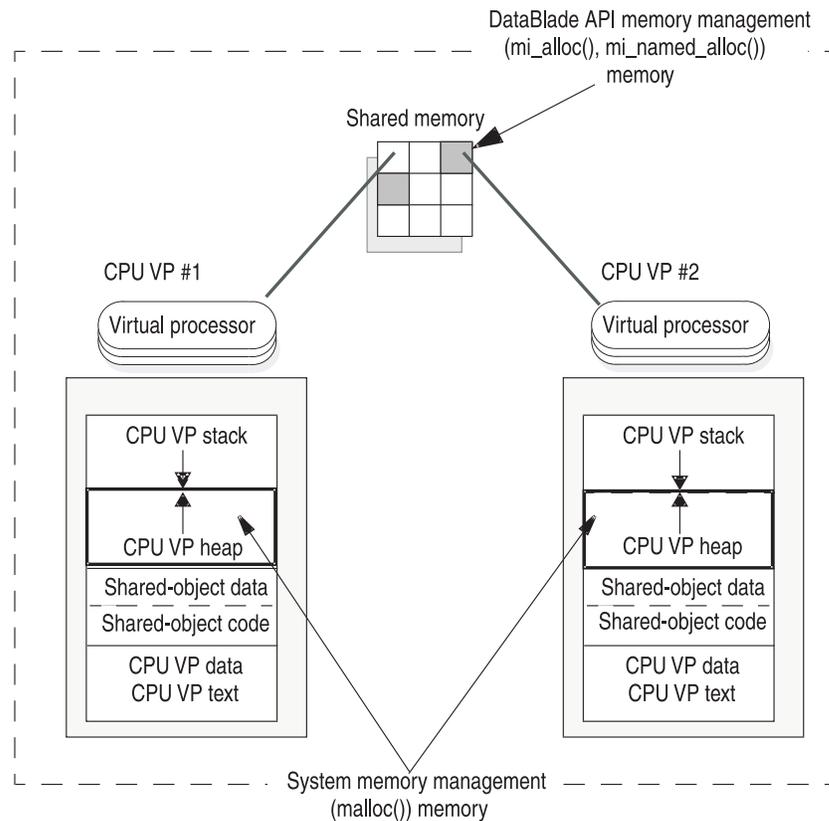
Figure 14-2. Location of Dynamically Allocated Memory for a C UDR

The DataBlade API memory-management functions allocate memory from shared memory, which remains accessible if a thread migrates to another virtual processor. All VPs can access information in memory that these memory-management functions allocate because all VPs can access the shared memory of the database server.

The system memory-management functions (such as **malloc( )** and **calloc( )**) allocate memory in the heap space of the VP. If a UDR migrates to another VP, it no longer has access to the heap space of the previous VP. Therefore, the address to dynamic memory in some variable is not valid once the UDR executes in the new VP.

Important: A C UDR must dynamically allocate memory from the shared memory of the database server, not from the memory of the VP that runs the UDR. Therefore, a C UDR must use the DataBlade API memory-management functions for all dynamic memory allocation.

To ensure that a C UDR does not retain unnecessary amounts of shared memory, it must use the following guidelines for the dynamic memory allocation:

• The C UDR must ensure that it can access both the memory and its address when it needs to.

Both the memory and the memory address must have a memory duration sufficient for all UDRs that need to access the information. For more information, see "Memory-Duration Considerations" on page 14-17.

- The C UDR must use the DataBlade API memory-management functions to dynamically allocate memory that has an associated memory duration.

The DataBlade API memory-management functions allocate memory from the memory-duration memory pools of the database server shared memory. Therefore, the database server can automatically reclaim this memory, reducing the chance of memory leaks. For more information, see "Managing Shared Memory" on page 14-19.

Choosing the Memory Duration

Because a C UDR executes in the memory space of the database server, its dynamic memory allocations can increase the memory usage of the database server. For this reason, it is *very* important that a UDR release its dynamically allocated memory as soon as it no longer needs to access this memory.

To help ensure that unneeded memory is freed, the database server associates a *memory duration* with memory allocation made from its shared memory. The portion of shared memory that the database server provides for dynamic allocation by C UDRs is organized into several *memory pools*. Each memory pool is associated with a memory duration, which specifies the lifetime of the memory allocated from the pool. Keeping related memory allocations in one pool helps to reduce memory fragmentation.

Figure 14-3 shows a schematic representation of the shared memory of the database server, including the memory-duration memory pools.

Database server shared memory

Other server memory pools

PER_STMT_PREP memory pool

PER_COMMAND memory pool

PER_ROUTINE memory pool

PER_STMT_EXEC memory pool

PER_TRANSACTION memory pool   PER_SESSION memory pool   PER_SYSTEM memory pool
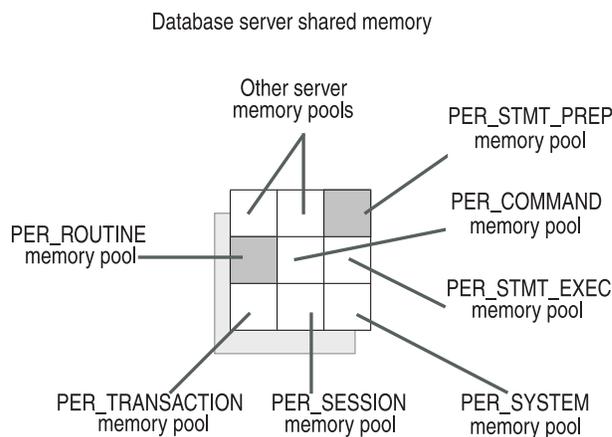
Figure 14-3. Memory-Duration Memory Pools in Database Server Shared Memory

Tip: For more information about the use and structure of database server memory pools, see your *IBM Informix Administrator's Guide*. For more information on how to monitor the amount of shared memory that exists in each of the memory pools, see "Monitoring Shared Memory" on page 14-33.

When the database server calls a UDR, it creates a *memory context*. This context records all of the allocations that the UDR makes before the routine returns. The UDR might run for some time, calling other UDRs or DataBlade API functions. The database server automatically reclaims shared memory based on its memory

duration. When a particular memory duration expires, the database server marks the associated memory pool for deallocation.

The DataBlade API provides the following regular and advanced groups of memory durations for dynamically allocated memory in C UDRs:

- Use the following *public* memory durations in *all* UDRs.

Available Memory Durations	Memory-Duration Constant
Current memory duration	PER_ROUTINE (by default)
For the duration of one iteration of the UDR	PER_ROUTINE, PER_FUNCTION
For the duration of the current SQL command	PER_COMMAND
For the duration of the current SQL statement	PER_STATEMENT *(Deprecated)*
For the duration of the *execution* of the current SQL statement	PER_STMT_EXEC
For the duration of the current prepared SQL statement	PER_STMT_PREP

   Most memory allocations can be allocated with a regular memory duration.
- Use the following *advanced* memory durations *only* in specialized cases.

Available Memory Durations	Memory-Duration Constant
For the duration of the current transaction	PER_TRANSACTION
For the duration of the current session	PER_SESSION
For the duration of the database server execution	PER_SYSTEM

   These memory durations are quite long and therefore increase the chance of memory leaks.

Warning: The advanced memory durations can adversely affect your UDR if you use them incorrectly. Use them only when no regular DataBlade API memory duration can perform the task you need.

Public Memory Durations

The DataBlade API memory-management functions support several *public* memory durations. A UDR can use a public memory duration for most dynamic allocations of memory. The DataBlade API provides the public memory durations that Table 14-1 shows.

Table 14-1. Public Memory Durations

Public Memory Duration	Memory-Duration Constant	Description
For the duration of one iteration of the UDR	PER_ROUTINE, PER_FUNCTION	The database server frees the memory after the UDR returns.
For the duration of the current SQL subquery	PER_COMMAND	The database server frees memory when an SQL command terminates.
For the duration of the current SQL statement	PER_STATEMENT *(Deprecated)*	The database server frees memory when an SQL statement terminates.
For the duration of the execution of the current SQL statement	PER_STMT_EXEC	The database server frees memory when the execution of an SQL statement is complete.

Table 14-1. Public Memory Durations (continued)

Public Memory Duration	Memory-Duration Constant	Description
For the duration of the current prepared SQL statement	PER_STMT_PREP	The database server frees memory when a prepared SQL statement terminates.

The PER_ROUTINE and PER_COMMAND memory durations are the most common for C UDRs. The memory-duration constants in Table 14-1 are of type **MI_MEMORY_DURATION**, which the **memdur.h** header file defines. All memory-duration constants in Table 14-1 are also declared in the **memdur.h** header file.

PER_ROUTINE Memory Duration: A PER_ROUTINE memory pool is associated with each UDR invocation. A *routine invocation* is one single execution of a UDR within a routine instance.

Tip: The two memory-duration constants PER_ROUTINE and PER_FUNCTION are synonyms for the same memory duration. PER_ROUTINE is the more current name.

When a C UDR allocates PER_ROUTINE memory, this memory is available to code within that single routine invocation of that UDR. The database server reclaims any PER_ROUTINE memory in the memory context when a single invocation of a UDR completes. This memory is actually freed on entry to the *next* routine invocation. The database server does *not* reclaim any memory in the memory context with a higher duration than PER_ROUTINE.

In a C UDR, the PER_ROUTINE memory duration is useful for information required for a single UDR invocation. A UDR *cannot* allocate memory, save a pointer to this memory in static space, and expect the pointer to be valid for the next routine invocation. To save information across invocations, use the user-state pointer of the **MI_FPARAM** structure. For more information, see "Saving a User State" on page 9-8.

Several DataBlade API constructor functions allocate their DataBlade API data type structure with a PER_ROUTINE memory duration. Table 14-2 shows the DataBlade API data type structures that have a memory duration of PER_ROUTINE.

Table 14-2. DataBlade API Data Type Structures with a PER_ROUTINE Memory Duration

DataBlade API Data Type Structure	DataBlade API Constructor Function	DataBlade API Destructor Function
UDR arguments that are passed by reference	Routine manager (when it invokes a UDR)	Routine manager (when it exits a UDR)
UDR return value that is passed by value	UDR with its declaration of its return value	Routine manager (when it exits a UDR)
UDR return value that is passed by reference	UDR with call to **mi_alloc( )**, **mi_dalloc( )**, or **mi_zalloc( )**	Routine manager (when it exits a UDR)

The current memory duration is initialized to this default memory duration. The default memory duration is PER_ROUTINE. For more information, see "Managing the Memory Duration" on page 14-21.

PER_COMMAND Memory Duration: A PER_COMMAND memory pool is associated with each SQL command. An *SQL command* is a subquery, which is a separate SQL statement initiated as part of the current SQL statement. The most common kind of subquery is a SELECT statement in the WHERE clause of a SELECT.

When a C UDR allocates PER_COMMAND memory, this memory is available to all routine instances that execute in the same SQL command. For example, the following SELECT statement contains two SQL commands:

```
SELECT a_func(x) FROM table1
   WHERE i <=
      (SELECT y FROM table2 WHERE a_func(x) <= 17);
```

The SELECT operation on **table1** is the main query and is one SQL command. The SELECT operation on **table2** is a subquery of the main query and is therefore a separate SQL command. All invocations of the **a_func( )** function in the main query can share any PER_COMMAND memory that this instance of **a_func( )** allocates; however, the invocations of **a_func( )** in the subquery have their own PER_COMMAND memory pool. These invocations would *not* share their memory pool with the invocations of **a_func( )** in the main query.

Other examples of subqueries follow:
- A SELECT statement after an IN, EXISTS, ALL, ANY, or SOME keyword in a WHERE clause:

```
SELECT stock_num, manu_code FROM stock
   WHERE NOT EXISTS
      (SELECT stock_num, manu_code FROM items
         WHERE stock.stock_num = items.stock_num
            AND stock.manu_code = items.manu_code);
```

- A SELECT statement after the table name in an INSERT statement:

```
INSERT INTO table1 (int_col)
   SELECT another_int_col FROM table2
      WHERE a_func(x) <= 17;
```

A separate SQL command is *not* created for simple WHERE clauses. For example, the following query contains only one SQL command:

```
SELECT a_func(x) FROM table1 WHERE a_func(y) > 6;
```

Both instances of **a_func( )** use the same PER_COMMAND memory pool for their PER_COMMAND allocations. Therefore, any PER_COMMAND memory that the **a_func( )** function allocates can be shared by *all* invocations of the **a_func( )** function in the select list *as well as* the invocations of **a_func( )** in the WHERE clause. If an SQL statement does *not* contain any subqueries, PER_COMMAND memory lasts for the duration of the SQL statement; that is, the PER_COMMAND and PER_STMT_EXEC memory durations are the same.

Tip: You can obtain the name of the SQL command that invoked the current UDR with the **mi_current_command_name( )** function.

The database server reclaims any PER_COMMAND memory in the memory context as follows:
- For an SQL statement with no subqueries, the database server deallocates PER_COMMAND memory when the SQL statement completes.
- For an SQL statement with one subquery, the database server deallocates PER_COMMAND memory as follows:

- For the main query, the database server frees PER_COMMAND memory after this main query completes.
- For a subquery, the database server frees PER_COMMAND memory each time the subquery finishes execution for one outer row of the main query, and after the main query completes.

The only exception to this rule is if this SQL statement is a cursor statement (DECLARE, OPEN, FETCH, UPDATE...WHERE CURRENT OF or DELETE...WHERE CURRENT OF, CLOSE), in which case the database server frees the PER_COMMAND memory when the cursor closes.

The PER_COMMAND memory duration is useful for accumulating calculations, in iterator functions, and for initialization of expensive resources. The most common way for UDR invocations within a routine instance to share information is to store this information in the user state of its **MI_FPARAM** structure. The routine manager allocates an **MI_FPARAM** structure for each C UDR instance. This **MI_FPARAM** structure has a PER_COMMAND memory duration. Therefore, to retain user state across a routine instance, a UDR can allocate PER_COMMAND memory and store its address in the **MI_FPARAM** structure. The UDR does *not* need to take special steps to preserve the address of this user-state memory. Each UDR invocation can use the **mi_fp_funcstate( )** function to obtain the address from the **MI_FPARAM** structure.

For example, if a UDR calculates a total, PER_ROUTINE memory would not be adequate to hold this total because the memory would be freed after a single routine invocation. PER_COMMAND memory would be available for the entire routine instance, regardless of the number of invocations involved. For more information on the user state in **MI_FPARAM**, see "Saving a User State" on page 9-8.

Several DataBlade API constructor functions allocate their DataBlade API data type structure with a PER_COMMAND memory duration. Table 14-3 shows the DataBlade API data type structures that have a memory duration of PER_COMMAND.

Table 14-3. DataBlade API Data Type Structures with a PER_COMMAND Memory Duration

DataBlade API Data Type Structure	DataBlade API Constructor Function	DataBlade API Destructor Function
Function descriptor (**MI_FUNC_DESC**)	**mi_cast_get( )**, **mi_func_desc_by_typeid( )**, **mi_routine_get( )**, **mi_routine_get_by_typeid( )**, **mi_td_cast_get( )**	**mi_routine_end( )**
MI_FPARAM structure	Routine manager (when it invokes a UDR)	Routine manager (when it exits a UDR)
MI_FPARAM structure (user-defined)	**mi_fparam_allocate( )**, **mi_fparam_copy( )**	**mi_fparam_free( )**

Switching the current memory duration before one of the constructor functions in Table 14-3 does *not* change the PER_COMMAND memory duration of the allocated DataBlade API data type structure. These data type structures are freed by their destructor function or when the current SQL command completes. To retain access to some of these DataBlade API data type structures after the command completes, you must save them at the per-session level.

Tip: The DataBlade API supports the ability to save information at a per-session level. This ability, however, is an advanced feature of the DataBlade API. For more information, see "Obtaining a Session-Duration Connection Descriptor" on page 7-13.

PER_STATEMENT Memory Duration: A PER_STATEMENT memory pool can be associated with each SQL statement, until execution of the statement is complete and for a prepared statement, until the statement terminates. The statement includes any SQL commands that the SQL statement initiates.

Important: The PER_STATEMENT memory duration is supported for compatibility with existing UDRs. In new code, you should use either the PER_STMT_EXEC or PER_STMT_PREP memory duration. These more precise memory durations replace PER_STATEMENT, which is deprecated.

When a C UDR allocates memory with the PER_STATEMENT memory duration, this memory is available to all routine instances that execute in the same SQL statement.

PER_STMT_EXEC Memory Duration: A PER_STMT_EXEC memory pool is associated with the execution of each SQL statement. A *statement* is the entire SQL statement plus any SQL commands that the SQL statement initiates, as follows:

- An SQL statement that the client application invokes
- An SQL statement that an SPL routine invokes
- An SQL statement that one of the following DataBlade API statement-execution functions executes:
  - **mi_exec( )**
  - **mi_exec_prepared_statement( )**
  - **mi_open_prepared_statement( )**

When a C UDR allocates memory with the PER_STMT_EXEC memory duration, this memory is available to all routine instances that execute in the same SQL statement. For example, suppose that the following SELECT statement invokes the **a_func2( )** user-defined function:

```
SELECT a_func2(x) FROM table1 WHERE y > 7;
```

Suppose also that the **a_func2( )** function calls **mi_exec( )** to execute a SELECT that also invokes **a_func2( )**, as follows:

```
mi_integer a_func2(arg)
   mi_integer arg;
{
   ...
   mi_exec(
      "select a_func2(y) from table2 where b_func(y) > 7;", ...)
```

The SELECT query in the call to **mi_exec( )** is a separate SQL command from the main SELECT query. All invocations of the **a_func2( )** function in the **mi_exec( )** SELECT statement can share any PER_STMT_EXEC memory that this instance of **a_func2( )** allocates. They can also share any PER_STMT_EXEC memory that the **b_func( )** function (in the WHERE clause) allocates.

The invocations of **a_func2( )** in the SELECT on **table1** have their own PER_STMT_EXEC memory pool. They would *not* share it with invocations of **a_func2( )** in the **mi_exec( )** call.

The database server reclaims any PER_STMT_EXEC memory in the current memory context as follows:

- If the SQL statement does not contain any subqueries, the statement consists of a single SQL command. The database server deallocates PER_STMT_EXEC and PER_COMMAND memory at the same time.
- If the SQL statement contains one or more subqueries, the statement consists of several SQL commands, one for the main query and one for each subquery. The PER_STMT_EXEC memory remains allocated until all SQL commands and UDRs complete.

At the completion of execution of a statement, the database server does *not* reclaim any memory in the memory context with a duration higher than PER_STMT_EXEC. The database server reclaims any PER_STMT_EXEC memory when the SQL statement completes execution, as follows:

- For a noncursor statement, the database server deallocates PER_STMT_EXEC memory as soon as the statement status is returned to the client application.

  This memory is actually freed on entry to the *next* execution of an SQL statement. After the last (or only) execution of the SQL statement, the database server deallocates the PER_STMT_EXEC memory after sending the status of the SQL statement to the client application. If a statement completes before the status is returned, the database server schedules the memory for release but does not free it until the return value is sent to the client application.

- For a cursor statement, the database server deallocates PER_STMT_EXEC memory as soon as the statement status of close cursor is returned to the client application.

  This memory is actually freed on entry to the *next* open of the cursor. After the last (or only) open of the cursor, the database server deallocates the memory after sending the status of the closed cursor to the client application.

Examples of Using PER_STMT_EXEC Memory Duration: For example, suppose the **a_func( )** user-defined function allocates PER_STMT_EXEC memory. The code fragment in Figure 14-4 shows a UDR that calls **a_func( )** in a noncursor statement that executes twice.

```
mi_integer udr_with_prepared_stmt( )
{
   ...
   stmt3 = mi_prepare(conn,
      "insert into tab3 values (a_func(87));", NULL);

   /* 1st execution of prepared INSERT */
   mi_exec_prepared_statement(stmt3, ...);

   /* Code that needs to access PER_STMT_EXEC memory is here */
   ...

   /* 2nd execution of prepared INSERT */
   mi_exec_prepared_statement(stmt3, ...);
   ...
   return stat;
}
```

Figure 14-4. PER_STMT_EXEC Memory in a Noncursor Statement

PER_STMT_EXEC memory that **a_func( )** allocates in the first call to **mi_exec_prepared_statement( )** is released just before the *second* execution of the prepared INSERT statement begins. Any code after the first

mi_exec_prepared_statement( ) call that needs to access this memory can do so. The PER_STMT_EXEC memory that **a_func( )** allocates in the second call to **mi_exec_prepared_statement( )** remains allocated until the database server returns to the client application the status of the SQL statement that has called the **udr_with_prepared_stmt( )** UDR.

The code fragment in Figure 14-5 shows use of **a_func( )** in a cursor statement.

```
mi_integer get_orders(start_with_cust, end_with_cust)
   mi_integer start_with_cust;
   mi_integer end_with_cust;
{
   mi_string *cmd =
       "select order_num, a_func(order_num) from orders \
        where customer_num = ?;";
   MI_STATEMENT *stmt;
   mi_integer i;
...
if ( (stmt = mi_prepare(conn, cmd, NULL)) == NULL )
   mi_db_error_raise(NULL, MI_EXCEPTION,
       "mi_prepare( ) failed");

if ( start_with_cust > end_with_cust )
   mi_db_error_raise(NULL, MI_EXCEPTION,
      "Arguments invalid.");

for ( i = start_with_cust; i <= end_with_cust; i++)
   {
   values[0] = i;
   types[0] = "integer";
   lengths[0] = 0;
   nulls[0] = MI_FALSE;

   /* Open the read-only cursor to hold the query rows */
   if ( mi_open_prepared_statement(stmt, MI_SEND_READ,
         MI_TRUE, 1, values, lengths, nulls, types,
         "cust_select", retlen, rettypes)
         != MI_OK )
      mi_db_error_raise(NULL, MI_EXCEPTION,
         "mi_open_prepared_statement( ) failed");
```

Figure 14-5. PER_STMT_EXEC Memory in a Cursor Statement (Part 1 of 2)

```
    /* Fetch the retrieved rows into the cursor */
    if ( mi_fetch_statement(stmt, MI_CURSOR_NEXT, 0, 3)
            != MI_OK )
        mi_db_error_raise(NULL, MI_EXCEPTION,
            "mi_fetch_statement( ) failed");

    if ( mi_get_result(conn) != MI_ROWS )
        mi_db_error_raise(NULL, MI_EXCEPTION,
    "mi_get_result( ) failed or found non-query statement");

    /* Retrieve the query rows from the cursor */
    if ( !(get_data(conn)) )
        mi_db_error_raise(NULL, MI_EXCEPTION,
            "get_data( ) failed");

    /* Close the cursor */
    if ( mi_close_statement(stmt) == MI_ERROR )
        mi_db_error_raise(NULL, MI_EXCEPTION,
            "mi_close_statement( ) failed");

    /* Code that needs to access PER_STMT_EXEC memory is here. */
    ...

    } /* end for */

/* Release resources */
if ( mi_drop_prepared_statement(stmt) == MI_ERROR )
    mi_db_error_raise(NULL, MI_EXCEPTION,
        "mi_drop_prepared_statement( ) failed");
if ( mi_close(conn) == MI_ERROR )
    mi_db_error_raise(NULL, MI_EXCEPTION,
        "mi_close( ) failed");
}
```

Figure 14-5. PER_STMT_EXEC Memory in a Cursor Statement (Part 2 of 2)

PER_STMT_EXEC memory that **a_func( )** allocated is released just before the cursor is reopened. Therefore, any code after the **mi_close_statement( )** function that needs to access this memory can do so. However, once the cursor is reopened, code can no longer access this same PER_STMT_EXEC memory. The PER_STMT_EXEC memory that **a_func( )** allocates in the *previous* (or only) open of the cursor remains allocated until the database server returns to the client application the status of the SQL statement that has called the **get_orders( )** UDR.

Uses of PER_STMT_EXEC Memory Duration: The PER_STMT_EXEC memory duration is useful for communications between UDRs, parallel execution, user-defined aggregates, and named memory, and for memory allocations within an end-of-statement callback (if you have information to pass to the callback).

Important: Any memory with a duration higher than PER_COMMAND could have multiple threads access it. Consider whether you need to handle concurrency issues for any PER_STMT_EXEC memory you allocate. For more information, see "Handling Concurrency Issues" on page 14-27.

Several DataBlade API constructor functions allocate their DataBlade API data type structure with a PER_STMT_EXEC memory duration. Table 14-4 lists DataBlade API data type structures that have a memory duration of PER_STMT_EXEC.

Table 14-4. DataBlade API Data Type Structures with a PER_STMT_EXEC Memory Duration

DataBlade API Data Type Structure	DataBlade API Constructor Function	DataBlade API Destructor Function
Connection descriptor (**MI_CONNECTION**)	**mi_open( )**	**mi_close( )**
Save-set structure (**MI_SAVE_SET**)	**mi_save_set_create( )**	**mi_save_set_destroy( )**

Switching the current memory duration before one of the constructor functions in Table 14-4 does *not* change the PER_STMT_EXEC memory duration of the allocated DataBlade API structure. These data type structures are freed by their destructor function or when execution of the current SQL statement completes. To retain access to some of these DataBlade API data type structures after the statement completes, you must save them at the per-session level.

Tip: The DataBlade API supports the ability to save information at a per-session level. This ability, however, is an advanced feature of the DataBlade API. For more information, see "Obtaining a Session-Duration Connection Descriptor" on page 7-13.

PER_STMT_PREP Memory Duration: A PER_STMT_PREP memory pool is associated with each prepared SQL statement. A *prepared statement* is an SQL statement that is parsed and ready for execution. The following table summarizes ways to create and drop a prepared statement.

Method	To Create a Prepared Statement	To Drop a Prepared Statement
Client application (SQL)	PREPARE	FREE
C UDR (DataBlade API)	**mi_prepare( )**	**mi_drop_prepared_statement( )**

When a C UDR allocates PER_STMT_PREP memory, this memory is available to all routine instances that execute before the current prepared statement is dropped. Unlike PER_STMT_EXEC memory, PER_STMT_PREP memory does *not* get freed upon re-execution of the prepared statement; that is, it remains allocated if the cursor is closed and reopened. For example, in Figure 14-5 on page 14-11, any PER_STMT_PREP memory that **a_func( )** allocated is not released when the cursor is reopened. Therefore, any code that needs to access this memory once the cursor is reopened can do so. The PER_STMT_PREP memory that **a_func( )** allocates remains allocated until the **mi_drop_prepared_statement( )** drops the **stmt** prepared statement.

When the prepared SQL statement is dropped, the database server reclaims any PER_STMT_PREP memory in the memory context. It does *not* reclaim any memory in the memory context with a duration higher than PER_STMT_PREP.

No DataBlade API constructor function allocates its data type structure with a memory duration of PER_STMT_PREP.

Advanced Memory Durations

The DataBlade API memory-management functions also support several *advanced* memory durations, which Table 14-5 shows.

Table 14-5. Advanced Memory Durations

Advanced Memory Duration	Memory-Duration Constant	Description
For the duration of the current transaction	PER_TRANSACTION	The database server frees the memory after the current transaction ends (commit or rollback).
For the duration of the current session	PER_SESSION	The database server frees memory at the end of the current session.
For the duration of the database server execution	PER_SYSTEM	The database server frees memory when it is brought down.

Warning: The memory durations in Table 14-5 are advanced and can adversely affect your UDR if you use them incorrectly. Use them only when no regular DataBlade API memory duration can perform the task you need.

As with the public memory-duration constants, the advanced memory-duration constants in Table 14-5 are of type **MI_MEMORY_DURATION**. However, these constants are declared in the **minmdur.h** header file, not the **memdur.h** header file. The **minmmem.h** header file automatically includes the **minmdur.h** header file. The **mi.h** header file, however, does *not* automatically include **minmmem.h**. To access advanced memory durations, you must include **minmmem.h** in any DataBlade API routine that uses these memory durations.

Important: Any memory with a duration higher than PER_COMMAND could have multiple threads access it. Therefore, consider whether you need to handle concurrency issues for any PER_TRANSACTION, PER_SESSION, or PER_SYSTEM memory you allocate. For more information, see "Handling Concurrency Issues" on page 14-27.

PER_TRANSACTION Memory Duration: A PER_TRANSACTION memory pool can be associated with either of the following:

- Each client transaction

  If the UDR makes a PER_TRANSACTION allocation during a client transaction, the database server uses memory from the PER_TRANSACTION memory pool. The way that a transaction begins and ends depends on whether the database is ANSI-compliant and whether it uses logging. (For more information, see "Transaction Management" on page 12-7.)

- A cursor started in a transaction

  Statements within a cursor are considered a type of transaction. If the UDR makes a PER_TRANSACTION allocation within a cursor, the database server allocates memory from a special PER_CURSOR memory pool, which lasts from the open to the close of the cursor.

  The PER_CURSOR memory duration is for *internal use only*. However, you might see information about the PER_CURSOR memory pool in the output of **onstat -g mem**. The database server creates a PER_CURSOR memory pool for each cursor in a transaction.

When a C UDR allocates PER_TRANSACTION memory, this memory is available to all routine instances that execute before the current transaction closes. The database server reclaims any PER_TRANSACTION shared memory in the memory context in either of the following situations:

- When the current transaction ends (with commit or rollback)

- If SQL statements execute in an explicit transaction, PER_TRANSACTION memory remains allocated until *all* statements in the transaction complete.
- If each SQL statement is a separate transaction, the database server deallocates PER_TRANSACTION and PER_STMT_EXEC memory at the same time.

  If a hold cursor is open when the transaction ends, the database server does not deallocate PER_TRANSACTION memory. However, it does deallocate PER_TRANSACTION memory whenever a hold cursor closes.

- When the cursor closes

  If the UDR allocated PER_TRANSACTION memory within a cursor, the database server reclaims this memory when the cursor closes.

Tip: An EXECUTE PROCEDURE statement does not create an implicit transaction. If EXECUTE PROCEDURE is not already part of an explicit transaction, the UDR that it calls can use a BEGIN WORK and COMMIT WORK (or ROLLBACK WORK) to specify a transaction. For more information, see "Transaction Management" on page 12-7.

At this time, the database server does *not* reclaim any memory in the memory context with a duration higher than PER_TRANSACTION.

The PER_TRANSACTION memory duration is useful for the following tasks:
- Cooperating UDRs in user-defined access methods (created with the IBM Informix VTI and VII interfaces)
- Committing and rolling back external resources (such as files and smart large objects)
- Allocating memory within an end-of-transaction callback (if you have information to pass to the callback)
- Allocating data type structures that need to persist during an implicit or explicit transaction

Allocate PER_TRANSACTION memory as named memory because this memory requires locking. To access it, a C UDR must know the name of the memory and it must be within the scope of the transaction. Such a UDR can explicitly free this memory with the **mi_named_free( )** function. However, consider PER_TRANSACTION memory as permanent to the current transaction. For more information, see "Managing Named Memory" on page 14-24.

No DataBlade API constructor function allocates its data type structure with a memory duration of PER_TRANSACTION.

PER_SESSION Memory Duration: A PER_SESSION memory pool is associated with each session. A *session* begins when a client connects to the database server, and it ends when the connection terminates. When a C UDR allocates PER_SESSION memory, this memory is available to all routine instances that execute before the current session ends. When the current session ends, the database server reclaims any PER_SESSION shared memory in the memory context. It does *not* reclaim any memory in the memory context with a duration higher than PER_SESSION.

The PER_SESSION memory duration is useful for the following tasks:
- External-resource management
- Session initialization

- Allocating memory within an end-of-session callback (if you have information to pass to the callback)
- Using cursors defined as hold cursors (hold cursors can span transactions)
- Caching expensive information between transactions for the life of the session or information that pertains to the session connection

Allocate PER_SESSION memory as named memory because this memory requires locking. To access it, a C UDR must know the name of the memory and it must be within the scope of the session. Such a UDR can explicitly free this memory with the **mi_named_free( )** function. However, consider PER_SESSION memory as permanent to the session. For more information, see "Managing Named Memory" on page 14-24.

Several DataBlade API constructor functions allocate their DataBlade API data type structures with a PER_SESSION memory duration. Table 14-6 shows the DataBlade API data type structures that have a memory duration of PER_SESSION.

Table 14-6. DataBlade API Data Type Structures with a PER_SESSION Memory Duration

DataBlade API Data Type Structure	DataBlade API Constructor Function	DataBlade API Destructor Function
Session-duration connection descriptor (**MI_CONNECTION**)	**mi_get_session_connection( )**	End of session
Session-duration function descriptor (**MI_FUNC_DESC**)	**mi_cast_get( )**, **mi_func_desc_by_typeid( )**, **mi_routine_get( )**, **mi_routine_get_by_typeid( )**, **mi_td_cast_get( )** (when these functions receive a session-duration connection descriptor as an argument)	End of session
File descriptor	**mi_file_open( )**	**mi_file_close( )**
Transient smart large object	**mi_lo_copy( )**, **mi_lo_create( )**, **mi_lo_expand( )**, **mi_lo_from_file( )**, **mi_lo_from_string( )** (but do *not* insert the LO handle into a column of the database)	**mi_lo_release( )**, **mi_lo_delete_immediate( )**

Switching the current memory duration before one of the constructor functions in Table 14-6 does *not* change the PER_SESSION memory duration of the allocated DataBlade API structure. These data type structures are freed by their destructor function or when the current session ends.

PER_SYSTEM Memory Duration: A PER_SYSTEM memory pool is associated with the database server instance. A *database server instance* begins when the **oninit** utility (or its equivalent) initializes the database server, and it ends when the database server is brought down. When a C UDR allocates PER_SYSTEM memory, this memory is available to all routine instances that execute before the database server instance is shut down. As the database server shuts down, it frees any PER_SYSTEM shared memory.

The PER_SYSTEM memory duration is useful for system-wide caching and resource initialization. Allocate PER_SYSTEM memory as named memory because this memory requires locking. To access it, a C UDR must know the name of the memory. The UDR can explicitly free this memory with the **mi_named_free( )** function. However, consider PER_SYSTEM memory as permanent to the database server. For more information, see "Managing Named Memory" on page 14-24.

Warning: The PER_SYSTEM memory duration takes up memory that the database server might use for other tasks. Restrict your use of memory with the PER_SYSTEM memory duration. For most uses, memory can be successfully allocated with shorter memory durations.

No DataBlade API constructor function allocates its data type structure with a memory duration of PER_SYSTEM.

Memory-Duration Considerations

When a UDR needs to allocate memory dynamically, it must take the following actions:

- Choose an appropriate memory duration for which to allocate the memory
- Save the address of the memory so that all UDRs that need to use the memory can access it

Choosing Memory Duration: When the UDR allocates memory, it must ensure that this memory has a appropriate memory duration. Choose a memory duration on the basis of which UDR instances need to share the information stored in the memory. Make sure you choose a memory duration that is appropriate to the use of the allocated memory. An inappropriate memory duration can cause the following problems:

- If you allocate memory with a duration that is too small, expect to see assertion failures in the message log file.

  For example, if you allocate PER_ROUTINE memory and store its address in the **MI_FPARAM** structure (which has a PER_COMMAND duration), the memory is freed after one invocation of the UDR, causing the address in the **MI_FPARAM** to be no longer valid.

- If you allocate memory with a duration that is too large, you might see memory leaks as your UDR executes.

  Memory leakage can occur when you allocate memory that has a higher duration than the structure that holds its address. For more information, see "Monitoring Shared Memory" on page 14-33.

Whenever possible, use the following public memory-management features of the DataBlade API:

- Public memory-management functions

mi_alloc( )	mi_zalloc(	mi_switch_mem_duration( )
mi_dalloc( )	mi_free( )	mi_realloc( )

These public functions are appropriate for a UDR that executes in the context of just one SQL statement. The current memory duration, which these functions use, is a useful way to ensure that all allocations occur with the same duration. For more information, see "Managing the Memory Duration" on page 14-21.

- Public memory durations

PER_ROUTINE	PER_STMT_EXEC
PER_COMMAND	PER_STMT_PREP

For more information, see "Public Memory Durations" on page 14-5. Advanced memory durations are necessary only in certain situations.

Warning: Keep track of the scope and memory duration of the memory that you allocate with the DataBlade API memory-management functions. Incorrect memory duration can create serious memory corruption.

Saving the Memory Address: In addition to ensuring that the allocated memory has an appropriate memory duration, you must ensure that the UDR can obtain the address of this memory when it needs to access the information within the memory. For example, if you allocate PER_COMMAND memory within a UDR but only store its address in a local variable, this address is deallocated when the UDR completes.

Important: The deallocation of the memory address but not the associated memory is one of the most common causes of memory leaks. Make sure that the duration of the memory address is compatible with that of its memory.

The following table summarizes common ways to save a memory address.

Memory Duration	Where To Store Memory Address	Scope of Address
PER_ROUTINE	Does not need to be handled because the memory and its address are only valid within a single routine invocation.	Current invocation of UDR
PER_COMMAND	Store the memory address in the user state of the **MI_FPARAM** structure.	All invocations of the UDR within the current SQL command
	You can take special steps (such as named memory) to store the memory address so that it can be accessed by other UDRs: • Named memory • Session-duration connection	All UDRs that know the name of the named-memory block All UDRs that have access to the session-duration connection descriptor
PER_STMT_EXEC	If the SQL statement does *not* contain any subqueries, you can store the memory address in the user state of the **MI_FPARAM** structure.	All invocations of the UDR within the current SQL statement
	If the SQL statement contains subqueries, you must take special steps (such as named memory) to store the memory address so that it can be accessed by other instances of the same UDR or by other UDRs.	

Managing Shared Memory

The following kinds of C functions can make allocations from the database server shared memory:

- C UDRs

  A C UDR has access to the following types of shared memory for dynamic allocations: user memory and named memory.

- DataBlade API constructor functions

  A constructor function allocates its DataBlade API data type structure in user memory. The constructor can allocate a particular data type structure with a specified memory duration (Table 14-2 through Table 14-4) or the current memory duration (Table 14-6 on page 14-16).

The DataBlade API provides the memory-management functions in Table 14-7 for the dynamic allocation of database server shared memory.

Table 14-7. Memory-Management Functions of the DataBlade API

Type of Shared Memory	Description	DataBlade API Functions
User memory	Memory that is accessible by its *address only*. User memory can be allocated in the current memory duration or have a specified memory duration.	**mi_alloc( )**, **mi_dalloc( )**, **mi_realloc( )**, **mi_zalloc( )**, **mi_switch_mem_duration( )**, **mi_free( )**
Named memory	Memory that has a name assigned and is *accessible by its address or its name*. Named memory has a specified memory duration.	**mi_named_alloc( )**, **mi_named_zalloc( )**, **mi_named_get( )**, **mi_named_free( )**
		mi_lock_memory( ), **mi_try_lock_memory( )**, **mi_unlock_memory( )**

Important: Named memory is an advanced feature that can adversely affect your UDR if you use it incorrectly. Use it only when no regular DataBlade API feature can perform the task you need done.

Tip: A client LIBMI application can also use DataBlade API memory-management functions to perform dynamic allocations. However, these memory-management functions allocate memory from the client process, not from the shared memory of the database server. Therefore, memory that these functions allocate from within a client LIBMI application does not have a memory duration associated with it. For more information, see "Managing Memory in Client LIBMI Applications" on page A-1.

These DataBlade API memory-management functions in Table 14-7 on page 14-19 work correctly with the transaction management and memory reclamation of the database server. In particular, they provide the following advantages:

- These functions allocate the memory from shared memory so that all VPs can access it.
- The database server automatically reclaims memory allocated with these functions.

These functions establish a memory duration for the memory they allocate. When this memory duration expires, the database server automatically marks the memory for reclamation. For more information, see "Choosing the Memory Duration" on page 14-4.

Managing User Memory

A C UDR allocates *user memory* from the database server shared memory. It is accessed by address. The DataBlade API provides memory-management functions to allocate user memory dynamically. These functions return a pointer to the address of the allocated memory and subsequent operations are performed on that pointer. Table 14-8 shows the memory-management functions that the DataBlade API provides for memory operations on user memory.

Table 14-8. DataBlade API User-Memory-Management Functions

User-Memory Task	DataBlade API Function
Allocating user memory	**mi_alloc( )**, **mi_dalloc( )**, **mi_realloc( )**, **mi_zalloc( )**
Changing the size of an existing memory block	**mi_realloc( )**
Changing current memory duration	**mi_switch_mem_duration( )**
Deallocating user memory	**mi_free( )**

Tip: The DataBlade API memory-management functions execute in client LIBMI applications as well as C UDRs. For DataBlade API modules that you design to run in both client LIBMI applications and UDRs, use these memory-management functions. For information on the behavior of these memory-management functions in a client LIBMI application, see Appendix A, "Writing a Client LIBMI Application," on page A-1.

The following table summarizes the memory operations for user memory.

Memory Duration	Memory Operation	Function Name
Current memory duration	Constructor	**mi_alloc( )**, **mi_dalloc( )**, **mi_zalloc( )**
	Destructor	**mi_free( )**

Allocating User Memory (Server)

To handle dynamic memory allocation of user memory, use one of the following DataBlade API memory-management functions.

Memory-Allocation Task	DataBlade API Function
To allocate user memory with the *current* memory duration	**mi_alloc( )**
To allocate user memory with a *specified* memory duration	**mi_dalloc( )**
To allocate user memory with the *current* memory duration that is *initialized with zeros*	**mi_zalloc( )**

These user-memory-allocation functions allocate memory from the shared memory of the database server at a particular memory duration.

Tip: The DataBlade API library also provides memory-management functions to manage named memory. These named-memory-management functions are advanced functions. Use them only when user-memory-management functions cannot perform the task you need done. For more information, see "Managing Named Memory" on page 14-24.

Managing the Memory Duration

The *current memory duration* is the memory duration that applies when the **mi_alloc( )** or **mi_zalloc( )** function allocates memory. These functions do not specify a memory duration for their allocation. Instead, they use the current memory duration for the memory they allocate. By default, the current memory duration is the default memory duration. The default memory duration is PER_ROUTINE; that is, the database server marks the memory for reclamation when the UDR completes. Therefore, the **mi_alloc( )** and **mi_zalloc( )** functions allocate memory with a duration of PER_ROUTINE by default.

Subsequent sections provide the following information:
- The DataBlade API structures allocated in the current memory duration
- How to change the current memory duration

Using the Current Memory Duration: Many of the DataBlade API constructor functions assign the current memory duration to the DataBlade API data type structures that they allocate. Table 14-9 shows the DataBlade API data type structures that are allocated with the current memory duration.

Table 14-9. DataBlade API Data Type Structures with the Current Memory Duration

DataBlade API Data Type Structure	DataBlade API Constructor Function	DataBlade API Destructor Function
Collection descriptor (**MI_COLL_DESC**)	**mi_collection_open( )**, **mi_collection_open_with_options( )**	**mi_collection_close( )**
Collection (**MI_COLLECTION**)	**mi_collection_copy( )**, **mi_collection_create( )**, **mi_streamread_collection( )**	**mi_collection_free( )**
Error descriptor (**MI_ERROR_DESC**)	**mi_error_desc_copy( )**	**mi_error_desc_destroy( )**
LO handle (**MI_LO_HANDLE**)	**mi_get_lo_handle( )**, **mi_lo_copy( )**, **mi_lo_create( )**, **mi_lo_expand( )**, **mi_lo_from_file( )**, **mi_lo_from_string( )**	**mi_lo_delete_immediate( )**, **mi_lo_release( )**
LO-specification structure (**MI_LO_SPEC**)	**mi_lo_spec_init( )**	**mi_lo_spec_free( )**
LO-status structure (**MI_LO_STAT**)	**mi_lo_stat( )**	**mi_lo_stat_free( )**
MI_LO_LIST	**mi_lo_lolist_create( )**	None
Row descriptor (**MI_ROW_DESC**)	**mi_row_desc_create( )**	**mi_row_desc_free( )**
Row structure (**MI_ROW**)	**mi_row_create( )**, **mi_streamread_row( )**	**mi_row_free( )**
Stream descriptor (**MI_STREAM**)	**mi_stream_init( )**, **mi_stream_open_fio( )**, **mi_stream_open_mi_lvarchar( )**, **mi_stream_open_str( )**	**mi_stream_close( )**
User memory	**mi_alloc( )**, **mi_zalloc( )**	**mi_free( )**
Varying-length structure (**mi_lvarchar, mi_sendrecv, mi_impexp, mi_impexpbin**)	**mi_new_var( )**, **mi_streamread_lvarchar( )**, **mi_string_to_lvarchar( )**, **mi_var_copy( )**	**mi_var_free( )**

To change the memory duration of a DataBlade API data type structure, call the **mi_switch_mem_duration( )** function with the desired duration *before* the DataBlade API function call that allocates the object. For more information, see "Changing the Memory Duration" on page 14-22.

Important: All the DataBlade API functions in Table 14-9 allocate structures with the current memory duration. If you switch the current memory duration, you affect not only explicit allocations you make with **mi_alloc( )** or **mi_zalloc( )** but the memory allocations that all these DataBlade API constructor functions do as well.

Changing the Memory Duration: The PER_ROUTINE memory duration is the default protection on a region of user memory. You can change the memory duration of user memory to another duration in either of the following ways:

- Use **mi_dalloc( )** instead of **mi_alloc( )** to allocate memory.

  The **mi_dalloc( )** function works in the same way as **mi_alloc( )** but provides the option of specifying the memory duration of the memory to allocate. This function does *not* switch the current memory duration.

- Call **mi_switch_mem_duration( )** before you call **mi_alloc( )**.

  The **mi_switch_mem_duration( )** function switches the current memory duration. All user-memory allocations by subsequent calls to **mi_alloc( )** or **mi_zalloc( )** have the new current memory duration.

You can use regular or advanced memory durations for user memory. For most memory allocations in a C UDR, use one of the regular memory-duration constants (PER_ROUTINE, PER_COMMAND, PER_STMT_EXEC, or PER_STMT_PREP).

Important: Use an advanced memory duration for user memory only if a regular memory duration cannot safely perform the task you need done. These advanced memory durations have long duration times and can increase the possibility of memory leakage.

Changing the current memory duration with **mi_switch_mem_duration( )** has an effect on the memory durations of all DataBlade API data type structures that Table 14-9 on page 14-21 lists. It does *not* have an effect on the memory duration of DataBlade API data type structures allocated at the PER_COMMAND (Table 14-3 on page 14-8) and PER_STMT_EXEC (Table 14-4 on page 14-13) durations or at the advanced memory durations (Table 14-6 on page 14-16).

The **mi_switch_mem_duration( )** function returns the previous memory duration. You can use this return value to restore memory duration after performing some allocations at a different duration. The following code fragment temporarily changes the current memory duration from PER_ROUTINE (the default) to PER_COMMAND:

```
/* Switch current memory duration to PER_COMMAND and save
 * old current memory duration in 'old_mem_dur'
 */
old_mem_dur = mi_switch_mem_duration(PER_COMMAND);

/* Perform allocations for a new current memory duration */
buffer = (char *)mi_alloc(BUFF_SIZE);
new_lvarch = mi_new_var(BUFF_SIZE-1);
save_set = mi_save_set_create(conn);

/* Restore old current memory duration */
(void)mi_switch_mem_duration(old_mem_dur);
```

In the preceding code fragment, the PER_COMMAND memory duration is in effect for the allocation of user memory that the call to **mi_alloc( )** makes. Because the **mi_new_var( )** function allocates a new varying-length structure in the current memory duration, this call to **mi_new_var( )** allocates the varying-length structure with a PER_COMMAND duration. However, the **mi_save_set_create( )** function does *not* allocate its save-set structure at the current memory duration. Therefore, the call to **mi_save_set_create( )** still allocates its save-set structure with the PER_STMT_EXEC duration. The second call to **mi_switch_mem_duration( )** restores the current memory duration to PER_ROUTINE.

Deallocating User Memory

The database server automatically reclaims the user memory that **mi_alloc( )**, **mi_dalloc( )**, and **mi_zalloc( )** allocate. The memory duration of the user memory determines when the database server marks the memory for deallocation.

Tip: If a DataBlade API function allocates memory to hold its return result, the database server automatically frees this memory when its duration expires unless otherwise noted in its description in function descriptions.

User memory remains valid until one of the following events occurs:
- The **mi_free( )** function frees the memory.
- The memory duration expires.
- The **mi_close( )** function closes the current connection (except memory with a PER_SYSTEM duration).
- A database server exception is raised.

A C UDR is not allowed to cache information from the database across transaction boundaries. Because the state of the database might change entirely when the current transaction commits, any cached information might be invalid. Therefore, UDRs must reinitialize any database state that they require when the next transaction begins. To enforce the policy of no caching across transactions, the database server automatically reclaims memory marked for deallocation at transaction boundaries. In addition, the database server reclaims memory when specified memory durations expire, usually when a UDR allocates and returns a value.

To conserve resources, use the **mi_free( )** function to explicitly deallocate the user memory once your DataBlade API module no longer needs it. The **mi_free( )** function is the destructor function for user memory.

Keep the following restrictions in mind about memory deallocation:
- Do *not* free user memory that you allocate for the return value of a UDR.
- Do *not* free memory until you are finished accessing the memory.
- Do *not* use **mi_free( )** to deallocate memory that you have not explicitly allocated.
- Do *not* use **mi_free( )** for data type structures that other DataBlade API constructor functions allocate.
- Do *not* attempt to free user memory after its memory duration has expired.
- Reuse memory whenever possible. Do not repeat calls to allocation functions if you can reuse the memory for another task.

Managing Named Memory

Named memory is memory allocated from the database server shared memory, just as user memory. You can, however, assign a name to a block of named memory and then access this memory block by name and memory duration. The database server stores the name and its corresponding address internally. By contrast, user memory is always accessed by its address.

The disadvantage of user memory is that the database server deallocates PER_COMMAND, PER_STMT_EXEC, PER_STMT_PREP, and PER_STATEMENT memory after the command or statement completes. Because a UDR might execute many times for a particular SQL statement (once for each row processed), you might want to retain the memory pointer across all calls to the same UDR.

Tip: The DataBlade API named-memory-management functions execute only in C UDRs. Do not use these memory-management functions in client LIBMI applications. For DataBlade API modules that you design to run in both client LIBMI applications and UDRs, use the user-memory-management functions. For information on memory management in a client LIBMI application, see Appendix A, "Writing a Client LIBMI Application," on page A-1.

To save memory across invocations of a UDR, you can perform one of the following tasks:
- You can save the memory pointer as part of the user state in the **MI_FPARAM** structure that is associated with the UDR.

  For more information, see "Saving a User State" on page 9-8.
- You can allocate *named memory* with an appropriate memory duration.

  The advantage of named memory is that it is global within the memory duration it was allocated. Therefore, it can be accessed by many UDRs that execute in the context of many queries, or even by more than one session. Named memory is useful as global memory for caching data across UDRs or for sharing memory between UDRs executing in the context of many SQL statements.

Possible uses for named memory follow:
- Semi-static lookup information that can be shared among UDRs or sessions
- Caching function descriptors at the session level for repeated calls to **mi_routine_exec( )**
- Index methods that need to store global information for an index scan across a fragmented index

The DataBlade API provides the memory-management functions to dynamically allocate named memory in a C UDR. These functions return a name of the named-memory block and subsequent operations are performed on that name and memory duration. Table 14-10 shows the memory-management functions that the DataBlade API provides for memory operations on named memory.

Table 14-10. DataBlade API Named-Memory-Management Functions

Named-Memory Task	DataBlade API Functions
Allocating named memory	**mi_named_alloc( )**, **mi_named_zalloc( )**
Obtaining an allocated named-memory block	**mi_named_get( )**
Controlling concurrency	**mi_lock_memory( )**, **mi_try_lock_memory( )**, **mi_unlock_memory( )**

Table 14-10. DataBlade API Named-Memory-Management Functions (continued)

Named-Memory Task	DataBlade API Functions
Deallocating named memory	**mi_named_free( )**

Warning: These advanced memory-management functions can adversely affect your UDR if you use them incorrectly. Use them only when the regular DataBlade API user-memory-management functions cannot perform the task you need done.

The **minmprot.h** header file defines the functions and data type structures of the named-memory-management functions. The **minmmem.h** header file automatically includes the **minmprot.h** header file. However, the **mi.h** header file does *not* automatically includes **minmmem.h**. To access the named-memory-management functions, you must include **minmmem.h** in any DataBlade API routine that calls these functions.

Tip: Each of the named-memory functions in Table 14-10 have tracepoints in them that generate output when the trace level is greater than zero (0). The output consists of the function name and the arguments passed to it. For more information on tracepoints, see "Using Tracing" on page 12-28.

The following table summarizes the memory operations for named memory.

Memory Duration	Memory Operation	Function Name
Specified memory duration	Constructor	**mi_named_alloc( )**, **mi_named_zalloc( )**
	Destructor	**mi_named_free( )**

Allocating Named Memory

To handle dynamic memory allocation of named memory, use one of the following DataBlade API memory-management functions.

Memory-Allocation Task	DataBlade API Function
To allocate named memory with a specified memory duration	**mi_named_alloc( )**
To allocate named memory with a specified memory duration that is initialized with zeros	**mi_named_zalloc( )**

These named-memory-allocation functions allocate a block of named memory of a specified size and a specified memory duration. You can use both regular and advanced memory durations for named memory. Usually, named-memory allocations are at memory durations longer than PER_COMMAND. With PER_ROUTINE and PER_COMMAND memory durations, you can use the **MI_FPARAM** structure to store information. You must ensure that the memory duration is sufficiently long that all UDRs that need to access it can access it.

Tip: These named-memory-management functions do *not* use the current memory duration.

If the allocation of the named-memory block is successful, these functions store a pointer to the allocated block in their *mem_ptr* argument. The UDR that allocated the named-memory block can access this named memory through this address.

However, this address is deallocated when the routine invocation completes. Other UDRs must use the block name to access the named-memory block. For more information, see "Obtaining a Block of Allocated Named Memory" on page 14-26.

These DataBlade API memory-management functions work correctly with the transaction management and memory reclamation of the database server. They provide the same advantages as the user-memory-management functions (see "Allocating User Memory (Server)" on page 14-20). In addition, they provide the advantage that the named-memory block can be accessed by a name, which facilitates access to the memory across UDRs.

Obtaining a Block of Allocated Named Memory

The benefit of named memory is that several UDRs can access it. Therefore, UDRs can cache data for sharing between UDRs executing in different contexts. To use named memory, UDRs that need to access it must take the following steps:

- One UDR needs to allocate the named-memory block with **mi_named_alloc( )** or **mi_named_zalloc( )**.

    This named-memory block must have a name that will be known to all UDRs that need to access the block. It must also have a memory duration that is sufficient for the required lifetime of the cached data. The UDR that allocates the named memory can access the data from the address that **mi_named_alloc( )** or **mi_named_zalloc( )** returns. However, once this UDR completes, the local copy of this address is deallocated.

- Any UDR that needs to access the data in the named-memory block can specify the name and memory duration of the memory block to **mi_named_get( )**.

    The **mi_named_get( )** function returns the address of the named-memory block. The UDR can use this address to access the desired data within the named memory.

For example, suppose a UDR named **initialize( )** allocates a named-memory block named **cache_blk** with the following **mi_named_alloc( )** call:

```
mi_integer *blk_ptr;
mi_integer status;
...
status = mi_named_alloc(sizeof(mi_integer), "cache_blk",
   PER_STMT_EXEC, &blk_ptr);

switch( status )
{
   case MI_ERROR:
      mi_db_error_raise(NULL, MI_EXCEPTION,
         "mi_named_alloc for cache_blk failed.");
      break;

   case MI_NAME_ALREADY_EXISTS:
      break;

   case MI_OK:
      *blk_ptr = 0;
      break;

   default:
      mi_db_error_raise(NULL, MI_EXCEPTION,
         "Invalid mi_named_alloc status for cache_blk");
}
```

If another UDR, for example, **some_task( )**, needs to access the integer in **cache_blk**, it can use the **mi_named_get( )** function, as the following code fragment shows:

```
mi_integer *blk_ptr;
mi_integer status;
...
status = mi_named_get("cache_blk", PER_STMT_EXEC, &blk_ptr);

switch( status )
{
   case MI_ERROR:
      mi_db_error_raise(NULL, MI_EXCEPTION,
         "mi_named_get for cache_blk failed.");
      break;

   case MI_NO_SUCH_NAME:
      /* maybe need to call mi_named_alloc( ) here */
      ...
      break;

   case MI_OK:
      if ( *blk_ptr > 0 )
         *blk_ptr++;
      break;

   default:
      mi_db_error_raise(NULL, MI_EXCEPTION,
         "Invalid mi_named_alloc status for cache_blk");
}
```

If **some_task( )** successfully obtains the address of the **cache_blk** named-memory block (**status** is MI_OK), it increments the cached integer.

Important: The preceding code fragment does not handle concurrency issues that result from multiple UDRs trying to access the **cache_blk** named memory at the same time. For more information, see "Handling Concurrency Issues" on page 14-27.

Handling Concurrency Issues

By default, the database server runs UDRs concurrently. A UDR that uses data it shares with other UDRs or with multiple instances of the same routine must implement concurrency control on its data. When the named-memory functions **mi_named_alloc( )**, **mi_named_zalloc( )**, and **mi_named_get( )** return the address of a named-memory block, they do *not* request a lock on this named memory. It is the responsibility of your UDRs or DataBlade to manage concurrency issues on the named-memory block.

The greater the memory duration that is associated with the named memory, the more likely that you must manage concurrency of that memory. If the named memory is never updated (it is read-only), there are no concurrency problems. However, if any UDR updates the named memory that has a duration of PER_COMMAND or greater, there are concurrency issues just like there are for any global variable that gets updated.

For example, suppose the function **myfunc( )** allocates a named-memory block named **named_mem1**. The memory duration of **named_mem1** determines possible concurrency issues when **myfunc( )** is called in the following query:

```
SELECT * FROM my_table
   WHERE myfunc(column1) = 1
   OR myfunc(column2) = 2
```

The following table shows possible concurrency issues of **named_mem1**.

Named-Memory Allocation	Concurrency Issues?
mi_named_alloc(2048, ″named_mem1″, *PER_COMMAND*, nmem1_ptr)	Yes Each invocation of **myfunc( )** in the query gets it own private instance of **named_mem1**, which expires when the UDR completes, but there might be multiple threads running in a subquery that share the same PER_COMMAND pool. If PER_COMMAND memory is cached in the **MI_FPARAM** user data, however, there are no concurrency issues because each thread has its own **MI_FPARAM** structure. Unless you need memory to be shared between threads, this is the preferable alternative for PER_COMMAND.
mi_named_alloc(2048, ″named_mem1″, *PER_SESSION*, nmem1_ptr)	Yes Each invocation of **myfunc( )** in the same query accesses the *same* **named_mem1**. This memory does not get deallocated until the session closes.
mi_named_alloc(2048, ″named_mem1″, *PER_SYSTEM*, nmem1_ptr)	Yes Every invocation of **myfunc( )** in every SQL statement accesses the *same* **named_mem1**. This memory does not get deallocated until the database server shuts down.

To handle concurrency problems, use the following DataBlade API memory-locking functions in UDRs that update named memory.

Memory-Locking Task	DataBlade API Memory-Locking Function
Request a lock on the specified named-memory block and wait for the lock to be obtained.	**mi_lock_memory( )**
Request a lock on the specified named-memory block and do *not* wait for the lock to be obtained.	**mi_try_lock_memory( )**
Unlock the specified named-memory block.	**mi_unlock_memory( )**

The following guidelines are recommended for handling concurrency problems:
- The safest approach, even for threads that only read named memory, is to lock the named-memory block.

  After the named-memory block is locked, you can guarantee that all accesses will obtain a consistent read.
- Keep the time that you lock a named-memory block as *short* as possible.

  The DataBlade API locking interface is intended to be used in a tightly coupled, fast section of code that protects a critical region during modification. Code should follow these steps:
  - Lock the named memory with **mi_lock_memory( )** or **mi_try_lock_memory( )**.
  - Perform the modification or consistent read.
  - Immediately unlock the memory with **mi_unlock_memory( )**.
- If you need to hold locks for a long time, do it with SQL in a client application.

Tip: Locking of named memory (with **mi_lock_memory( )** and **mi_try_lock_memory( )**) uses its own locking mechanism to keep track of named-memory locks. It does not consume database locks.

Suppose you have a user-defined structure named **MyInfo** with the following declaration:

```
typedef struct
{
   mi_integer is_initialized;
   ... other members here....
} MyInfo;
```

The following sample code allocates a named-memory block named **MyInfo_memory** for the **MyInfo** structure. It then locks a critical section of code before updating the **is_initialized** integer in this named-memory block.

```
MyInfo *GetMyInfo( )
{
   mi_string *memname="MyInfo_memory",
             msgbuf[80];
   mi_integer status;
   MyInfo         *my_info = NULL;

/* Allocate the named-memory block. If it has already been
 * allocated, obtain a pointer to this block.
 */
   status = mi_named_zalloc(sizeof(MyInfo),
         memname, PER_SESSION, (void **)&myinfo);
   if( status == MI_NAME_ALREADY_EXISTS )
      status = mi_named_get(memname, PER_SESSION,
         (void **)&my_info);

   switch(status)
   {
      case MI_ERROR:
         mi_db_error_raise(NULL, MI_EXCEPTION,
   "GetMyInfo: mi_named_get or mi_named_zalloc failed.");
         return (MyInfo *)NULL;
         break;

      /* Have a pointer to the named_memory block. */
      case MI_OK:
         break;

      case MI_NO_SUCH_NAME:
         mi_db_error_raise(NULL, MI_EXCEPTION,
            "GetMyInfo: no name after good get");
         return (MyInfo *)NULL;
         break;

      default:
         sprintf(msgbuf,
         "GetMyInfo: mi_named memory case %d.", status);
         mi_db_error_raise(NULL, MI_EXCEPTION, msgbuf);
         return (MyInfo *)NULL;
         break;
   }

   /*
    * BEGIN CRITICAL SECTION.
    *
    * All access to the my_info structure is done
    * inside this lock-protected section of code.
    *
    * If two threads try to initialize information
    * at the same time, the second one blocks on
    * the mi_lock_memory call.
    *
    * A reader also blocks so that it gets a
    * consistent read if another thread is updating
```

```
 * that memory.
 */
status = mi_lock_memory(memname, PER_SESSION);
switch(status)
{
   case MI_ERROR:
      mi_db_error_raise(NULL, MI_EXCEPTION,
         "GetMyInfo: mi_lock_memory call failed.");
      return (MyInfo *)NULL;
      break;

   case MI_OK:
      break;

   case MI_NO_SUCH_NAME:
      mi_db_error_raise(NULL, MI_EXCEPTION,
         "mi_lock_memory got MI_NO_SUCH_NAME.");
      return (MyInfo *)NULL;
      break;

   default:
      sprintf(msgbuf,
         "GetMyInfo: mi_lock_memory case %d.",
         status);
      mi_db_error_raise(NULL, MI_EXCEPTION, msgbuf);
      return (MyInfo *)NULL;
      break;
}

/* The lock on the named-memory block has been
 * obtained.
 */

/* The mi_named_zalloc( ) call above zeroed out
 * the structure, like calloc( ). So if the is_initialized
 * flag is set to zero, named memory has not been
 * initialized yet.
 */
if (my_info->is_initialized == 0)
{
   /* In this block we populate the named-memory
    * structure. After initialization succeeds, set the
    * is_initialized flag.
    *
    * If any operation fails, MUST release the lock
    * before calling mi_db_error_raise( ):
    *
    *  if (whatever != MI_OK)
    *  {
    *     mi_unlock_memory(memname, PER_SESSION);
    *     mi_db_error_raise(NULL, MI_EXCEPTION,
    *           "operation X failed!");
    *     return (MyInfo *)NULL;
    *  }
    *
    */

   my_info->is_initialized = 1;

}  /* endif: MyInfo structure not initialized */
else
{
   /* Update or get a consistent read here. Again,
    * before any exception is raised with
    * mi_db_error_raise( ), the lock MUST be released.
    */
}
```

```
   /*
    * END CRITICAL SECTION.
    */
   mi_unlock_memory (memname, PER_SESSION);

   return my_info;
}
```

The preceding code fragment uses the **mi_lock_memory( )** function to obtain the
lock on the named memory. The following code fragment uses
mi_try_lock_memory( ) to try to get a lock on a named-memory block 10 times
before it gives up:

```
for ( lockstat=MI_LOCK_IS_BUSY, i=0;
      lockstat == MI_LOCK_IS_BUSY && i < 10;
      i++ )
{
   lockstat = mi_try_lock_memory(mem_name, PER_STMT_EXEC);
   switch( lockstat )
   {
      case MI_OK:
         break;

      case MI_LOCK_IS_BUSY:
         mi_yield( ); /* Yield the processor. */
         break;

      case MI_NO_SUCH_NAME:
         mi_db_error_raise(NULL, MI_EXCEPTION,
            "Invalid name of memory after good get");
         return MI_ERROR;
         break;

      case MI_ERROR:
         mi_db_error_raise(NULL, MI_EXCEPTION,
            "Lock request failed.");
         return MI_ERROR;
         break;

      default:
         mi_db_error_raise(NULL, MI_EXCEPTION,
            "Invalid status from mi_try_lock_memory( )");
         return MI_ERROR;
         break;
   }
}
/* Check the status after coming out of the loop. */
if( lockstat == MI_LOCK_IS_BUSY )
   {
      mi_db_error_raise(NULL, MI_EXCEPTION,
         "Could not get lock on named memory.");
      return MI_ERROR;
   }

/* Now have a locked named-memory block. Can perform a
 * read or update on the memory.
 */
...
mi_unlock_memory(mem_name, PER_STMT_EXEC);
```

Usually, the **mi_try_lock_memory( )** function is a better choice than
mi_lock_memory( ) for lock requests because **mi_try_lock_memory( )** does not
hang if the lock is busy.

The database server does *not* release any locks you acquire on named memory. You must ensure that your code uses the **mi_unlock_memory( )** function to release locks in the following cases:

- *Immediately* after you are done accessing the named memory
- *Before* you raise an exception with **mi_db_error_raise( )**
- *Before* you call another DataBlade API function that raises an exception internally (For more information, see "Handling Errors from DataBlade API Functions" on page 10-26.)
- *Before* the session ends
- *Before* the memory duration of the named memory expires
- *Before* you attempt to free the named memory

Warning: After you obtain a lock on a named-memory block, you must explicitly release it with the **mi_unlock_memory( )** function. Failure to release a lock before one of the previous conditions occurs can severely impact the operation of the database server.

Deallocating Named Memory

The database server automatically reclaims the named memory that **mi_named_alloc( )** and **mi_named_zalloc( )** allocate. The memory duration of the named memory determines when the database server marks the memory for deallocation. Named memory remains valid until either of the following events occurs:

- The **mi_named_free( )** function frees the memory.
- The memory duration expires.

To conserve resources, use the **mi_named_free( )** function to explicitly deallocate the named memory once your DataBlade API module no longer needs it. The **mi_named_free( )** function is the destructor function for named memory.

Keep the following restrictions in mind about memory deallocation of named memory:

- Do *not* free memory until you are finished accessing the memory.
- Do *not* free memory that is still locked.
- Do *not* use **mi_named_free( )** to deallocate memory that you have not explicitly allocated with **mi_named_alloc( )** or **mi_named_zalloc( )**.
- Do *not* attempt to free named memory after its memory duration has expired.
- Reuse memory whenever possible. Do not repeat calls to allocation functions if you can reuse the memory for another task.

The **mi_named_free( )** function cannot free a named-memory block that is currently locked by another owner. If a UDR with another owner has a lock on the requested memory block, **mi_named_free( )** marks the block as "deallocation pending" but does not actually free the memory. A subsequent call to **mi_named_get( )** would return the MI_NO_SUCH_NAME return value for this named-memory block. Once the UDR with another owner has explicitly unlocked the memory block with **mi_unlock_memory( )**, a "deallocation pending" memory block is automatically freed. A subsequent call to **mi_named_get( )** from this other UDR would return the MI_NO_SUCH_NAME return value for this named-memory block.

Monitoring Shared Memory

You can monitor use of memory that your UDR allocates (explicitly or implicitly) with the following options of the **onstat** utility:

- The **-g ses** option outputs session information.

  You can specify a particular session identifier after the **ses** keyword. If you do not include a session identifier, **onstat -g ses** outputs a one-line summary for each active session. Look for any session whose memory allocation or usage steadily increases.

- The **-g mem** option outputs statistics for a memory pool.

  Internally, the database server organizes memory allocations into memory pools by duration. You can specify a particular pool name after the **mem** keyword. If you do not include a pool name, **onstat -g mem** outputs a one-line summary for each memory pool. To detect memory leakage, look for any pool whose memory allocation or usage steadily increases.

Tip: You can use the **-r** option of **onstat** in conjunction with either of the preceding options to have **onstat** repeat the command every specified number of seconds.

Monitoring memory is useful for tracking down memory leakage. Memory leakage occurs when memory remains allocated after the structure that holds its address was deallocated. There is no way to access the memory once its address is gone. Therefore, the memory remains with no way to remove it.

Suppose that the **onstat -g ses** command produces the following sample output:

```
session                               #RSAM    total      used
id      user     tty     pid  hostname threads memory    memory
24      informix -        0   -        0       8192       5880
23      informix -        13453 bison  1       6701056    6608512
8       informix -        0   -        0       8192       5880
7       informix -        0   -        0       16384      14344
6       informix -        0   -        0       8192       5880
4       informix -        0   -        0       16384      14344
3       informix -        0   -        0       8192       5880
2       informix -        0   -        0       8192       5880
ps auxw | grep 13453
```

Suppose also that your DB–Access session is hooked to session 23 (the italicized line in the preceding sample output).

─────────────────────── **UNIX/Linux Only** ───────────────────────

You can determine the session identifier of the DB–Access session on UNIX or Linux with the following command:

```
ps auxw | grep 13453
```

─────────────────────── **End of UNIX/Linux Only** ───────────────────────

You can now obtain information about memory-pool usage with the **-g mem** option of **onstat**:

```
onstat –g mem
```

Suppose that the preceding **onstat** command generated the following sample output (some output lines are omitted for brevity):

```
Pool Summary:
name         class    addr
resident       R      a002018
res-buff       R      a118018
global         V      a18a018
mt             V      a18e018
smartblob      V      a192018
...
23             V      a3e0018
24             V      a3e2018
23.RTN.SAPI    V      a40c018
23.CMD.SAPI    V      a3ee018
...
```

Figure 14-6 shows the lines of the **onstat -g mem** output that indicate user memory allocations.
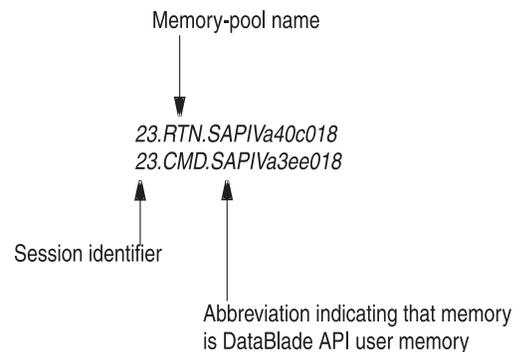
Figure 14-6. Memory Pools in onstat -g mem Output

Each memory duration has a separate memory pool. The three letters before "SAPI" identify each memory pool. The following table shows the memory-pool names for regular and advanced memory durations.

Memory-Pool Name	Associated Memory Pool
RTN	PER_ROUTINE
CMD	PER_COMMAND
STM	PER_STATEMENT (*deprecated duration*)
EXE	PER_STMT_EXEC
PRP	PER_STMT_PREP
TRX	PER_TRANSACTION (*advanced duration*)
UNK	PER_CURSOR (*advanced duration*)
SES	PER_SESSION (*advanced duration*)
SYS	PER_SYSTEM (*advanced duration*)

After you determine a specific session identifier or memory-pool name that exhibits a problem, you can find out which specific kind of memory is affected with the **-g ufr** option of **onstat**. The **-g ufr** option of **onstat** shows memory fragments by usage. For example, the following **onstat** command captures a snapshot every 30 seconds of memory pools that session 23 uses:

```
onstat -g ufr 23 -r 30
```

Sample output for the preceding command follows:

```
Memory usage for pool name 23:
size        memid
2152        log
2016        ostcb
2600        sqtcb
8472        gentcb
1664        osenv
6392648     sqscb
792         filetable
112         rdahead
120         overhead
96          scb
416         sapi
3296        fragman
18808       opentable
280         hashfiletab
10056       temprec
592         GenPg
224         ru
56          sort
94848       ralloc
```

In the preceding sample output, the main memory consumer is **sqscb**.

Any memory leakage from a DataBlade API function would show up in the **memid** column entry labelled **sapi**. For more information on the **onstat** command, see the *IBM Informix Administrator's Reference*.

Managing Stack Space

Session threads execute C UDRs and their thread-stack space is allocated from a common region in shared memory. The *thread stack* stores nonshared data for the UDRs and system routines that the thread executes. This stack contains everything that would normally be on the execution stack, including the following:

- Routine arguments, including the **MI_FPARAM** structure
- Local (stack) variables
- Function return values

Like all memory that UDRs use, stack segments can be overrun. The database server can only check for stack violations when the UDR yields. Therefore, you must ensure that you perform the following tasks within a UDR:

- Efficiently manage stack space usage in your UDR

  Limit usage of stack space in your UDR and ensure sufficient stack-space allocation when you register the UDR.
- Use the **mi_call( )** function for UDRs that potentially use unlimited recursion.

Managing Stack Usage

To avoid stack overflow, follow these design restrictions in your UDR:

- Do not use large automatic arrays.
- Avoid excessively deep calling sequences.
- Use **mi_call( )** within a UDR to manage recursive calls.

By default, when a thread of a virtual processor executes a UDR, the database server uses a thread-stack size that the STACKSIZE configuration parameter specifies (32 kilobytes, if STACKSIZE is not set). To determine how much stack space a UDR requires, monitor the UDR from the system prompt with the following **onstat** command:

```
onstat -g sts
```

The **-g sts** option prints the maximum and current stack usage per thread. For more information on the **onstat** utility and its **-g sts** option, see the *IBM Informix Administrator's Reference*.

You must ensure that your UDR has sufficient stack space for its execution. That is, the UDR must have enough stack space to hold all local variables of the routine. If you see errors in the message log file of the following format when you try to allocate a large block of memory, your stack space is being overrun:

```
Assert Failed: Condition Failed (Bad pool pointer 0xe2fe018),
in (mt_shm_free)

Assert Failed: Memory block header corruption detected in mt_shm_free
```

To determine if there is enough stack space for your UDR, use the **mi_stack_limit( )** function. This function checks if the space available on the stack exceeds the sum of the stack margin and the specified stack size.

To override the stack size for a particular UDR, use the STACK routine modifier of the CREATE FUNCTION or CREATE PROCEDURE statement when you register your UDR. For example, the following CREATE FUNCTION statement specifies a stack size of 64 kilobytes for the **func1( )** UDR:

```
CREATE FUNCTION func1(INTEGER)
   RETURNS INTEGER
   WITH (STACK=65536)
   EXTERNAL NAME '/usr/srv_routs/funcs.so'
   LANGUAGE C;
```

When the UDR completes, the database server allocates thread stacks for subsequent UDRs based on the STACKSIZE parameter (unless these subsequent UDRs have also specified the STACK routine modifier).

Increasing Stack Space

The DataBlade API provides the **mi_call( )** function to dynamically manage stack space. This function performs the following tasks:

- It checks the amount of unused stack space and allocates additional stack segments if necessary.
- It executes the specified UDR.

Use the **mi_call( )** function to increase stack space for recursive UDRs.

Keep in mind that **mi_call( )** does *not* know the size of the routine arguments. When **mi_call( )** creates a new stack and copies an argument onto this new stack, the function uses the size of the **MI_DATUM** data type for the argument. If the data type of the routine argument is larger than **MI_DATUM**, **mi_call( )** does *not* copy all the argument bytes.

For example, consider a UDR that includes an **mi_double** argument.

───────────────────── **UNIX/Linux Only** ─────────────────────

On UNIX or Linux, an **mi_double_precision** argument takes the space of two **long int** values. Therefore, the **mi_call( )** function pushes only half of the argument onto the new stack. Any arguments after the **mi_double_precision** might get

garbled, and the last one might be truncated.

When you design UDRs that require the use of **mi_call( )**, make sure you use the correct passing mechanism for the argument data type. Pass all data types larger than **MI_DATUM** by reference. Examples of large data types are floating-point types (such as **mi_real** and **mi_double_precision**) and data type structures.

The following example illustrates stack-space allocation with **mi_call( )**:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "mi.h"

mi_integer factorial(mi_integer value)
{
   mi_integer callstatus=0,
            retval=0;

   if ( value < 0 )
      return -1;
   else if ( value == 1 || value == 0 )
      return 1;
   else if ( value > 30 )
      mi_db_error_raise(NULL, MI_EXCEPTION,
   "factorial: input value too big for result.");

   callstatus = mi_call(&retval, factorial, 1, value-1);
   switch( callstatus )
      {
      case MI_TOOMANY:
         mi_db_error_raise(NULL, MI_EXCEPTION,
            "factorial: too many parameters.");

      case MI_CONTINUE:
         return (value * factorial(value-1));

      case MI_NOMEM:
         mi_db_error_raise(NULL, MI_EXCEPTION,
            "factorial: not enough memory");

      case MI_DONE:
         /* At the end of the factorial recursion, the
          * function still needs to calculate:
          *      value * factorial(value-1)
          */
         retval *= value;
         break;
      }

   return retval;
}
```

This code sample implements a factorial function. If the **mi_call( )** function determines that there is sufficient stack space, the code recursively calls the **handle_row( )** function to process the row value. The return value of the **mi_call( )** function indicates whether **mi_call( )** has allocated additional thread-stack memory, as follows.

mi_call( ) Return Value	Description	Action
MI_CONTINUE	The thread stack currently has room for another invocation of **factorial( )**.	The **mi_call( )** function does *not* need to allocate a new thread stack.
		The code fragment explicitly calls **factorial( )** on the *value*-1 value.
MI_DONE	The thread stack currently does *not* have room for another invocation of **factorial( )**.	The **mi_call( )** function allocates a new thread stack, copies the arguments onto this stack, and invokes the **factorial( )** function on the *value*-1 value, returning its value in **callstatus**.
		The code fragment does *not* need to explicitly call **factorial( )** on the *value*-1 value. The **mi_call( )** function did the work of invoking the routine; however, **mi_call( )** completed only the following portion of the calculation:
		`factorial(value-1)`
		To complete the factorial, the function needs to complete the following calculation:
		`value * factorial(value-1)`

The other **mi_call( )** return values (MI_NOMEM and MI_TOOMANY) indicate error conditions. For these return values, the function uses the **mi_db_error_raise( )** function to raise a database server exception and provide an error message.

The following CREATE FUNCTION registers the **factorial( )** function:

```
CREATE FUNCTION factorial (INTEGER)
   RETURNS INTEGER
   EXTERNAL NAME
      "$INFORMIXDIR/extend/misc/fact_ius.bld"
   LANGUAGE C;
```

The following EXECUTE FUNCTION invokes the **factorial( )** function:

```
EXECUTE FUNCTION factorial(5);
```

Chapter 15. Creating Special-Purpose UDRs

In This Chapter

This chapter describes how to write C user-defined routines (UDRs) with the following purposes.

Type of UDR	More Information
Cast function	page 15-2
Cost function	page 15-54
End-user routine	page 15-2
Iterator function	page 15-3
Negator function	page 15-60
Parallelizable UDR	page 15-61
Selectivity function	page 15-54

Writing an End-User Routine

A C UDR can implement an end-user routine. An *end-user routine* provides some additional functionality to the SQL end user. It is an SQL-invoked routine; that is, it is called directly from an SQL statement. An end-user routine can provide any task that is useful to SQL users. This user might be a database administrator or an SQL end user.

For more information on possible tasks of an end-user routine, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Writing a Cast Function

A *cast function* is a user-defined function that converts one data type (the source data type), to another data type (the target data type). A cast can be one of the following types:

- An *implicit* cast is a cast that the database server can invoke automatically when it encounters data types that cannot be compared with the system-defined casts.
- An *explicit* cast is a cast that you must specifically invoke, with either the CAST AS keywords or with the cast operator (::).

Tip: This section describes how to create a cast function that is written in C. For general information on how to create user-defined functions and casts, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

To register a C UDR as a cast function:

1. Use the CREATE FUNCTION statement to register the C UDR as a cast function.

   For more information, see "Registering a C UDR" on page 12-14.

2. Use the CREATE CAST statement to register the cast in the database.

   Casts are stored in the **syscasts** system catalog table. For more information on the syntax of the CREATE CAST statement, see the *IBM Informix Guide to SQL: Syntax*

The following lines register the C function, **a_to_b( )**, as an implicit cast from the **a** to **b** data type:

```
CREATE FUNCTION a_to_b(source a)
RETURNS b
EXTERNAL NAME '/usr/udrs/casts.so(a_to_b)'
LANGUAGE C;

CREATE CAST (a AS b WITH a_to_b);
```

These SQL statements assume that **a** and **b** are already registered as user-defined types. These statements only provide the ability to convert from type **a** to type **b**. To provide the ability to cast from the **b** to the **a** data type, you must create a second cast, as the following sample lines show:

```
CREATE FUNCTION b_to_a(source b)
RETURNS a
EXTERNAL NAME '/usr/udrs/casts.so(b_to_a)'
LANGUAGE C;

CREATE CAST (b AS a WITH b_to_a);
```

The following lines declare the C function, **a_to_b( )**, which accepts the **a** fixed-length opaque type as an argument and returns the **b** fixed-length opaque type:

```
b_t *a_to_b(source_type)
   a_t *source_type;
{
   b_t *target;

   target = (b_t *)mi_alloc(sizeof(b_t));

   /* Perform necessary conversions from a to b */

   return ( target );
}
```

Writing an Iterator Function

An *iterator function* is a user-defined function that returns to its calling SQL statement several times, each time returning a value. The database server gathers these returned values together in an *active set*. To access a value in the active set, you must obtain it from a database cursor. Therefore, an iterator function is a *cursor function* because it must be associated with a cursor when it is executed.

Tip: This section describes how to create an iterator function that is written in C. For general information on how to create user-defined functions, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

The database server might execute an iterator function many times. It groups these iterations into the iterator-status values and puts the *iterator status* for a given iteration in the **MI_FPARAM** structure. Within an iterator function, you examine the **MI_FPARAM** structure for an iterator status to determine which actions the iterator function must take.

Tip: The IBM Informix BladeSmith development tool, which is part of the Informix DataBlade Developers Kit, automatically generates C source code for an iterator function as well as the SQL statements to register the iterator function. For more information, see the *IBM Informix DataBlade Developers Kit User's Guide*.

To specify the different points at which the database server calls an iterator function, the iterator-status flag (of type **MI_SETREQUEST**) supports the constants in Table 15-1.

Table 15-1. Iterator-Status Constants for Calls to an Iterator Function

When Is the Iterator Function Called?	What Does the Iterator Function Do?	Iterator-Status Constant in MI_FPARAM
The first time that the iterator function is called	Initializes the iterations	SET_INIT
Once for each item in the active set	Returns one item of the active set	SET_RETONE
After the last item of the active set is returned	Releases iteration resources	SET_END

To implement an iterator function with a C user-defined function:

1. Declare the iterator function so that its return value has a data type that is compatible with one of the items in the active set.

   For example, to return an active set of integer values, declare the iterator function to return the **mi_integer** data type.

2. Include an **MI_FPARAM** structure as the *last* parameter of the C declaration of the iterator function.

   The **MI_FPARAM** structure holds the iterator status, the iterator-completion flag, and the user-state pointer.

3. Within the iterator function, obtain the iterator status from the **MI_FPARAM** structure with the **mi_fp_request( )** function.

   This function returns the iterator-status constant (SET_INIT, SET_RETONE, or SET_END) that the database server has set for the distinct groups of iterations of the iterator function.

4. For each of the iterator-status values, take the appropriate actions within the iterator function.

Iterator-Status Value	More Information
SET_INIT	"Initializing the Iterations" on page 15-6
SET_RETONE	"Returning One Active-Set Item" on page 15-8
SET_END	"Releasing Iteration Resources" on page 15-9

5. Register the iterator function as a user-defined function with the ITERATOR routine modifier in the CREATE FUNCTION statement.

   Omit the **MI_FPARAM** parameter from the parameter list when you register the iterator function. For more information, see "Registering a C UDR" on page 12-14.

The Fibonacci series is a list of numbers for which each value is the sum of the previous two. For example, the Fibonacci series up to a stop value of 20 is as follows:

```
0, 1, 1, 2, 3, 5, 8, 13
```

Figure 15-1 is an implementation of an iterator function named **fibGen( )**. This function builds an active set that contains a Fibonacci series of numbers up to a specified stop value.

```
typedef struct fibState1 /* function-state structure */
{
    mi_integer fib_prec1; /* second most recent number in series */
    mi_integer fib_prec2; /* most recent number in series */
    mi_integer fib_ncomputed; /* number computed */
    mi_integer fib_endval; /* stop value */
}fibState;

/* fibGen( ): an iterator function to return the Fibonacci series.
 * This function takes a stop value as a parameter and returns the
 * Fibonacci series up to this stop value.
 *
 * Three states of iterator status:
 *    SET_INIT : Allocate the defined user-state structure.
 *    SET_RETONE : Compute the next number in the series.
 *    SET_END : Free the user-allocated user-state structure.
 */
mi_integer fibgen(stop_val,fparam)
    mi_integer stop_val;
    MI_FPARAM  *fparam;
{
    mi_integer next;
    fibState *fibstate = NULL;

    switch( mi_fp_request(fparam) )
        {
        case SET_INIT:
            next = fibGen_init(stop_val, fparam);
            break;

        case SET_RETONE:
            next = fibGen_retone(fparam);
            fibstate = (fibState *)mi_fp_funcstate(fparam);
            if ( next > fibstate->fib_endval )
                {
                mi_fp_setisdone(fparam, 1);
                next = 0; /* return value ignored */
                }
            break;

        case SET_END:
            next = fibGen_end(fparam);
            break;
        }
    return (next);
}
```

Figure 15-1. The fibGen( ) Iterator Function

The database server calls this **fibGen( )** iterator function at the following execution points:

• *Once*, to initialize the calculation of the Fibonacci series of numbers

  At this point, the database server has set the iterator status to SET_INIT and **fibGen( )** calls the **fibGen_init( )** function (see Figure 15-2 on page 15-7).

• *Repeatedly*, to calculate each number in the series until a number exceeds the stop value

  As long as the number is less than the stop value, the database server sets the iterator status to SET_RETONE and **fibGen( )** calls the **fibGen_retone( )** function (see Figure 15-3 on page 15-8).

• *Once*, to deallocate resources that the iterator function uses

  At this point, the database server has set the iterator status to SET_END and **fibGen( )** calls the **fibGen_end( )** function (see Figure 15-4 on page 15-9).

Tip: For end users to be able to use an iterator function within an SQL statement, you must register the iterator function with the ITERATOR routine modifier

of the CREATE FUNCTION statement. For more information, see "Calling an Iterator Function from an SQL Statement" on page 15-9.

When the iterator function reaches the last item, call the **mi_fp_setisdone( )** function to set the *iterator-completion flag* of the **MI_FPARAM** structure to one (1). This flag indicates to the database server that it has reached the end condition for the iterator function. The database server no longer needs to continue calling the iterator function with the SET_RETONE iterator-status value. Instead, it calls the iterator function one more time, with the SET_END status value.

Important: Make sure that you include a call to the **mi_fp_setisdone( )** function within your iterator function that sets the iterator-completion flag to one (*1*). Without this call, the database server never reaches an end condition for the iteration, which causes it to iterate the function in an infinite loop.

In Figure 15-1 on page 15-5, the **fibGen( )** iterator function determines if it has reached an end condition after it calls **fibGen_retone( )**. It makes this determination as follows:

- If this number is greater than the user-specified stop value (in the **fib_endval** field of the user-state information), the end condition was reached.

  The **fibGen( )** function calls the **mi_fp_setisdone( )** function to set the iterator-completion flag to 1. The function then exits with a return value of zero (0). However, this last return value of 0 is *not* returned as part of the active set. The database server calls the next iteration of **fibGen( )** with an iterator-status value of SET_END.

- If the next Fibonacci number is less than or equal to the stop value, the end condition was *not* reached.

  The function returns this next number in the Fibonacci series to the active set. The database server calls the next iteration of **fibGen( )** with an iterator-status value of SET_RETONE.

Initializing the Iterations

The first time that the database server calls the iterator function, the database server passes it an **MI_FPARAM** structure with the iterator status set to SET_INIT and the user-state pointer set to NULL. When the iterator function obtains this iterator-status value, it can perform the following initialization tasks for the iterator function:

- Allocate memory for the structure in which you save the user-state information with a DataBlade API memory-management function such as **mi_dalloc( )**

  Make sure that this memory has a memory duration of PER_COMMAND so that the user state remains for the duration of *all* iterations of the iterator function. If the memory has the default PER_ROUTINE memory duration, the database server automatically deallocates it after only *one* iteration of the iterator function. For more information, see "Choosing the Memory Duration" on page 14-4.

- Save the user-state pointer in the **MI_FPARAM** structure with the **mi_fp_funcstate( )** function

  Each subsequent call to the iterator function uses the same **MI_FPARAM** structure, so each iteration can reuse the cached user-state memory. The iterator function must save enough information to return values one at a time on demand. For more information, see "Saving a User State" on page 9-8.

You can perform these initialization tasks directly in the iterator function or you can declare a separate *iterator-initialization function*, which the iterator function calls when it receives the SET_INIT iterator-status value. Declare the iterator-initialization function to return the same data type as the main iterator function. However, the database server ignores the return value of this function; it does *not* put this return value in the active set.

Figure 15-2 implements an iterator-initialization function, call **fibGen_init( )**, which the **fibGen( )** iterator function (Figure 15-1 on page 15-5) calls when it obtains the SET_INIT iterator-status value.

```
mi_integer fibGen_init(stop_val, fparam)
   mi_integer stop_val;
   MI_FPARAM *fparam;
{
   fibState    *fibstate;

/* Allocate the user-state structure, fibState. This user-state
 * structure is allocated with PER_COMMAND duration to hold the memory
 * until the end of all iterations of the iterator function.
 */
   fibstate = (fibState *)mi_dalloc(sizeof(fibState), PER_COMMAND);

/* Save a pointer to the user-state structure in the MI_FPARAM structure.
 */
   mi_fp_setfuncstate(fparam, (void *)fibstate);

/* Set return value of function to NULL for either of the following:
 *      - no argument passed into function (MI_FPARAM has a NULL argument)
 *      - stop value is < 0
 */
   if ( mi_fp_argisnull(fparam, 0) || stop_val < 0 )
      {
      mi_fp_setreturnisnull(fparam,0,1);
      return;
      }

/* Set the first two numbers of the series: 0 and 1. Set the stop value
 * field in the user-state structure (stop_val) to the stop value passed
 * to the function.
 */
   if ( stop_val < 1 )
      {
      fibstate->fib_prec1 = 0;
      fibstate->fib_prec2 = 1;
      fibstate->fib_ncomputed = 1;
      fibstate->fib_endval = stop_val;
      }
   else
      {
      fibstate->fib_prec1 = 0;
      fibstate->fib_prec2 = 1;
      fibstate->fib_ncomputed = 0;
      fibstate->fib_endval = stop_val;
      }
   return (0); /* return value is ignored */
}
```

Figure 15-2. The fibGen_init( ) Initialization Function

The **fibGen_init( )** function returns an **mi_integer** value (0) because the main iterator function, **fibGen( )**, returns an active set of **mi_integer** values. However, the database server does *not* return this value as part of the active set. Once **fibGen_init( )** completes, the database server calls the next iteration of **fibGen( )** with an iterator-status value of SET_RETONE to return the first item of the active set.

Returning One Active-Set Item

When the iterator status is SET_RETONE, the iterator function can return one item
of the active set. When the iterator function obtains this iterator-status value, it can
perform the iteration tasks needed to generate one item of the active set.

You can perform these iterator tasks directly in the iterator function or you can
declare a separate *iterator-value-return function*, which the iterator function calls
when it receives the SET_RETONE iterator-status value. Declare the
iterator-value-return function to return the same data type as the main iterator
function. The database server puts the return value of this function in the active
set.

Figure 15-3 implements an iterator-value-return function, named **fibGen_retone( )**,
that the **fibGen( )** iterator function (Figure 15-1 on page 15-5) calls each time it
obtains the SET_RETONE iterator status.

```
/* fibGen_retone( ):
 *    Generates the next number in the series. Compares it with the stop
 *    value to check if the end condition is met. Then performs following
 *    calculations:
 *        num1 = num2;
 *        num2 = next number in the series.
 */
mi_integer fibGen_retone(fparam)
   MI_FPARAM *fparam;
{
   fibState   *fibstate;
   mi_integer next;

   fibstate = (fibState *)mi_fp_funcstate(fparam);

   /* Generate next Fibonacci number */
   if ( fibstate->fib_ncomputed < 2 )
      return((fibstate->fib_ncomputed++ == 0) ? 0 : 1);

   /* Update user state for next iteration */
   next = fibstate->fib_prec1 + fibstate->fib_prec2;

   if ( next == 0 )
      {
      fibstate->fib_prec1 = 0;
      fibstate->fib_prec2 = 1;
      }
   else
      {
      fibstate->fib_prec1 = fibstate->fib_prec2;
      fibstate->fib_prec2 = next;
      }
   return (next);
```

Figure 15-3. The fibGen_retone( ) Value-Return Function

Each item in the active set that the **fibGen( )** function generates is one call to the
fibGen_retone( ) function. The **fibGen_retone( )** function returns one number of
the Fibonacci series. It uses the **mi_fp_funcstate( )** function to obtain the user-state
pointer from the **MI_FPARAM** structure. This user-state pointer points to a
fibstate structure (which the **fibGen_init( )** function in Figure 15-2 on page 15-7
allocated).

From the information in the **fibstate** structure, the **fibGen_retone( )** function
determines the next Fibonacci number and stores it in the **next** variable. The
function then updates the **fibstate** structure for the next iteration of **fibGen( )**.
Finally, the function returns one item of the active set: the value of **next**.

Releasing Iteration Resources

Once the **mi_fp_setisdone( )** function sets the iterator-completion flag to 1, the database server calls the iterator function one last time with the iterator-status value in the **MI_FPARAM** structure set to SET_END. When the iterator function obtains this iterator-status value, it can perform any tasks needed to deallocate resources that the iterator function has allocated.

Important: Free only resources that you have allocated. Do not attempt to free resources that the database server has allocated (such as the **MI_FPARAM** structure).

You can perform these deallocation tasks directly in the iterator function or you can declare a separate *iterator-end function*, which the iterator function calls when it receives the SET_END iterator-status value. Declare the iterator-end function to return the same data type as the main iterator function. However, the database server ignores the return value of this function; it does *not* put this return value in the active set.

Figure 15-4 implements an iterator-end function, named **fibGen_end( )**, that the **fibGen( )** iterator function (see Figure 15-1 on page 15-5) calls when it obtains the SET_END iterator-status value.

```
mi_integer fibGen_end(fparam)
   MI_FPARAM *fparam;
{
   fibState   *fibstate;

   fibstate = (fibState *)mi_fp_funcstate(fparam);
   mi_free(fibstate);

   return (0); /* return value is ignored */
}
```

Figure 15-4. The fibGen_end( ) Iterator-End Function

The **fibGen_end( )** function uses the **mi_fp_funcstate( )** function to obtain the user-state pointer from the **MI_FPARAM** structure. It then calls the **mi_free( )** function to free the resources in the **fibstate** state structure, which the **fibGen_init( )** function (see Figure 15-2 on page 15-7) has allocated. The **fibGen_end( )** function returns an **mi_integer** value (0) because the main iterator function, **fibGen( )**, returns an active set of **mi_integer** values.

Calling an Iterator Function from an SQL Statement

For end users to be able to use an iterator function within an SQL statement, take the following actions:
* Register the iterator function with the ITERATOR routine modifier.
* Associate the iterator function with a cursor to execute the function.

Registering the Iterator Function

Register the iterator function with the CREATE FUNCTION statement. The CREATE FUNCTION must include the ITERATOR routine modifier to tell the database server that it must call the function until the iterator-completion flag is set to 1.

Tip: If your iterator function calls other functions (such as an iteration-initialization function, iterator-value-return function, or iterator-end function)

to implement its iterations, you do not have to register these other functions with the CREATE FUNCTION statement.

The following CREATE FUNCTION statement registers the **fibGen( )** iterator function, which Figure 15-1 on page 15-5 defines, in the database:

```
CREATE FUNCTION fibgen(arg INTEGER)
RETURNING INTEGER
WITH (ITERATOR)
EXTERNAL NAME "$USERFUNCDIR/fib.so"
LANGUAGE C;
```

This statement assumes that the object code for the **fibGen( )** function resides in the UNIX or Linux **fib.so** shared-object file in the directory that the **USERFUNCDIR** environment variable specifies. It also assumes that **USERFUNCDIR** was set in the server environment.

For more information on how to register a user-defined function, see "Registering a C UDR" on page 12-14.

Executing the Iterator Function

After you register an iterator function (and assign it the appropriate privileges), users who have the Execute privilege can execute it. However, because an iterator function returns an active set of items, you must associate the function with a cursor. The cursor holds the active set, which the application can then access one at a time.

Each iteration of the iterator function returns one item of the active set. To execute an iterator function, you must associate it with a cursor. This EXECUTE FUNCTION statement generates an active set that contains the following Fibonacci values:

```
0, 1, 1, 2, 3, 5, 8
```

To obtain these values from within an application, you must associate the EXECUTE FUNCTION statement with a cursor to execute the function.

Once you register the **fibGen( )** function, you can execute the following SQL statement from an interactive database utility (such as DB–Access):

```
EXECUTE FUNCTION  fibgen(10);
```

From within a DataBlade API module, execute **fibGen( )** with the **mi_exec_prepared_statement( )** function, as follows:

```
MI_CONNECTION *conn;
mi_string *cmd = "EXECUTE FUNCTION fibgen(10);";
MI_STATEMENT *stmt;
mi_integer error, col_val;
MI_ROW *row;
...
/* Prepare the EXECUTE FUNCTION to execute fibGen( ) */
stmt = mi_prepare(conn, cmd, NULL);

/* Open the cursor to allocate the active set */
if ( mi_open_prepared_statement(stmt, MI_SEND_READ, 1, 0,
     NULL, NULL, NULL, NULL, NULL, 0, NULL) == MI_OK )

   /* Initialize the fetch direction */
   if ( mi_fetch_statement(stmt, MI_CURSOR_NEXT, 0, 0)
        == MI_OK )

     if ( mi_get_result(conn) == MI_ROWS )
```

```
        {
        /* Fetch the items of the active set.
         * Process each item of active set until
         * last item is found
         */
        while ( (row = mi_next_row(conn, &error)) != NULL )
            {
            colval = NULL;

            /* Obtain one number of Fibonacci series */
            mi_value(row, 0, (mi_integer)&col_val,
                  sizeof(mi_integer));

                /* Process current Fibonacci number */
                ...

            } /* end while */
        } /* end if mi_get_result */

/* Close the cursor to deallocate active set */
mi_close_statement(stmt);

/* Release statement descriptor */
mi_drop_prepared_statement(stmt);
```

For more information on how to use **mi_exec_prepared_statement( )**, see
"Executing Prepared SQL Statements" on page 8-11.

Writing an Aggregate Function

An *aggregate* is a function that returns one value for a group of queried rows. The
aggregate function performs one iteration for each of the queried rows. For each
row, an aggregate iteration receives one column value (called the *aggregate
argument*). The value that the aggregate returns to the SQL statement is called the
aggregate result. For example, the following query calls the built-in SUM aggregate
to determine the total cost of item numbers in order 1002:

```
SELECT SUM(total_price) FROM items
WHERE order_num = 1002;
```

For this invocation of the SUM aggregate, each value of the **total_price** column
that is passed into SUM is one aggregate argument. The total sum of all **total_price**
values, which SUM returns to the SELECT, is the aggregate result.

The database server supports extensions of aggregates in the following ways:

• Extensions of built-in aggregates
• User-defined aggregates

You can write C user-defined functions to implement these aggregate extensions.
For an overview of how to create aggregate functions and how to write them in an
SPL routine, see the chapter on this topic in the *IBM Informix User-Defined Routines
and Data Types Developer's Guide*. The following sections provide information
specific to the creation of aggregate functions as C user-defined functions.

Tip: The IBM Informix BladeSmith development tool automatically generates C
source code for a user-defined aggregate as well as the SQL statements to
register the aggregate function. For more information, see the *IBM Informix
DataBlade Developers Kit User's Guide*.

Extending a Built-In Aggregate

This section explains how to extend a built-in aggregate by overloading an operator function.

To extend a built-in aggregate function with a C user-defined function:

1. Determine the appropriate operator function that you must overload to implement the desired built-in aggregate function.

   For a list of built-in aggregate functions and the associated operator functions to overload, see the chapter on aggregate functions in the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

2. Write the C UDR that implements the required operator function for the data type that you want the aggregate to handle.

   To extend built-in aggregates so that they handle user-defined data types, write an operator function that accepts the user-defined data type as an argument. Compile the C UDR and link it into a shared-object file.

3. Register the overloaded operator function with the CREATE FUNCTION statement.

4. Use the newly extended aggregate on the data.

Suppose you want to use the SUM aggregate on complex numbers, which are stored in the following user-defined data type: a named row type named **complexnum_t**. Figure 15-5 shows the CREATE ROW TYPE statement that registers the **complexnum_t** named row type.

```
CREATE ROW TYPE complexnum_t
   (real_part SMALLFLOAT,
   imaginary_part SMALLFLOAT);
```

Figure 15-5. A Named Row Type to Hold a Complex Number

The following sections show how to extend the SUM aggregate on the **complexnum_t** named row type.

Choosing the Operator Function

The SUM built-in aggregate function uses the plus operator (+), which the **plus( )** user-defined function implements. The database server provides implementations of the **plus( )** function over the built-in data types. Therefore, the SUM aggregate function works over built-in data types. To have the SUM aggregate operate on the **complexnum_t** row type, you implement a **plus( )** function that handles this named row type; that is, it adds the two parts of the complex number and returns a complex number with the summed parts.

The following C function, **complex_plus( )**, defines such a **plus( )** function:

```
MI_ROW *complex_plus(arg1, arg2)
   MI_ROW *arg1;
   MI_ROW *arg2;
```

Writing the Operator Function

The code segment shows the implementation of the **complex_plus( )** function, which implements a **plus( )** function for the **complexnum_t** data type:

```
MI_ROW *complex_plus(arg1, arg2, fparam)
   MI_ROW *arg1;
   MI_ROW *arg2;
   MI_FPARAM *fparam;
```

```
{
   mi_real real_zero, imag_zero = 0.0;
   mi_real *real_value1, *real_value2;
   mi_integer real_len1, real_len2;
   mi_real *imag_value1, *imag_value2;
   mi_integer imag_len1, imag_len2;

   mi_real sum_real, sum_imag;

   MI_CONNECTION *conn;
   MI_TYPEID *type_id;
   MI_ROW_DESC *row_desc;

   mi_integer i;
   MI_ROW *ret_row;
   MI_DATUM values[2];
   mi_boolean nulls[2] = {MI_FALSE, MI_FALSE};

   for ( i=0; i<=1; i++ )
      {
      if ( mi_fp_argisnull(fparam, i) == MI_TRUE )
         {
         /* Put initialized complex number into 'values'
          * array
          */
         values[0] = (MI_DATUM)&real_zero;
         values[1] = (MI_DATUM)&imag_zero;

         /* Generate initialized row type for arg1 */
         conn = mi_open(NULL, NULL, NULL);
         type_id = mi_typestring_to_id(conn,
            "complexnum_t");
         row_desc = mi_row_desc_create(type_id);

         ret_row = mi_row_create(conn, row_desc, values,
            nulls);

         if ( i == 0 )
            arg1 = ret_row;
         else
            arg2 = ret_row;
         }
      }

   /* Extract values from arg1 row type */
   mi_value_by_name(arg1, "real_part",
      (MI_DATUM *)&real_value1, &real_len1);
   mi_value_by_name(arg1, "imaginary_part",
      (MI_DATUM *)&imag_value1, &imag_len1);

   /* Extract values from arg2 row type */
   mi_value_by_name(arg2, "real_part",
      (MI_DATUM *)&real_value2, &real_len2);
   mi_value_by_name(arg2, "imaginary_part",
      (MI_DATUM *)&imag_value2, &imag_len2);

   /* Sum the complex numbers */
   sum_real = *real_value1 + *real_value2;
   sum_imag = *imag_value1 + *imag_value2;

   /* Put sum into 'values' array */
   values[0] = (MI_DATUM)&sum_real;
   values[1] = (MI_DATUM)&sum_imag;

   /* Generate return row type */
   conn = mi_open(NULL, NULL, NULL);
   type_id = mi_typestring_to_id(conn, "complexnum_t");
```

```
row_desc = mi_row_desc_create(type_id);
ret_row = mi_row_create(conn, row_desc, values, nulls);

return (ret_row);
}
```

This version of the **plus( )** function performs the following tasks:

- Checks for a NULL-valued state (which indicates the first invocation of the ITER function) to initialize the state
- Checks for a NULL-valued aggregate argument to initialize a NULL the argument
- Accepts the two **complexnum_t** arguments as row-type pointers (**MI_ROW ***) and uses the **mi_value_by_name( )** function to extract the individual fields of the row type from the arguments
- Calculates the sum of the complex numbers by adding the real values together and the imaginary values together
- Creates an **MI_ROW** type to hold the **complexnum_t** value with the final sum: the **mi_row_desc_create( )** function creates a row descriptor for the **complexnum_t** data type and the **mi_row_create( )** function populates the associated row structure with the final sum values
- Returns a pointer to the MI_ROW structure (because a row type must be returned by reference, not by value)

Once the **complex_plus( )** function is written, you compile it and put it into a shared-object file. Suppose that **complex_plus( )** is compiled and linked into a shared-object module named **sqsum**.

UNIX/Linux Only

On UNIX or Linux, the executable code for the **complex_plus( )** operator function would be in a shared library named **sqsum.so**.

End of UNIX/Linux Only

For more information, see "Compiling a C UDR" on page 12-11.

To extend a built-in aggregate over a user-defined data type, you overload the appropriate operator function to handle the user-defined type. However, operator functions can also be used as part of an expression that does *not* involve aggregates. Therefore, aggregate support functions for built-in aggregates on user-defined data types (opaque types, distinct types, and named row types) must allocate a *new* state when they need to modify the state.

For example, the following SUM aggregate uses the overloaded **plus( )** operator to calculate the sum of values in the **col1** column:

```
SELECT SUM(col1) FROM tab2 WHERE ....;
```

For each aggregate argument, the SUM aggregate invokes the **plus( )** operator to add the aggregate argument (*agg_arg*) into the sum of the previous values in the aggregate state (*agg_state*), as follows:

```
plus(agg_state, agg_arg)
```

When you modify the aggregate state in-place, the value of the *agg_state* argument to **plus( )** changes. When **plus( )** exits, the *agg_state* argument holds the new sum of the aggregate arguments, which includes the *agg_arg* value.

However, the **plus( )** function is also valid in expressions that do *not* involve aggregates, as in the following query:

```
SELECT col1 FROM tab2 WHERE col1 + 4 > 17;
```

In this WHERE clause, the database server invokes the **plus( )** operator to add 4 to the **col1** value, as follows:

```
plus(col1, 4)
```

If the **plus( )** operator modifies the aggregate state in-place, the value of its first argument changes to hold the sum of **col1** and 4. It is not safe to modify arguments in place because the values of arguments (**col1** and 4) must not change. Therefore, when you modify the aggregate state in an operator function of a built-in aggregate, you must be careful *not* to use the "in-place" modification method.

Registering the Overloaded Operator Function

With the operator function written, compiled, and linked into a shared-object file, you can register this function in the database with the CREATE FUNCTION statement. You must have the appropriate privileges for this registration to be successful. For more information, see the chapter on user-defined aggregates in the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Figure 15-6 shows the CREATE FUNCTION statement that overloads the **plus( )** function with a new version that handles the **complexnum_t** named row type.

```
CREATE FUNCTION plus(arg1 complexnum_t, arg2 complexnum_t)
RETURNS complexnum_t
EXTERNAL NAME '/usr/udrs/aggs/sums/sqsum.so(complex_plus)'
LANGUAGE C;
```

Figure 15-6. Registering the Overloaded plus( ) Function

Tip: Because SUM is a built-in aggregate, you do not have to use the CREATE AGGREGATE statement to define the SUM aggregate.

Using the Extended Aggregate

Once you execute the CREATE FUNCTION statement in Figure 15-6 on page 15-15, you can use the SUM aggregate on **complexnum_t** columns. For example, suppose you create the **tab1** table as Figure 15-7 shows.

```
CREATE TABLE tab1
   (col1 INTEGER,
    col2 complexnum_t,
    col3 INTEGER);

INSERT INTO tab1
   VALUES (1, row(1.5, 3.7)::complexnum_t, 24);
INSERT INTO tab1
   VALUES (2, row(6.9, 2.3)::complexnum_t, 13);
INSERT INTO tab1
   VALUES (3, row(4.2, 9.4)::complexnum_t, 9);
INSERT INTO tab1
   VALUES (4, row(7.0, 8.5)::complexnum_t, 5);
INSERT INTO tab1
   VALUES (5, row(5.1, 6.2)::complexnum_t, 31);
INSERT INTO tab1
   VALUES (6, row(3.9, 4.6)::complexnum_t, 19);
```

Figure 15-7. A Table with a complexnum_t Column

The following query uses the SUM aggregate function on the **complexnum_t**
column, **col2**:

```
SELECT SUM(col2) FROM tab1;
```

With the rows that Figure 15-7 has inserted, the preceding query yields a
complexnum_t value of:

```
ROW(28.6, 34.7)
```

As a side effect of the new **plus( )** function, you can also add two **complexnum_t**
columns in an SQL expression, as follows:

```
SELECT complex_num1 + complex_num2 FROM complex_nums
WHERE id > 6;
```

Creating a User-Defined Aggregate

The built-in aggregates provide basic aggregations. However, if your data requires
some special aggregation, you can create a custom aggregate function, called a
user-defined aggregate. To implement your custom aggregation, you design an
aggregate algorithm, which consists of the following parts:

- The *aggregate state*, which contains the information that needs to be passed
  between iterations of the aggregate
- The *aggregation tasks*, which are implemented as special-purpose user-defined
  functions called *aggregate support functions*

To implement a user-defined aggregate function with C user-defined functions:

1. Determine the content and data type of the aggregate state.
2. Write the C UDRs that implement the required aggregate support functions for
   the data type on which you want to implement the user-defined aggregate.
3. Define the user-defined aggregate in the database with the CREATE
   AGGREGATE and CREATE FUNCTION statements.

After you complete these steps, you can use the aggregate in an SQL statement.

The following sections describe each of these development steps in more detail and
use the SQSUM1 user-defined aggregate (which Table 15-2 describes) as an
example.

Table 15-2. A Sample User-Defined Aggregate

User-Defined Aggregate	Description	Definition
SQSUM1	Sums all values and calculates the square of this sum	$(x_1 + x_2 + x_3 + \dots)^2$

Determining the Aggregate State

An aggregate is a series of iterations. Each iteration processes one aggregate argument (which contains one column value) and performs the necessary computations to merge it into a partial result. The partial result is a snapshot of the aggregate arguments that the aggregate has merged so far. Once the aggregate has received *all* column values, it returns a value to the calling statement, based on the final partial result.

Each iteration of the aggregate is a separate invocation of a user-defined function. If user-allocated memory has the default PER_ROUTINE memory duration, the database server automatically deallocates it after only *one* iteration of the iteration function. (For more information, see "Choosing the Memory Duration" on page 14-4.) Therefore, while an iteration executes, it can access *only* the following information:

- Its own local variables, which are deallocated at the end of each iteration and therefore unavailable to other iterations
- The aggregate argument, which contains a new column value for each iteration
- The *aggregate state*, which contains any nonlocal information that the iteration needs to perform its merge, including the partial result and any other external information (such as an operating-system file) that an iteration might need to access

During its invocation, each aggregate iteration merges the aggregate argument into the aggregate state and returns the updated state. The database server preserves the updated aggregate state and passes it into the next iteration of the aggregate. When you create a user-defined aggregate, you must determine what nonlocal information each iteration needs and then define an aggregate state to contain this information. Otherwise, the aggregate iterations cannot obtain the information they need to perform their computations.

Important: Design the aggregate state so that each aggregate support function can obtain all the state information that it needs.

As a starting point, try using the data type of the aggregate argument for the aggregate state. Such a state is called a *simple state*. For more information, see "Aggregate Support Functions for the Aggregate State" on page 15-27.

For example, to determine the state for the SQSUM1 user-defined aggregate (which Table 15-2 on page 15-17 describes), assess what tasks need to be performed in each iteration of SQSUM1. For each aggregate argument, SQSUM1 needs to add together the following values:

- The aggregate argument, which is passed to each iteration of the aggregate
- The partial sum of previous argument values, which must exist in the aggregate state

The data type that you choose for the aggregate state affects how the state must be managed. When the SQSUM1 aggregate receives INTEGER values as its aggregate

arguments, the sum of these values is also an INTEGER value. Therefore, the SQSUM1 aggregate has an integer state, which holds the partial sum.

Writing the Aggregate Support Functions

An *aggregate support function* is a special-purpose user-defined function that implements some task in the aggregate algorithm. You write aggregate support functions to initialize, calculate, and return the aggregate result to the calling code. An aggregate algorithm can include the following kinds of aggregate support functions.

Aggregate Support Function	Algorithm Step
INIT	Possible initialization tasks that must be performed before the iterations can begin
ITER	An iteration step, which is performed on each aggregate argument and merges this argument into a partial result
COMBINE	Merging one partial result with another partial result, thus allowing parallel execution of the user-defined aggregate
FINAL	Post-iteration tasks that must be performed after all aggregate arguments were merged into a partial result

The following sections summarize each of the aggregate support functions.

INIT Function: The INIT aggregate support function performs the initialization for the user-defined aggregate. Table 15-3 summarizes possible initialization tasks.

Table 15-3. Initialization Tasks for the INIT Aggregate Support Function

Initialization Task	More Information
Set-up for any additional resources outside the state that the aggregation might need	"Aggregate Support Functions That the Algorithm Requires" on page 15-26
Initial calculations that the user-defined aggregate might need	"Aggregate Support Functions That the Algorithm Requires" on page 15-26
Allocation and initialization of the aggregate state that the rest of the aggregation computation might need	"Aggregate Support Functions for the Aggregate State" on page 15-27
Handling of an optional set-up argument	"Implementing a Set-Up Argument" on page 15-34

The INIT support function is an *optional* aggregate support function. If your aggregate algorithm does not require any of the tasks in Table 15-3, you do *not* need to define an INIT function. When you omit the INIT function from your user-defined aggregate, the database server performs the state management for the aggregate state. For more information, see "Handling a Simple State" on page 15-28.

To declare an INIT support function as a C function, use the following syntax:

```
agg_state init_func(dummy_arg, set_up_arg)
    agg_arg_type dummy_arg;
    set_up_type set_up_arg; /* optional */
```

agg_state   is the data type of the aggregate state.

dummy_arg   is a dummy parameter that has the same data type as the

aggregate argument that this aggregate support function is to handle for the user-defined aggregate.

init_func is the name of the INIT aggregate support function.

set_up_arg is an optional parameter for the set-up argument. For more information, see "Implementing a Set-Up Argument" on page 15-34.

In the execution of a UDA, the database server calls the INIT function *before* it begins the actual aggregation computation. It passes in any optional set-up argument (*set_up_arg*) from the user-defined aggregate and copies any initialized aggregate state that INIT returns into the state buffer. For more information on the state buffer, see "Aggregate Support Functions for the Aggregate State" on page 15-27.

The first argument of the INIT function serves only to identify the data type of the aggregate argument that the user-defined aggregate handles. The routine manager uses this argument in routine resolution to determine the correct version of the overloaded INIT function. At the time of invocation, the routine manager just passes in a NULL value for the first argument of the INIT function, as the following syntax shows:

```
agg_state init_func(NULL, optional set-up argument)
```

Tip: For more information on how the routine manager resolves the overloaded aggregate support functions, see the chapter on aggregates in the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Figure 15-8 shows the INIT aggregate support function that handles an INTEGER argument for the SQSUM1 user-defined aggregate (which Table 15-2 on page 15-17 describes).

```
/* SQSUM1 INIT support function on INTEGER */
mi_integer init_sqsum1(dummy_arg)
    mi_integer dummy_arg;
{
    return (0);
}
```

Figure 15-8. INIT Aggregate Support Function for SQSUM1 on INTEGER

For other aggregate support functions of SQSUM1, see Figure 15-9 on page 15-21, Figure 15-10 on page 15-22, and Figure 15-11 on page 15-23.

ITER Function: The ITER aggregate support function performs the sequential aggregation or iteration for the user-defined aggregation. It merges a single aggregate argument into the partial result, which the aggregate state contains. The ITER function is a *required* aggregate support function; that is, you *must* define an ITER function for every user-defined aggregate. If you do *not* define an ITER function for your user-defined aggregate, the database server generates an error.

Tip: If the UDA does not have an INIT support function, the ITER support function can initialize the aggregate state. For more information, see "Handling a Simple State" on page 15-28.

If the UDA was registered with the HANDLESNULLS modifier in the CREATE AGGREGATE statement, the database server calls the ITER support function once

for each aggregate argument that passed to the user-defined aggregate. Each aggregate argument is one column value. If you omit the HANDLESNULLS modifier from CREATE AGGREGATE, the database server does *not* call ITER for any NULL-valued aggregate arguments. Therefore, NULL-valued aggregate arguments do not contribute to the aggregate result.

For example, suppose you execute the SQSUM1 user-defined aggregate (which Table 15-2 on page 15-17 describes) in the following query:

```
SELECT SQSUM1(col3) FROM tab1;
```

The **tab1** table (which Figure 15-7 on page 15-16 defines) contains 6 rows. Therefore, the preceding query (which contains no WHERE clause) causes 6 invocations of the SQSUM1 ITER function. Each invocation of this ITER function processes one value from the **col3** column.

To declare an ITER function as a C function, use the following syntax:

```
agg_state iter_func(current_state, agg_argument)
   agg_state current_state;
   agg_arg_type agg_argument;
```

agg_state   is the data type of the current and updated aggregate state. After the ITER function merges the *agg_argument* into the *current_state* state, it returns a pointer to the updated state.

agg_arg_type   is the data type of the *agg_argument*, which is the data type that the aggregate support function handles for the user-defined aggregate.

agg_argument   is a single aggregate argument, usually a column value, which the ITER function merges into the partial result in the *current_state* aggregate state.

current_state   is the current aggregate state, which previous calls to the ITER function and to the INIT function have generated.

iter_func   is the name of the ITER aggregate support function.

> **Important:** Make sure that the ITER support function obtains all state information that it needs from its "current_state" argument. The INIT function cannot maintain additional state information as user data in its **MI_FPARAM** structure because **MI_FPARAM** is not shared among the other aggregate support functions. However, the ITER function can store user data in **MI_FPARAM** that is not part of the aggregate result.

Figure 15-9 shows the ITER aggregate support function that handles an INTEGER argument for the SQSUM1 user-defined aggregate (which Table 15-2 on page 15-17 describes).

```
/* SQSUM1 ITER support function on INTEGER */
mi_integer iter_sqsum1(state, value)
   mi_integer state;
   mi_integer value;
{
   /* add 'state' and 'value' together */
      return (state + value);
}
```

Figure 15-9. ITER Aggregate Support Function for SQSUM1 on INTEGER

For other aggregate support functions of SQSUM1, see Figure 15-8 on page 15-19, Figure 15-10 on page 15-22, and Figure 15-11 on page 15-23.

COMBINE Function: The COMBINE aggregate support function allows your user-defined aggregate to execute in a parallel query. When a query that contains a user-defined aggregate is processed in parallel, each parallel thread operates on a one subset of selected rows. The COMBINE function merges the partial results from two such subsets. This aggregate support function ensures that the result of aggregating over a group of rows sequentially is the same as aggregating over two subsets of the rows in parallel and then combining the results.

The COMBINE function is required for parallel execution. When a query includes a user-defined aggregate, the database server uses parallel execution when the query includes only aggregates. However, the COMBINE function might be used even when a query is not parallelized. For example, when a query contains both distinct and non-distinct aggregates, the database server can decompose the computation of the non-distinct aggregate into sub-aggregates based on the distinct column values. Therefore, you *must* provide a COMBINE function for every user-defined aggregate.

If you do *not* define an COMBINE function for your user-defined aggregate, the database server generates an error. However, if your user-defined aggregate uses a simple state, the COMBINE function can be the same as the ITER function. For more information, see "Handling a Simple State" on page 15-28.

To declare a COMBINE function as a C function, use the following syntax:

agg_state combine_func(agg_state1, agg_state2)
  agg_state agg_state1, agg_state2;

agg_state   is the data type of the two partial aggregate states (*agg_state1* and *agg_state2*) as well as the updated aggregate state, which the COMBINE function returns.

agg_state1   is the aggregate state from one parallel thread.

agg_state2   is the aggregate state from the second parallel thread.

combine_func   is the name of the COMBINE aggregate support function.

In the execution of a UDA, the database server calls the COMBINE once for each pair of threads (*agg_state1* and *agg_state2*) that execute a parallel query that contains the user-defined aggregate. When the COMBINE function combines two partial results, it might also need to release resources associated with one of the partial results.

Figure 15-10 shows the COMBINE aggregate support function that handles an INTEGER argument for the SQSUM user-defined aggregate (which Table 15-2 on

page 15-17 describes).

```
/* SQSUM1 COMBINE support function on INTEGER */
mi_integer combine_sqsum1(state1, state2)
   mi_integer state1, state2;
{
   /* Return the new partial sum from two parallel partial
    * sums
    */
   state1 += state2;
   return (state1);
}
```

Figure 15-10. COMBINE Aggregate Support Function for SQSUM1 on INTEGER

For other aggregate support functions of SQSUM1, see Figure 15-8 on page 15-19, Figure 15-9 on page 15-21, and Figure 15-11 on page 15-23. For more information on parallel execution of a UDA, see "Executing a User-Defined Aggregate in Parallel Queries" on page 15-35.

FINAL Function: The FINAL aggregate support function performs the post-iteration tasks for the user-defined aggregate. Table 15-4 summarizes possible post-iteration tasks.

Table 15-4. Post-Iteration Tasks for the FINAL Aggregate Support Function

Post-Iteration Task	More Information
Type conversion of the final state into the return type of the user-defined aggregate	"Returning an Aggregate Result Different from the Aggregate State" on page 15-35
Post-iteration calculations that the user-defined aggregate might need	"Aggregate Support Functions That the Algorithm Requires" on page 15-26
Deallocation of memory that the INIT aggregate support function has allocated	"Managing a Pointer-Valued State" on page 15-32

The FINAL function is an *optional* aggregate support function. If your user-defined aggregate does not require one of the tasks in Table 15-4, you do *not* need to define a FINAL function. When you omit the FINAL function from your user-defined aggregate, the database server returns the final aggregate state as the return value of the user-defined aggregate. If this state does not match the expected data type of the aggregate return value, the database server generates a data type mismatch error.

Important: In general, the FINAL support function must not deallocate the aggregate state. Only for a pointer-valued state (in which the aggregate support functions must handle all state management) does the FINAL support function need to deallocate the state. For more information, see "Managing a Pointer-Valued State" on page 15-32.

To declare a FINAL function as a C function, use the following syntax:

agg_type final_func(final_state)
   agg_state final_state;

agg_type   is the data type of the aggregate result, which is what the user-defined aggregate returns to the SQL statement in which it was invoked.

final_state   is the final aggregate state, as previous calls to the INIT and ITER functions have generated.

final_func   is the name of the FINAL aggregate support function.

In the execution of a UDA, the database server calls the FINAL function *after* all iterations of the ITER function are complete.

Figure 15-11 shows the aggregate support functions that handle an INTEGER argument for the SQSUM user-defined aggregate (which Table 15-2 on page 15-17 describes).

```
/* SQSUM1 FINAL support function on INTEGER */
mi_integer final_sqsum1(state)
   mi_integer state;
{
   /* Calculate square of sum */
   state *= state;

   return (state);
}
```

Figure 15-11. FINAL Aggregate Support Function for SQSUM1 on INTEGER

For other aggregate support functions of SQSUM1, see Figure 15-8 on page 15-19, Figure 15-9 on page 15-21, and Figure 15-10 on page 15-22.

Defining the User-Defined Aggregate

You can define the user-defined aggregate *before* you create the C implementation of the aggregate support functions. However, you must ensure that the names of the C functions match the names in the CREATE FUNCTION statements that register them.

To define a user-defined aggregate in a database with SQL:

1. Register the user-defined aggregate in the database with the CREATE AGGREGATE statement
2. Register the aggregate support functions in the database with the CREATE FUNCTION statement

The development steps for a UDA list the definition of the UDA *after* the aggregate support functions are written. However, the CREATE AGGREGATE statement does *not* verify that the aggregate support functions it lists were registered *nor* does the CREATE FUNCTION statement verify that the executable C code exists.

Figure 15-12 shows the CREATE AGGREGATE statement that defines the SQSUM1 user-defined aggregate (which Table 15-2 on page 15-17 describes) in the database.

```
CREATE AGGREGATE sqsum1
   WITH (INIT = init_sqsum1,
      ITER = iter_sqsum1,
      COMBINE = combine_sqsum1,
      FINAL = final_sqsum1);
```

Figure 15-12. Registering the SQSUM1 User-Defined Aggregate

Suppose that the INIT, ITER, COMBINE, and FINAL aggregate support functions for the SQSUM1 aggregate are compiled and linked into a shared-object module

named **sqsum**.

UNIX/Linux Only

On UNIX or Linux, the executable code for the SQSUM1 aggregate support
functions would be in a shared library named **sqsum.so**.

End of UNIX/Linux Only

Figure 15-13 shows the CREATE FUNCTION statements that register the aggregate
support functions for the SQSUM1 user-defined aggregate to handle the INTEGER
data type.

```
CREATE FUNCTION init_sqsum1(dummy_arg INTEGER)
    RETURNS INTEGER
    WITH (HANDLESNULLS)
    EXTERNAL NAME '/usr/udrs/aggs/sums/sqsum.so'
    LANGUAGE C;

CREATE FUNCTION iter_sqsum1(state INTEGER,
        one_value INTEGER)
    RETURNS INTEGER
    EXTERNAL NAME '/usr/udrs/aggs/sums/sqsum.so'
    LANGUAGE C;

CREATE FUNCTION combine_sqsum1(state1 INTEGER,
        state2 INTEGER)
    RETURNS INTEGER
    EXTERNAL NAME '/usr/udrs/aggs/sums/sqsum.so'
    LANGUAGE C;

CREATE FUNCTION final_sqsum1(state INTEGER)
    RETURNS INTEGER
    EXTERNAL NAME '/usr/udrs/aggs/sums/sqsum.so'
    LANGUAGE C;
```

Figure 15-13. Registering the Aggregate Support Functions for SQSUM1 to Handle INTEGER

The registered names of the aggregate support functions must match the names
that the CREATE AGGREGATE statement lists. For example, in Figure 15-13, a
CREATE FUNCTION statement registers the INIT support function as
init_sqsum1, which is the same name that the INIT option lists in the CREATE
AGGREGATE statement (see Figure 15-12 on page 15-23).

In addition, the CREATE FUNCTION for the INIT support function *must* include
the HANDLESNULLS routine modifier so that the database server can pass the
INIT function the dummy NULL-valued argument.

For more information on the use of the CREATE AGGREGATE and CREATE
FUNCTION statements to define user-defined aggregates, see the chapter on
aggregates in the *IBM Informix User-Defined Routines and Data Types Developer's
Guide*. For the syntax of these statements, see the *IBM Informix Guide to SQL:
Syntax*.

Using the User-Defined Aggregate

After you complete the aggregate-development steps, the end user can use the
user-defined aggregate on the defined data type in SQL statements. However, use
of the user-defined aggregate does assume the appropriate privileges.

For the **tab1** table, which Figure 15-7 on page 15-16 defines, the following query uses the new SQSUM1 aggregate function on the INTEGER column, **col3**:

```
SELECT SQSUM1(col3) FROM tab1;
```

With the rows that Figure 15-7 has inserted, the preceding query yields an INTEGER value of 10201.

To be able to use SQSUM1 on other data types, you need to ensure that the appropriate aggregate support functions exist for this data type. For example, "SQSUM2 User-Defined Aggregate" on page 15-39 shows the definition of a version of the SQSUM aggregate on both an INTEGER and a named row type.

Determining Required Aggregate Support Functions

Figure 15-14 shows the execution sequence of aggregate support functions for a user-defined aggregate that is *not* executed in a parallel query.
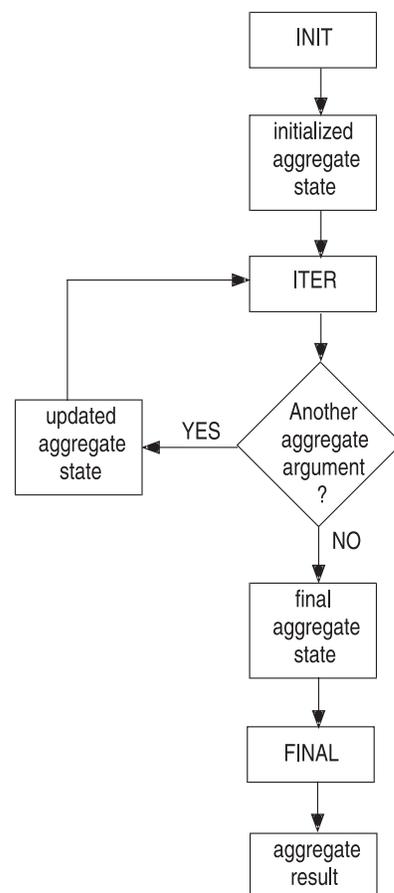
Figure 15-14. Execution of a Nonparallel User-Defined Aggregate

Tip: For information about how to execute a user-defined aggregate in parallel queries, see "Executing a User-Defined Aggregate in Parallel Queries" on page 15-35.

These aggregate support functions use the aggregate state to pass shared information between themselves.

As you design your aggregate algorithm, you must determine which of the support functions the algorithm requires. As a minimum, the user-defined

aggregate *must* have an ITER function. It is the ITER function that performs a single iteration of the aggregate on one aggregate argument. Although Figure 15-14 on page 15-25 shows the execution of both the INIT and FINAL support functions, these functions are optional for a user-defined aggregate. In addition, the COMBINE function, though required, often does not require separate code; it can simply call the ITER function.

Writing aggregate support functions that your user-defined aggregate does not require means unnecessary coding and execution time. Therefore, it is important to assess your aggregate for required functions. The following table shows the design decisions in the determination of required aggregate support functions.

Design Decision	Aggregate Support Functions Involved	More Information
Does the algorithm require initialization or clean-up tasks?	INIT and FINAL	"Aggregate Support Functions That the Algorithm Requires" on page 15-26
Does the aggregate have a simple aggregate state?	INIT, COMBINE, and FINAL	Yes: "Handling a Simple State" on page 15-28 No: "Handling a Nonsimple State" on page 15-29
Does the aggregate have a set-up argument?	INIT	"Implementing a Set-Up Argument" on page 15-34
Does the aggregate return a value whose data type is different from the aggregate state?	FINAL	"Returning an Aggregate Result Different from the Aggregate State" on page 15-35
Does the aggregate have special needs to run in a parallel query?	COMBINE	"Executing a User-Defined Aggregate in Parallel Queries" on page 15-35

You can overload the aggregate support functions to provide support for different data types. Any overloaded version of the UDA, however, cannot omit any of the aggregate support functions that the CREATE AGGREGATE statement has listed or use any support function that CREATE AGGREGATE has *not* specified.

When the database server executes a UDA (regardless of the data type of the aggregate argument), the database expects to find *all* the aggregate support functions that the CREATE AGGREGATE statement has registered. Therefore, if you omit a support function for one of the reasons in the preceding table, *all* versions of the aggregate for all data types must be able to execute using only the aggregate support functions that CREATE AGGREGATE specifies.

Aggregate Support Functions That the Algorithm Requires: To implement a user-defined aggregate, you must develop an algorithm that calculates an aggregate return value based on the aggregate-argument values. You need to break this algorithm into the following steps.

Algorithm Step	Aggregate Support Function
Calculations or initializations that must be done *before* the iterations can begin	INIT
Calculations that must be done on *each* aggregate argument to merge this argument into a partial result	ITER

Algorithm Step	Aggregate Support Function
Post-iteration tasks must be performed *after* all aggregate arguments were merged into a partial result	FINAL

All aggregate algorithms *must* include an ITER function to handle each aggregate argument. However, the INIT and FINAL support functions are optional. To determine whether your algorithm requires an INIT or FINAL function, make the following design assessments:

- Are there calculations or initializations that must be done *before* the iterations can begin?

  If the algorithm requires additional resources to perform its task (such as operating-system files or smart large objects), use the INIT function to set up these resources. The INIT function can also initialize the partial result.

- Are there post-iteration tasks that must be performed *after* all aggregate arguments were merged into a partial result?

  If the INIT function has set up resources to perform the aggregation, the FINAL function can deallocate or close resources so that they are free for other users. In addition, if the aggregation requires calculations that must be performed on the final partial result, use the FINAL function to perform these calculations.

For example, the following algorithm defines the SQSUM1 aggregate (which Table 15-2 on page 15-17 describes):

$$(x_1 + x_2 + \ldots)^2$$

where each x_i is one column value; that is, one aggregate argument. The ITER function for SQSUM1 takes a single aggregate argument and adds it to a partial sum (see Figure 15-9 on page 15-21). The algorithm does not require initialization of additional resources. Therefore, no INIT function is required for this task. However, the INIT function can initialize the partial result (see Figure 15-8 on page 15-19).

The SQSUM1 user-defined aggregate does require post-iteration calculations. When the last iteration is reached, the partial sum needs to be squared to obtain the aggregate return value. This final calculation is performed in a FINAL function and returned as the return value for the SQSUM1 aggregate (see Figure 15-11 on page 15-23).

The SUMSQ user-defined aggregate (described on page 15-37) is an example of a user-defined aggregate that requires neither initialization nor post-iteration tasks. Therefore, it does not require the INIT and FINAL support functions.

Aggregate Support Functions for the Aggregate State: The aggregate support functions pass information about the aggregate among themselves in the aggregate state.

Tip: For an explanation of the aggregate state, see "Determining the Aggregate State" on page 15-17.

The database server invokes *all* aggregate support functions as regular UDRs. Therefore, each support function has a default memory duration of PER_ROUTINE, which means that the database server frees any memory that the support function allocates when that function terminates. However, the aggregate state must be valid across *all* invocations of all aggregate support functions,

including possible multiple iterations of the ITER function. Therefore, a special state buffer (with a PER_COMMAND memory duration) must exist so that the aggregate state is available to all aggregate support functions. The database server then passes this state buffer to each invocation of an aggregate support function.

The purpose of the INIT support function is to return a pointer to the initialized aggregate state. To determine whether your aggregate state requires an INIT support function for state management, assess the data type of this state. The aggregate-state data type determines whether the INIT function must handle state management, as follows:

- If the data type of the aggregate state is the *same* as the aggregate argument, the aggregate has a *simple state*.

  If your user-defined aggregate uses a simple state, the database server can perform the state management. It can automatically allocate the state buffer for the aggregate state. Therefore, the INIT support function does *not* need to handle this allocation.

- If the data type of the aggregate state is *different* from the aggregate argument, the aggregate has a *nonsimple state*.

  There are several types of non-simple states possible. The database server cannot perform state management for these non-simple states. Instead, the INIT support functions must perform the state-management tasks to handle non-simple states.

Once the user-defined aggregate has an aggregate state, the database server passes a pointer to this state buffer to *each* invocation of the ITER function and to the FINAL function.

Handling a Simple State: A *simple state* is an aggregate state whose data type is the same as the aggregate argument. At any point in the iteration, a simple state contains *only* the partial result of the aggregation. For example, the SUM built-in aggregate uses a simple state because its state contains only the partial result: the running total of the aggregate arguments. When the SUM aggregate operates on INTEGER aggregate arguments, it creates an integer partial sum for these arguments. Therefore, the data type of its aggregate argument and aggregate state is the same.

However, the AVG built-in aggregate does *not* use a simple state. Because it must divide the total by the number of values processed, its state requires two values: the running total and the number of arguments processed. When the AVG aggregate operates on INTEGER aggregate arguments, it creates an integer partial sum and an integer count for these arguments. Therefore, the data type of its aggregate argument (INTEGER) cannot be not the same as its aggregate state (two INTEGER values).

When a user-defined aggregate has a simple state, the following items apply:

- The INIT aggregate support function does *not* need to allocate the aggregate state.

  In this case, the database server automatically performs the state management. If a UDA with a simple state does not include any other tasks that require an INIT support function (see Table 15-3 on page 15-18), you can omit the INIT function from the definition of the UDA. The only possible state-management task you might want to perform in the INIT function is state initialization. For more information, see "When to Allocate and Deallocate a State" on page 15-33.

- The COMBINE aggregate support function can just call the ITER support function.

In this case, you do not have to create special code in the COMBINE function for the handling of parallel execution. Instead, the ITER function can perform the merge of two partial results. For more information, see "Executing a User-Defined Aggregate in Parallel Queries" on page 15-35.

• The FINAL function is *not* required if the data type of the simple state is the *same* as the aggregate result.

In this case, the aggregate argument, aggregate state, and aggregate result have the same data type. Such a user-defined aggregate is called a *simple binary operator*. If a UDA is a simple binary operator and does not include any other tasks that require a FINAL support function (see Table 15-4 on page 15-22), you can omit the FINAL function from the definition of the UDA. For more information, see "Returning an Aggregate Result Different from the Aggregate State" on page 15-35.

When a UDA does *not* include an INIT function, the database server takes the following state-management steps:

• Allocates the PER_COMMAND state buffer to hold the aggregate state

The database server can determine the size of the state buffer from the data type of the aggregate argument, which is passed into the user-defined aggregate.

• Initializes this aggregate state to an SQL NULL value whose data type is the same as the aggregate argument

• Passes the NULL-valued aggregate state to the first invocation of the ITER support function

The ITER support function can just use the system-allocated state buffer to hold the state information. If you have some minor initialization tasks that you need to perform, the ITER function can check for a NULL-valued aggregate state on its first iteration. If the state is NULL, ITER can initialize the state to its appropriate value. In this way, you can perform minor state initialization without the overhead of a separate invocation of the INIT function.

When a UDA does *not* include a FINAL function, the database server passes the final state as the aggregate result of the user-defined aggregate.

The implementation on the SQSUM1 aggregate includes an INIT support function that initializes the aggregate state (see Figure 15-8 on page 15-19). However, because SQSUM1 has a simple state, this INIT function is *not* required. Instead, an ITER function can check for a NULL-valued state and perform the state initialization. The ITER support function of the SQSUM2 aggregate (see Figure 15-18 on page 15-40) shows this type of implementation.

The SUMSQ user-defined aggregate (described on page 15-37) also has a simple state and therefore does not require an INIT support function for state management.

Handling a Nonsimple State: When the data type of the aggregate argument is not adequate for the state information, you must use a *nonsimple state* for your UDA. A nonsimple state is an aggregate state whose data type is *not* the same as the aggregate argument. Possible uses for a nonsimple state include an aggregate state that contains:

• An aggregate state that contains more information than the aggregate-argument data type can hold

• An aggregate state that contains information of a data type different than that of the aggregate argument

When the aggregate-argument data type is not adequate for the state information, you must determine the appropriate data structure to hold the state information. This data structure depends upon the size of the aggregate state that you need to maintain, as the following table shows.

Nonsimple State	Description	More Information
Single-valued state	Consists of information that can be stored in an Informix built-in data type	"Managing a Single-Valued State" on page 15-30
Opaque-type state	Consists of several values, but these values do *not* exceed the maximum size of an opaque data type	"Managing an Opaque-Type State" on page 15-31.
Pointer-valued state	Consists of values whose size *does* exceed the maximum size of an opaque data type	"Managing a Pointer-Valued State" on page 15-32.

If your user-defined aggregate uses a single-valued or opaque-type state, the database server can still perform the state management. However, it cannot provide all necessary state management for a pointer-valued state.

Managing a Single-Valued State: A *single-valued state* uses a built-in SQL data type to hold the aggregate state. Use a single-valued state for an aggregate state that can fit into a built-in data type but whose data type does *not* match that of the aggregate argument.

Tip: Built-in SQL data types are provided data types. For more information, see the chapter on data types in the *IBM Informix Guide to SQL: Reference*.

To use a single-valued state for a UDA:

1. Write the appropriate aggregate support functions so that they handle a single-valued state.

   Declare the state parameters and return values of the aggregate support functions to use the DataBlade API data type that corresponds to the built-in SQL data type that your state requires. For a list of these data type correspondences, see Table 1-1 on page 1-8. Information on how to handle state management for a single-valued state is provided below.

2. Register the aggregate support functions with the CREATE FUNCTION statement.

   Specify the built-in SQL data type as the data type of the state parameters and return values in the function signatures of the aggregate support functions.

The database server can perform memory management for a single-valued state because it can determine the size of a built-in data type. Therefore, you do not need to allocate memory for a single-valued state in the INIT support function.

In the ITER function, you can initialize or update a single-valued state in either of the following ways:

- In-place state update

  To modify the state, change the value (or values) of the DataBlade API variable that the database server has passed into the ITER function as the state argument.

- Allocate a new state

To modify the state, declare a local variable or allocate PER_ROUTINE memory for a new variable and put the new values into this variable.

For more information, see "When to Allocate and Deallocate a State" on page 15-33.

For a single-valued state, the FINAL support function does *not* need to perform any state-management tasks. However, it must convert to the data type of the final aggregate state to that of the aggregate result. For more information, see "Returning an Aggregate Result Different from the Aggregate State" on page 15-35.

Managing an Opaque-Type State: An *opaque-type state* uses an opaque data type to hold the aggregate state. A possible use for an opaque-type state is to include an aggregate state that contains more information than the aggregate-argument data type or a built-in data type can hold. The size of an aggregate state that can be implemented as an opaque-type state is limited by the maximum size of an opaque type.

Important: The maximum size of an opaque type is system dependent. On many systems, this limit is 32 kilobytes. Consult your machine notes for the limit on your system. If your aggregate state might contain more data than the opaque-type limit, you must use a pointer-valued state instead. For more information, see "Managing a Pointer-Valued State" on page 15-32.

To use an opaque-type state for a UDA, write the appropriate aggregate support functions so that they handle an opaque-type state.

Declare the state parameters and return values of the aggregate support functions to use the *internal* format of the opaque type. This internal format is usually a C **struct** structure. For more information, see "Determining Internal Representation" on page 16-3.

Handling State Management for an Opaque-Type State: Register the opaque data type in the database with the CREATE OPAQUE TYPE statement.

After you register the opaque data type, the database server can obtain information about the data type of the state value when the CREATE FUNCTION statement registers the function signatures of the aggregate support functions.

You need to write the opaque-type support functions *only* if you need such functionality in the aggregate support functions. For example, the input and output support functions might be useful when you debug your UDA. If you do write opaque-type support functions, you must compile and link them into a shared-object module, as "Compiling a C UDR" on page 12-11 describes.

Register the aggregate support functions with the CREATE FUNCTION statement.

Specify the registered opaque type as the data type of the state parameters and return values in the function signatures of the aggregate support functions.

The database server can perform memory management for an opaque-type state because it can determine the size of the opaque data type. The CREATE OPAQUE TYPE statement registers the opaque type, including its size, in the system catalog tables of the database. From the system catalog tables, the database server can

determine the size of the aggregate state to allocate. Therefore, you do *not* need to allocate the opaque-type state in the INIT support function.

In the ITER function, you can initialize or update an opaque-type state in either of the following ways:

- In-place state update

  To modify the state, change the values in the internal opaque-type structure that the database server has passed into the ITER function as an argument.

- Allocate a new state

  To modify the state, allocate PER_ROUTINE memory for a new internal opaque-type structure and put the new values into this structure.

  If you need to initialize the internal structure of the opaque type, the INIT or ITER function can allocate PER_ROUTINE memory for the structure perform the appropriate initializations. When the support function exits, the database server copies the contents of this PER_ROUTINE structure into the PER_COMMAND system-allocated state buffer.

For more information, see "When to Allocate and Deallocate a State" on page 15-33. For an example of a user-defined aggregate that uses an opaque-type state, see the description of the PERCENT_GTR aggregate on page 15-43.

Managing a Pointer-Valued State: A *pointer-valued state* uses the POINTER data type as the aggregate state. The **mi_pointer** data type is the DataBlade API type that represents the SQL data type, POINTER. (For more information, see "Pointer Data Types (Server)" on page 2-31.) Use a pointer-valued state when an aggregate state might contain more information than can fit into the maximum opaque-type size.

Important: The maximum size of an opaque type is system dependent. On many systems, this limit is 32 kilobytes. Consult your machine notes for the limit on your system. If your aggregate state contains less data than the opaque-type limit, use an opaque-type state instead. For more information, see "Managing an Opaque-Type State" on page 15-31.

To use a pointer-valued state for a UDA:

1. Write the appropriate aggregate support functions so that they handle a pointer-valued state.

   Declare the state parameters and return values of the aggregate support functions to use the **mi_pointer** data type. Information on how to handle state management of a pointer-valued state follows.

2. Register the aggregate support functions with the CREATE FUNCTION statement.

   Specify the POINTER data type for the state parameters and return values in the function signatures of the aggregate support functions.

The database server *cannot* perform state management for a pointer-valued state because it cannot determine the size of the state. The DataBlade API data type **mi_pointer** is a **typedef** for the following C data type:

```
void *
```

Because this data type is only a pointer, the database server cannot determine how large the aggregate state is. Therefore, it cannot allocate the PER_COMMAND system-allocated state buffer. In this case, the INIT and FINAL aggregate support functions are *not* optional. They must perform state management of the nonsimple aggregate state, as follows:

- The INIT function can allocate and initialize the aggregate state.

  The INIT function must also allocate any related resources that the aggregate state might need. Keep in mind that the database server does *not* interpret the contents of the pointer-valued state. It cannot manage any objects that the state type might reference. Therefore, use states with embedded pointers with caution.

- The ITER function must perform an in-place update to initialize or modify a pointer-valued state.

  Once you allocate the pointer-valued state, the database server passes a pointer to this state to the other aggregate support functions. Initialize or update the pointer-valued state *only* with an in-place update. For more information, see "When to Allocate and Deallocate a State" on page 15-33.

- The FINAL function can handle deallocation of resources that the INIT function has set up.

  For a pointer-valued state, the FINAL function must always deallocate the aggregate state. If your INIT support function has allocated related resources that the aggregate state uses, make sure that the FINAL function deallocates these resources.

Important: Make sure that you use a memory duration that extends for the life of the user-defined aggregate. A PER_ROUTINE memory duration (the default) expires after one invocation of the ITER function completes. Therefore, you must use a memory duration of at least PER_COMMAND for memory associated with the state.

When to Allocate and Deallocate a State: To each invocation of the ITER support function, the database server automatically passes a pointer to the state buffer. When you need to initialize or update the state information, the ITER function can handle the modification in either of two ways, as the following table describes.

Changing the State	State Memory Duration	Results
Merge the aggregate argument into the existing state *in-place* and return the existing state.	The existing state has a PER_COMMAND memory duration: • For single-valued and opaque-type states, this state is the system-allocated state buffer. • For a pointer-valued state, this state is a user-allocated state buffer.	The new state value is at the address that the database server has passed into the ITER function. The ITER function then returns this address as the updated state. Because the state memory has a PER_COMMAND memory duration, the database server can re-use the same state for subsequent invocations of ITER.
Allocate fresh memory for a *new state*, merge the existing state with the new aggregate argument into this state, and return this new state.	The new state has a PER_ROUTINE memory duration: • For a single-valued state, this state can be either a declared local variable or user-allocated PER_ROUTINE memory. • For an opaque-type state, the new state must be user-allocated PER_ROUTINE memory. • For a pointer-valued state, this state is user-allocated memory with either a PER_ROUTINE or PER_COMMAND memory duration. However, for PER_COMMAND memory, you must also handle deallocation of the old state. For more information, see "Managing a Pointer-Valued State" on page 15-32.	The new state value is at the address of the new state. The ITER function then returns the address of the new state as the updated state. Because this memory has a PER_ROUTINE memory duration, the database server must copy the returned state back into the PER_COMMAND buffer.

The new state method can be slower than the in-place method. Design your ITER support function to use the in-place method whenever possible. When the database server can skip the copy operation, you can improve performance of your UDA.

To determine which of these methods was used in the ITER support function, the database server compares the state value that ITER returns and the state value that was passed into ITER. If these two pointers identify the *same* memory location, the ITER function has modified the state in-place. Therefore, the database server does *not* need to perform the copy operation. If these two pointers identify different memory locations, the database server proceeds with the copy operation.

Aggregate support functions have the following restrictions on the deallocation of an aggregate state:

- For any state other than a pointer-valued state, no aggregate support function must deallocate the state memory.
- No aggregate support function can return a NULL-valued pointer as the state.

Implementing a Set-Up Argument: You can define a UDA so that the end user can supply a *set-up argument* to the aggregate. The set-up argument can customize the aggregate for a particular invocation. For example, the PERCENT_GTR user-defined aggregate (see page 15-43) determines the percentage of numbers greater than a particular value. The UDA could have been implemented so that the value to compare against is hardcoded into the UDA. However, this would mean a separate user-defined aggregate that checks for values greater than 10, another that checks for values greater than 15, and so on.

Instead, the PERCENT_GTR aggregate accepts the value to compare against as a set-up argument. In this way, the end user can determine what values are needed, as follows:

```
SELECT PERCENT_GTR(col1, 10) FROM tab1; -- values > 10;
SELECT PERCENT_GTR(col1, 15) FROM tab1; -- values > 15;
```

The database server passes in the set-up argument as the second argument to the INIT function. Therefore, the INIT support function must handle the set-up argument. Usually, this handling involves performing any initial processing required for the value and then saving this value in the aggregate state. It might also check for a possible SQL NULL value as a set-up argument.

This set-up argument is optional, in the sense that you can define a UDA with one or without one. However, if you define your UDA to include a set-up argument, the end user *must* provide a value for this argument. When the UDA is invoked with two arguments (aggregate argument and set-up argument), the database server looks for an INIT function with two arguments. If you omit the set-up argument when you invoke the UDA, the database server looks for an INIT function with just one argument.

To indicate no set-up argument, the end user can provide the SQL NULL value as a set-up value. However, if you really want to make the set-up argument truly optional for the end user, you must create and register two INIT functions:

- One that takes two arguments
- One that takes only one argument

  In this case, you could assign the set-up argument some known default value.

As the writer of the UDA, you need to decide whether this feature is useful.

Returning an Aggregate Result Different from the Aggregate State: The aggregate result is the value that the user-defined aggregate returns to the calling SQL statement. If the user-defined aggregate does not include a FINAL support function, the database server returns the final aggregate state; that is, it returns the value of the aggregate state after the *last* aggregate iteration. However, if your UDA needs to return a value whose data type is different from the aggregate state, use a FINAL support function to convert the final aggregate state to the data type that you want to return from the aggregate.

For example, the PERCENT_GTR user-defined aggregate (see page 15-43) returns the percentage of values greater than some value as a percentage; that is, as a fixed-point number in the range 0.00 to 100.00. To handle integer values, the user-defined aggregate would require an aggregate state that holds the following values:

- The total number of aggregate arguments greater than 10
- The total number of aggregate arguments
- The value to compare against

However, the aggregate result of PERCENT_GTR is a fixed-point number. Therefore, you would not want the aggregate to return the final state to the calling SQL statement. Instead, the FINAL support function needs to perform the following steps:

1. Divide the total number of arguments that are greater than 10 by the total number of arguments and multiply by 100.
2. Return the fixed-point quotient, which is the percentage of values greater than 10.

For the complete example of the PERCENT_GTR user-defined aggregate, see page 15-43.

Executing a User-Defined Aggregate in Parallel Queries: The database server can break up the aggregation computation into several pieces and compute them in parallel. Each piece is computed sequentially as follows:

1. The INIT support function initializes execution in the parallel thread.
2. For each aggregate argument in the subset, the ITER support function merges the aggregate argument into a partial result.

The database server then calls the COMBINE support function to merge the partial states, two at a time, into a final state. For example, for the AVG built-in aggregate, the COMBINE function would add the two partial sums and adds the two partial counts. Finally, the database server calls the FINAL support function on the final state to generate the aggregate result.

Figure 15-15 shows the execution sequence of aggregate support functions for a user-defined aggregate that is executed in two parallel threads.
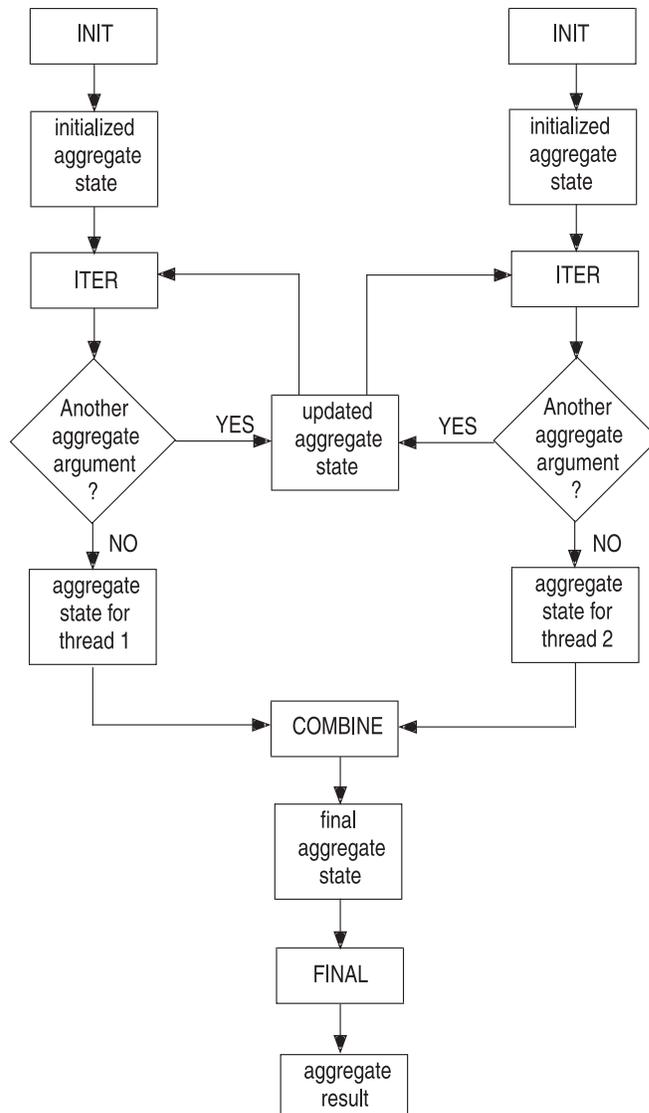
Figure 15-15. Parallel Execution of a UDA

Figure 15-15 shows how the COMBINE function is used to execute a user-defined aggregate with two parallel threads. For more than two parallel threads, the database server calls COMBINE on two thread states to obtain one, combines this state with another thread state, and so on until it has processed all parallel threads. The database server makes the decision whether to parallelize execution of a user-defined aggregate and the degree of such parallelism. However, these decisions are invisible to the end user.

Parallel aggregation must give the same results as an aggregate that is not computed in parallel. Therefore, you must write the COMBINE function so that the result of aggregating over the entire group of selected rows is the same as aggregating over two partitions of the group separately and then combining the results.

For an example of COMBINE functions in user-defined aggregates, see "Sample User-Defined Aggregates" on page 15-37.

Sample User-Defined Aggregates

This section provides the sample user-defined aggregates that the following table describes.

User-Defined Aggregate	Description
SUMSQ	Squares each value and calculates the sum of these squared values
SQSUM2	Sums all values and calculates the square of this sum
PERCENT_GTR	Determines the percentage of values greater than a user-specified value
X_PERCENTILE	Determines the value in a group of values that is the x-percentile, where x is a percent that the end user specifies.

Each description includes the aggregate support functions written in C and the SQL statements to define the user-defined aggregate in the database.

SUMSQ User-Defined Aggregate: The SUMSQ user-defined aggregate squares each value and calculates the sum of these squared values. It has the following algorithm:

$$x_1^2 + x_2^2 + \ldots$$

where each x_i is one column value; that is, one aggregate argument.

To determine the aggregate state for SUMSQ, examine what information needs to be available for each iteration of the aggregate. To perform one iteration of SUMSQ, the ITER function must:

1. Square the aggregate argument.

   The ITER function has access to the aggregate argument because the database server passes it in. Therefore, ITER does not require additional information to perform this step.

2. Add the squared argument to the partial sum of previous squared values.

   To add in the squared argument, the aggregate must keep a partial sum of the previous squared values. For the ITER function to have access to the partial sum from the previous iterations, the aggregate state must contain it.

The SUMSQ has a *simple state* because the data type of the partial sum is the same as that of the aggregate argument. For example, when the SUMSQ aggregate receives INTEGER values, this partial sum is also an INTEGER value. Therefore, SUMSQ can allow the database server to manage this state, which has the following effect on the design of the aggregate support functions:

- The INIT support function does *not* need to perform state management.

  An aggregate with a simple state does not need to explicitly handle the allocation and deallocation of the aggregate state. Instead, the database server automatically allocates the aggregate state and initializes it to NULL. Therefore, the INIT function does not require other INIT-function tasks (see Table 15-3 on page 15-18). Therefore, this support function can safely be omitted from the aggregate definition.

- The COMBINE support function can be the same as its ITER function.

  No special processing is required to merge two partial states. The ITER function can adequately perform this merge.

Before the iterations begin, the partial sum needs to be initialized to zero (0). However, because the INIT function is *not* required for state management, this aggregate initializes the state in the first invocation of its ITER function. The ITER function then calculates the square of a single aggregate argument, and adds this value to a partial sum. When the last iteration is reached, the final partial sum is the value that the SUMSQ aggregate returns. Therefore, the SUMSQ algorithm does *not* require a FINAL function for post-iteration tasks.

Figure 15-16 shows the required aggregate support functions that handle an INTEGER argument for the SUMSQ user-defined aggregate.

```
/* SUMSQ ITER support function on INTEGER */
mi_integer iter_sumsq(state, value, fparam)
   mi_integer state;
   mi_integer value;
   MI_FPARAM *fparam;
{
   /* If 'state' is NULL, this is the first invocation.
    * Just return square of 'value'.
    */
   if ( mi_fp_argisnull(fparam, 0) )
      return (value * value);
   else /* add 'state' and square of 'value' together */
      return (state + (value * value));
}

/* SUMSQ COMBINE support function on INTEGER */
mi_integer combine_sumsq(state1, state2)
   mi_integer state1, state2;
{
   /* Return the new partial sum from two parallel partial
    * sums
    */
   return (iter_sumsq(state1, state2));
}
```

Figure 15-16. Aggregate Support Functions for SUMSQ on INTEGER

The following SQL statement registers the SUMSQ user-defined aggregate in the database:

```
CREATE AGGREGATE sumsq
   WITH (ITER = iter_sumsq,
      COMBINE = combine_sumsq);
```

This CREATE AGGREGATE statement lists *only* the aggregate support functions that are required to implement SUMSQ: ITER and COMBINE.

Suppose that the ITER and COMBINE aggregate support functions for the SUMSQ aggregate are compiled and linked into a shared-object module named **sumsq**.

─── **UNIX/Linux Only** ───

On UNIX or Linux, the executable code for the SUMSQ aggregate support functions would be in a shared library named **sumsq.so**.

─── **End of UNIX/Linux Only** ───

Figure 15-17 shows the CREATE FUNCTION statements that register the aggregate support functions for SUMSQ to handle INTEGER aggregate arguments.

```
CREATE FUNCTION iter_sumsq(state INTEGER, one_value INTEGER)
   RETURNS INTEGER
   WITH (HANDLESNULLS)
   EXTERNAL NAME '/usr/udrs/aggs/sums/sumsq.so'
   LANGUAGE C;

CREATE FUNCTION combine_sumsq(state1 INTEGER, state2 INTEGER)
   RETURNS INTEGER
   EXTERNAL NAME '/usr/udrs/aggs/sums/sumsq.so'
   LANGUAGE C;
```

Figure 15-17. Registering the SUMSQ Aggregate Support Functions for INTEGER

For the **tab1** table, which Figure 15-7 on page 15-16 defines, the following query uses the new SUMSQ aggregate function on the INTEGER column, **col3**:

```
SELECT SUMSQ(col3) FROM tab1;
```

With the rows that Figure 15-7 has inserted, the preceding query yields an INTEGER value of 2173. To be able to use SUMSQ on other data types, you need to ensure that the appropriate aggregate support functions exist for this data type.

SQSUM2 User-Defined Aggregate: The SQSUM2 user-defined aggregate is another version of the SQSUM1 aggregate, which Table 15-2 on page 15-17 describes. Its algorithm is the same as SQSUM1:

$$(x_1 + x_2 + \ldots \;)^2$$

where each x_i is one column value; that is, one aggregate argument.

However, the SQSUM2 aggregate takes advantage of the fact that this aggregate has a simple state. Because the database server automatically handles state management, the SQSUM2 aggregate can safely omit the INIT function.

Figure 15-18 shows the aggregate support functions that handle an INTEGER argument for the SQSUM2 user-defined aggregate.

```
/* SQSUM2 ITER support function on INTEGER */
mi_integer iter_sqsum2(state, value, fparam)
   mi_integer state;
   mi_integer value;
   MI_FPARAM *fparam;
{
   /* If 'state' is NULL, this is the first invocation.
    * Just return 'value'.
    */
   if ( mi_fp_argisnull(fparam, 0) )
      return (value);
   else /* add 'state' and 'value' together */
      return (state + value);
}

/* SQSUM2 COMBINE support function on INTEGER */
mi_integer combine_sqsum2(state1, state2)
   mi_integer state1, state2;
{
   /* Return the new partial sum from two parallel partial
    * sums
    */
   return (iter_sqsum2(state1, state2));
}

/* SQSUM2 FINAL support function on INTEGER */
mi_integer final_sqsum2(state)
   mi_integer state;
{
   /* Calculate square of sum */
   state *= state;

   return (state);
}
```

Figure 15-18. Aggregate Support Functions for SQSUM2 on INTEGER

In its first invocation, the ITER function performs the state initialization. It then takes a single aggregate argument and adds it to a partial sum. For aggregates with a simple state, the COMBINE function can be the same as the ITER function. Therefore, this COMBINE function just calls **iter_sumsq2( )** to perform the merge of two partial states.

Tip: The ITER function in Figure 15-18 could use the binary operator **plus( )** to perform the addition. This operator is already defined on the INTEGER data type and therefore would not need to be written or registered. To use **plus( )** in ITER, you would need to ensure that it is defined for the data type on which the SQSUM2 aggregate is defined.

The data type of the aggregate result is also the same as the aggregate state. Therefore, SQSUM2 is a simple binary operator and the FINAL support function is *not* needed to convert the data type of the final state. However, the SQSUM2 aggregate still *does* require a FINAL support function. The SQSUM2 algorithm involves a post-iteration calculation: it must square the final sum to obtain the aggregate return value. The FINAL function performs this final calculation and returns it as the aggregate result for the SQSUM2 aggregate.

Suppose that the ITER, COMBINE, and FINAL aggregate support functions for the SQSUM2 aggregate are compiled and linked into a shared-object module named

sqsum.

Once you have successfully compiled and linked the aggregate support functions,
you can define the SQSUM2 aggregate in the database. Figure 15-19 shows the
CREATE AGGREGATE statement that registers the SQSUM2 user-defined
aggregate. This statement specifies the registered SQL names of the required
aggregate support functions.

```
CREATE AGGREGATE sqsum2
   WITH (ITER = iter_sqsum2,
       COMBINE = combine_sqsum2,
       FINAL = final_sqsum2);
```

Figure 15-19. Registering the SQSUM2 User-Defined Aggregate

Figure 15-20 shows the CREATE FUNCTION statements that register the SQSUM2
aggregate support functions for the aggregate argument of the INTEGER data type.

```
CREATE FUNCTION iter_sqsum2(state INTEGER, one_value INTEGER)
   RETURNS INTEGER
   WITH (HANDLESNULLS)
   EXTERNAL NAME '/usr/udrs/aggs/sums/sqsum.so'
   LANGUAGE C;

CREATE FUNCTION combine_sqsum2(state1 INTEGER, state2 INTEGER)
   RETURNS INTEGER
   EXTERNAL NAME '/usr/udrs/aggs/sums/sqsum.so'
   LANGUAGE C;

CREATE FUNCTION final_sqsum2(state INTEGER)
   RETURNS INTEGER
   EXTERNAL NAME '/usr/udrs/aggs/sums/sqsum.so'
   LANGUAGE C;
```

Figure 15-20. Registering the SQSUM2 Aggregate Support Functions for INTEGER

In Figure 15-20, the CREATE FUNCTION statement that registers the ITER support
function requires the HANDLESNULLS routine modifier because the aggregate
does *not* have an INIT support function.

For the **tab1** table, which Figure 15-7 on page 15-16 defines, the following query
uses the new SQSUM2 aggregate function on the INTEGER column, **col3**:

```
SELECT SQSUM2(col3) FROM tab1;
```

With the rows that Figure 15-7 has inserted, the preceding query yields an
INTEGER value of 10201, which is the same value that the SQSUM1 aggregate
returned for these same rows.

Now, suppose that you want to define the SQSUM2 user-defined aggregate on the
complexnum_t named row type, which Figure 15-5 on page 15-12 defines. This
version of SQSUM2 must have the same aggregate support functions as the version

that handles INTEGER (see Figure 15-19 on page 15-41).

Aggregate Support Function	SQL Function Name	C Function Name
ITER	iter_sqsum2( )	iter_sqsum2_complexnum( )
COMBINE	combine_sqsum2( )	combine_sqsum2_complexnum( )
FINAL	final_sqsum2( )	final_sqsum2_complexnum( )

The following code shows the aggregate support functions that handle a **complexnum_t** named row type as an argument for the SQSUM2 user-defined aggregate:

```
/* SQSUM2 ITER support function for complexnum_t */
MI_ROW *iter_sqsum2_complexnum(state, value, fparam)
   MI_ROW *state;
   MI_ROW *value;
   MI_FPARAM *fparam;
{
   /* Compute the new partial sum using the complex_plus( )
    * function. Put the sum in a new MI_ROW, which
    * complex_plus( ) allocates (and returns a pointer to)
    */
   return (complex_plus(state, value, fparam));
}

/* SQSUM2 COMBINE support function for complexnum_t */
MI_ROW *combine_sqsum2_complexnum(state1, state2, fparam)
   MI_ROW *state1, *state2;
   MI_FPARAM *fparam;
{
   MI_ROW *ret_state;

   ret_state =
      iter2_sqsum2_complexnum(state1, state2, fparam);

   mi_free(state1);
   mi_free(state2);

   return (ret_state);
}

/* SQSUM2 FINAL support function for complexnum_t */
MI_ROW *final_sqsum2_complexnum(state)
   MI_ROW *state;
{
   MI_CONNECTION *conn;
   MI_TYPEID *type_id;
   MI_ROW_DESC *row_desc;

   MI_ROW *ret_row;
   MI_DATUM values[2];
   mi_boolean nulls[2] = {MI_FALSE, MI_FALSE};

   mi_real *real_value, *imag_value;
   mi_integer real_len, imag_len;
   mi_real sqsum_real, sqsum_imag;

   /* Extract complex values from state row structure */
   mi_value_by_name(state, "real_part",
      (MI_DATUM *)&real_value, &real_len);
   mi_value_by_name(state, "imaginary_part",
      (MI_DATUM *)&imag_value, &imag_len);

   /* Calculate square of sum */
```

```
    sqsum_real = (*real_value) * (*real_value);
    sqsum_imag = (*imag_value) * (*imag_value);

    /* Put final result into 'values' array */
    values[0] = (MI_DATUM)&sqsum_real;
    values[1] = (MI_DATUM)&sqsum_imag;

    /* Generate return row type */
    conn = mi_open(NULL, NULL, NULL);
    type_id = mi_typestring_to_id(conn, "complexnum_t");
    row_desc = mi_row_desc_create(type_id);
    ret_row = mi_row_create(conn, row_desc, values, nulls);

    return (ret_row);
}
```

Figure 15-21 shows the CREATE FUNCTION statements that register the SQSUM2 aggregate support functions for an aggregate argument of the **complexnum_t** data type.

```
CREATE FUNCTION iter_sqsum2(state complexnum_t,
      one_value complexnum_t)
   RETURNS complexnum_t
   WITH (HANDLESNULLS)
   EXTERNAL NAME
'/usr/udrs/aggs/sums/sqsum.so(iter_sqsum2_complexnum)'
   LANGUAGE C;

CREATE FUNCTION combine_sqsum2(state1 complexnum_t,
      state2 complexnum_t)
   RETURNS complexnum_t
   EXTERNAL NAME
'/usr/udrs/aggs/sums/sqsum.so(combine_sqsum2_complexnum)'
   LANGUAGE C;

CREATE FUNCTION final_sqsum2(state complexnum_t)
   RETURNS complexnum_t
   EXTERNAL NAME
'/usr/udrs/aggs/sums/sqsum.so(final_sqsum2_complexnum)'
   LANGUAGE C;
```

Figure 15-21. Registering the SQSUM2 Aggregate Support Functions for the complexnum_t Named Row Type

The following query uses the SQSUM2 aggregate function on the **complexnum_t** column, **col2**:

```
SELECT SQSUM2(col2) FROM tab1;
```

With the rows that Figure 15-7 on page 15-16 has inserted, the preceding query yields a **complexnum_t** value of:

```
ROW(817.96, 1204.09)
```

PERCENT_GTR User-Defined Aggregate: Suppose you want to create a user-defined aggregate that determines the percentage of values greater than some user-specified value and returns this percentage as a fixed-point number in the range 0 to 100. The implementation of this UDA uses the following aggregate features:

• Uses a set-up argument to allow the end user to specify the value to compare against

- Uses an opaque-type state to hold the state information and initialize the state in the INIT support function
- Uses a COMBINE function that must do more than just call the ITER support function
- Returns an aggregate result whose data type is different from that of the aggregate argument
- Handles NULL values as aggregate arguments

The PERCENT_GTR user-defined aggregate needs the following state information:
- The user-specified set-up argument
- The current number of values greater than the set-up argument
- The current number of values processed

Therefore, it uses the following C structure, named **percent_state_t**, to hold the aggregate state:

```
typedef struct percent_state
   {
   mi_integer gtr_than;
   mi_integer total_gtr;
   mi_integer total;
   } percent_state_t;
```

Because the size of the **percent_state_t** structure never exceeds the maximum opaque-type size, PERCENT_GTR can use an opaque-type state to hold its aggregate state. The following code shows the INIT aggregate support function that handles an INTEGER argument for the PERCENT_GTR aggregate:

```
/* PERCENT_GTR INIT support function for INTEGER */
percent_state_t *init_percentgtr(dummy_arg, gtr_than, fparam)
   mi_integer dummy_arg;
   mi_integer gtr_than;
   MI_FPARAM *fparam;
{
   percent_state_t *state;

   /* Allocate PER_ROUTINE memory for state and initialize it */
   state = mi_alloc(sizeof(percent_state_t));

   /* Check for a NULL-valued set-up argument */
   if ( mi_fp_argisnull(fparam, 1) )
      state->gtr_than = 0;
   else
      state->gtr_than = gtr_than;
   state->total_gtr = 0;
   state->total = 0;

   return (state);
}
```

This INIT function performs the following tasks:
- Handles a set-up argument

  This set-up argument is the value that the end user specifies so that the aggregate knows which value to compare the aggregate arguments against. If the end user provides a NULL value for the set-up argument, PERCENT_GTR checks for values greater than zero (0).
- Allocates PER_ROUTINE memory for the opaque-type state

  The INIT function does not need to allocate memory for an opaque-type state because the database server can perform the state management. However,

because PERCENT_GTR already requires an INIT function to handle the set-up argument, INIT allocates a PER_ROUTINE **percent_state_t** structure so that it can initialize the opaque-type state.

The following code implements the ITER aggregate support function that handles an INTEGER argument for the PERCENT_GTR aggregate:

```
/* PERCENT_GTR ITER support function for INTEGER */
percent_state_t *iter_percentgtr(curr_state, agg_arg, fparam)
   percent_state_t *curr_state;
   mi_integer agg_arg;
   MI_FPARAM *fparam;
{
   if ( mi_fp_argisnull(fparam, 1) == MI_TRUE )
      agg_arg = 0;

   if ( agg_arg > curr_state->gtr_than )
      curr_state->total_gtr += 1;

   curr_state->total += 1;

   return (curr_state);
}
```

The PERCENT_GTR aggregate is defined to handle NULL values (see Figure 15-22 on page 15-46). This ITER function must check for a possible NULL aggregate argument. The function converts any NULL value to a zero (0) so that the numeric comparison can occur.

The following COMBINE aggregate support function handles an INTEGER argument for the PERCENT_GTR aggregate:

```
/* PERCENT_GTR COMBINE support function for INTEGER */
percent_state_t *combine_percentgtr(state1, state2)
   percent_state_t *state1;
   percent_state_t *state2;
{
   state1->total += state2->total;
   state1->total_gtr += state2->total_gtr;

   mi_free(state2);

   return(state1);
}
```

Because PERCENT_GTR does not have a simple state, its COMBINE function must explicitly perform the merging of two parallel threads, as follows:

• It adds the two partial sums (**total** and **total_gtr**).

• It deallocates the PER_COMMAND memory for the second parallel thread (merging of the two states was done "in-place" in **state1**).

The following code shows the FINAL aggregate support function that handles an INTEGER argument for the PERCENT_GTR aggregate:

```
/* PERCENT_GTR FINAL support function for INTEGER */
mi_decimal *final_percentgtr(final_state)
   percent_state_t *final_state;
{
   mi_double_precision quotient;
   mi_decimal return_val;
   mi_integer ret;

   quotient =
      ((mi_double_precision)(final_state->total_gtr)) /
```

```
          ((mi_double_precision)(final_state->total)) * 100;

   if ( (ret = deccvdbl(quotient, &return_val)) < 0 )
      ret = deccvasc("0.00", 4, &return_val);

   return (&return_val);
}
```

The PERCENT_GTR aggregate returns a data type different from the aggregate state. The FINAL function must convert the final state from the aggregate-state data type (**percent_state_t**) to the aggregate-result data type (DECIMAL).

Once you have successfully compiled and linked the aggregate support functions, you can define the PERCENT_GTR aggregate in the database. For a user-defined aggregate that uses an opaque-type state, this definition includes the following steps:

1. Use CREATE OPAQUE TYPE to register the opaque type that holds the opaque-type state.
2. Use CREATE AGGREGATE to register the aggregate.
3. Use CREATE FUNCTION to register the aggregate support functions.

The PERCENT_GTR aggregate uses a fixed-length opaque type, **percent_state_t**, to hold its opaque-type state. The following CREATE OPAQUE TYPE statement registers this opaque type:

```
CREATE OPAQUE TYPE percent_state_t (INTERNALLENGTH = 12);
```

The INTERNALLENGTH modifier specifies the size of the fixed-length C data structure, **percent_state_t**, that holds the opaque-type state.

Figure 15-22 shows the CREATE AGGREGATE statement that defines the PERCENT_GTR user-defined aggregate. This statement specifies the registered SQL names of the required aggregate support functions. It also includes the HANDLESNULLS modifier to indicate that the PERCENT_GTR aggregate does process NULL values as aggregate arguments. By default, the database server does *not* pass a NULL value to an aggregate.

```
CREATE AGGREGATE percent_gtr
   WITH (INIT = init_percent_gtr,
      ITER = iter_percent_gtr,
      COMBINE = combine_percent_gtr,
      FINAL = final_percent_gtr,
      HANDLESNULLS);
```

Figure 15-22. Registering the PERCENT_GTR User-Defined Aggregate

Suppose that the INIT, ITER, COMBINE, and FINAL aggregate support functions for the PERCENT_GTR aggregate are compiled and linked into a shared-object module named **percent**.

─────────────────── **UNIX/Linux Only** ───────────────────

On UNIX or Linux, the executable code for the PERCENT_GTR aggregate support functions would be in a shared library named **percent.so**.

─────────────────── **End of UNIX/Linux Only** ───────────────────

The following CREATE FUNCTION statements register the PERCENT_GTR aggregate support functions for an aggregate argument of the INTEGER data type:

```
CREATE FUNCTION init_percent_gtr(dummy INTEGER, gtr_val INTEGER)
RETURNING percent_state_t
WITH (HANDLESNULLS)
EXTERNAL NAME '/usr/udrs/aggs/percent/percent.so(init_percentgtr)'
LANGUAGE C;

CREATE FUNCTION iter_percent_gtr(state percent_state_t, one_value INTEGER)
RETURNS percent_state_t
WITH (HANDLESNULLS)
EXTERNAL NAME '/usr/udrs/aggs/percent/percent.so(iter_percentgtr)'
LANGUAGE C;

CREATE FUNCTION combine_percent_gtr(state1 percent_state_t,
    state2 percent_state_t)
RETURNS percent_state_t
WITH (HANDLESNULLS)
EXTERNAL NAME '/usr/udrs/aggs/percent/percent.so(combine_percentgtr)'
LANGUAGE C;

CREATE FUNCTION final_percent_gtr (state percent_state_t)
RETURNS DECIMAL(5,2)
WITH (HANDLESNULLS)
EXTERNAL NAME '/usr/udrs/aggs/percent/percent.so(final_percentgtr)'
LANGUAGE C;
```

These CREATE FUNCTION statements register an SQL name for each of the aggregate support functions that you have written in C. They must *all* include the HANDLESNULLS routine modifier because the PERCENT_GTR aggregate handles NULL values.

The following query uses the PERCENT_GTR aggregate function on the INTEGER column, **col3**, to determine the percentage of values greater than 25:

```
SELECT PERCENT_GTR(col3, 20) FROM tab1;
```

With the rows that Figure 15-7 on page 15-16 has inserted, the preceding query yields a DECIMAL(5,2) value of 33.33 percent: 2 of the 6 values are greater than 20 (24 and 31).

X_PERCENTILE User-Defined Aggregate: Suppose you want to create a user-defined aggregate that calculates the x-percentile for a group of values. The *x-percentile* is the number within the group of values that separates *x* percent of the values below and (100-*x*) percent above. The median is a special case of the x-percentile. It represents the 50th-percentile:

```
X_PERCENTILE(y, 50)
```

That is, the above aggregate returns the value within a sample of *y* values that has an equal number of values (50 percent) above and below it in the sample.

The implementation of this UDA uses the following aggregate features:
- Uses a set-up argument to enable the end user to specify the x-percentile to obtain
- Uses a pointer-valued state to hold the state information, and allocates and initializes the state in the INIT support function
- Uses a COMBINE function that must do more than just call the ITER support function

- Handles NULL values as aggregate arguments, including returning an SQL NULL value if the aggregate argument to return was NULL

The X_PERCENTILe user-defined aggregate needs the following state information:
- The user-specified set-up argument
- The current number of values processed
- The current list of values processed
- The current list of whether the values processed are NULL.

Therefore, X_PERCENTILE uses a C structure named **percentile_state_t** to hold the aggregate state:

```
#define MAX_N 1000

typedef struct percentile_state
   {
   mi_integer percentile;
   mi_integer count;
   mi_integer value_array[MAX_N];
   mi_integer value_is_null[MAX_N];
   } percentile_state_t;
```

Important: The **percentile_state_t** structure stores the number of values processed in an in-memory array within the state. You could also choose to store these values elsewhere, such as in an operating-system file or in a separate location in memory. Each of these locations has advantages and disadvantages. Choose the structure that best fits your application needs.

The size of the **percentile_state_t** structure depends on the number of aggregate arguments stored in the **value_array** array; that is, values less then or equal to the MAX_N constant. On a system with four-byte **mi_integer** values, the size of this structure is:

```
8 + 4(MAX_N)
```

If X_PERCENTILE used an opaque-type state, this structure must be less than the maximum opaque-type size. For systems that have a 32 kilobyte maximum opaque-type size, the X_PERCENTILE aggregate could use an opaque-type state as long as it is called in a query that finds 8190 or fewer rows. If the query finds more than 8190 rows, the state would not fit into an opaque type. To avoid this restriction, X_PERCENTILE implements the aggregate state as a pointer-valued state.

The following code shows the INIT aggregate support function that handles an INTEGER argument for the X_PERCENTILE aggregate:

```
/* X_PERCENTILE INIT support function on INTEGER */
mi_pointer init_xprcnt(dummy, prcntile, fparam)
   mi_integer dummy;
   mi_integer prcntile;
   MI_FPARAM *fparam;
{
   percentile_state_t *state;

   /* Allocate memory for the state from the PER_COMMAND
    * pool
    */
   state = (percentile_state_t *)
      mi_dalloc(sizeof(percentile_state_t), PER_COMMAND);
```

```
    /* Initialize the aggregate state */
    if ( mi_fp_argisnull(fparam, 1) )
       state->percentile = 50; /* median */
    else
       state->percentile = prcntile;
    state->count = 0;

    return ((mi_pointer)state);
}
```

This INIT support function performs the following tasks:

- Handles a set-up argument

  This set-up argument is the value that the end user specifies so that the aggregate can determine the value that has x percent values below and (100-x) percent above. If the end user provides an SQL NULL for the set-up argument, X_PERCENTILE assumes a value of 50 and therefore calculates the median.

- Allocates PER_COMMAND memory for the pointer-valued state

  The database server does not perform state management for pointer-valued states. Therefore, the INIT function *must* allocate the memory for the state. It also assigns the appropriate values to the **percentile_state_t** structure to initialize the state.

The following code implements the ITER aggregate support function that handles an INTEGER argument for the X_PERCENTILE aggregate:

```
/* X_PERCENTILE ITER support function on INTEGER */
mi_pointer iter_xprcnt(state_ptr, value, fparam)
  mi_pointer state_ptr;
  mi_integer value;
  MI_FPARAM *fparam;
{
  mi_integer i, j;
  mi_integer is_null = 0;
  percentile_state_t *state =
     (percentile_state_t *)state_ptr;

  /* Check for NULL-valued 'value' */
  if ( mi_fp_argisnull(fparam, 1) )
     {
     value = 0;
     is_null = 1;
     }

  /* Find position of 'value' in ordered 'value_array' */
  for ( i=0; i < state->count; i++ )
     {
     if ( state->value_array[i] > value )
        break;
     }

  /* Increment number of values (count) */
  ++state->count;

  /* Put value into ordered list of existing values */
  for (j=state->count - 1; j > i; j--)
     {
     state->value_array[j] = state->value_array[j-1];
     state->value_is_null[j] = state->value_is_null[j-1];
     }
  state->value_array[i] = value;
  state->value_is_null[i] = is_null;

return ((mi_pointer)state);
}
```

The ITER support function updates the aggregate state in-place with the following information:

- Increments the number of aggregate arguments processed (**count**)
- Stores the new aggregate argument in increasing sorted order in the **value_array** array
- Stores the is-NULL flag that corresponds to each aggregate argument in its corresponding position in the **value_is_null** array

The ITER function also handles a possible NULL-valued aggregate argument. Because the X_PERCENTILE aggregate is defined to handle NULL values (see Figure 15-23 on page 15-51), the database server calls ITER for NULL-valued aggregate arguments.

The following COMBINE aggregate support function handles an INTEGER argument for the X_PERCENTILE aggregate:

```
/* X_PERCENTILE COMBINE support function on INTEGER */
mi_pointer combine_xprcnt(state1_ptr, state2_ptr)
   mi_pointer state1_ptr, state2_ptr;
{
   mi_integer i;
   percentile_state_t *state1 =
      (percentile_state_t *)state1_ptr;
   percentile_state_t *state2 =
      (percentile_state_t *)state2_ptr;

   /* Merge the two ordered value arrays */
   for ( i=0; i < state2->count; i++ )
      (void) iter_xprcnt(state1_ptr,
         state2->value_array[i]);

   /* Free the PER_COMMAND memory allocated to the state of
    * the 2nd parallel thread (state2). The memory
    * allocated to the state of the 1st parallel thread
    * (state1) holds the updated state. It is in the FINAL
    * support function.
    */
   mi_free(state2);

   return (state1_ptr);
}
```

This COMBINE support function merges two aggregate states, as follows:

- Two ordered lists are merged into a single ordered list.
- Two counts are added together.
- Memory for one of the partial states is freed.
- A pointer to the merged aggregate state is returned.

The following FINAL aggregate support function handles an INTEGER argument for the X_PERCENTILE aggregate:

```
/* X_PERCENTILE FINAL support function on INTEGER */
mi_integer final_xprcnt(state_ptr, fparam)
   mi_pointer state_ptr;
   MI_FPARAM *fparam;
{
   mi_integer index, trunc_int;
   mi_integer x_prcntile;
   percentile_state_t *state =
      (percentile_state_t *)state_ptr;

/* Obtain index position of x-percentile value */
```

```
   trunc_int = (state->count) * (state->percentile);
   index = trunc_int/100;
   if ( (trunc_int % 100) >= 50 )
      index++;

/* Obtain x-percentile value from sorted 'value_array' */
   x_prcntile = state->value_array[index];

/* Check for NULL value so it can be returned as such */
   if ( state->value_is_null[index] )
      mi_fp_setreturnisnull(fparam, 0, MI_TRUE);

/* Free the PER_COMMAND memory allocated to the state */
   mi_free(state);

/* Return retrieved x-percentile value */
   return (x_prcntile);
}
```

This FINAL support function performs the following tasks:

- Calculates the x-percentile for the values in the sorted array

  The FINAL function must obtain the index position for the value that represents the *x*-percentile, where *x* is the percentage that the end user has passed in as a set-up argument.

- Deallocates PER_COMMAND memory for the pointer-valued state

  The database server does *not* perform any state management for pointer-valued states. Therefore, the FINAL function *must* deallocate the PER_COMMAND state memory that the INIT function has allocated.

After you successfully compile and link the aggregate support functions, you can define the PERCENT_GTR aggregate in the database. Figure 15-23 shows the CREATE AGGREGATE statement that defines the X_PERCENTILE user-defined aggregate. This statement specifies the registered SQL names of the required aggregate support functions. It also includes the HANDLESNULLS modifier to indicate that the PERCENT_GTR aggregate does process NULL values as aggregate arguments. By default, the database server does not pass a NULL value to an aggregate.

```
CREATE AGGREGATE x_percentile
   WITH (INIT = init_x_prcntile,
         ITER = iter_x_prcntile,
         COMBINE = combine_x_prcntile,
         FINAL = final_x_prcntile,
         HANDLESNULLS);
```

Figure 15-23. Registering the X_PERCENTILE User-Defined Aggregate

Suppose that the INIT, ITER, COMBINE, and FINAL aggregate support functions for the X_PERCENTILE aggregate are compiled and linked into a shared-object module named **percent**.

─────────────── **UNIX/Linux Only** ───────────────

On UNIX or Linux, the executable code for the X_PERCENTILE aggregate support functions would be in a shared library named **percent.so**.

─────────────── **End of UNIX/Linux Only** ───────────────

The following CREATE FUNCTION statements register the X_PERCENTILE aggregate support functions for an aggregate argument of the INTEGER data type:

```
CREATE FUNCTION init_x_prcntile(dummy INTEGER, x_percent INTEGER)
   RETURNING POINTER
   WITH (HANDLESNULLS)
   EXTERNAL NAME
      '/usr/udrs/aggs/percent/percent.so(init_xprcnt)'
   LANGUAGE C;

CREATE FUNCTION iter_x_prcntile(agg_state POINTER,
      one_value INTEGER)
   RETURNS POINTER
   WITH (HANDLESNULLS)
   EXTERNAL NAME
      '/usr/udrs/aggs/percent/percent.so(iter_xprcnt)'
   LANGUAGE C;

CREATE FUNCTION combine_x_prcntile(agg_state1 POINTER,
      agg_state2 POINTER)
   RETURNS POINTER
   WITH (HANDLESNULLS)
   EXTERNAL NAME
   '/usr/udrs/aggs/percent/percent.so(combine_xprcnt)'
   LANGUAGE C;

CREATE FUNCTION final_x_prcntile(agg_state POINTER)
   RETURNS INTEGER
   WITH (HANDLESNULLS)
   EXTERNAL NAME
      '/usr/udrs/aggs/percent/percent.so(final_xprcnt)'
   LANGUAGE C;
```

These CREATE FUNCTION statements use the SQL data type, POINTER, to indicate that the aggregate support functions accept a generic C pointer and perform their own memory management. They must *all* include the HANDLESNULLS routine modifier because the X_PERCENTILE aggregate handles NULL values.

The following query uses the X_PERCENTILE aggregate function on the INTEGER column, **col3**, to determine the quartile (the 25th percentile) for the values in **col3**:

```
SELECT X_PERCENTILE(col3, 25) FROM tab1;
```

For the **tab1** rows that Figure 15-7 on page 15-16 has inserted, X_PERCENTILE creates the following sorted list for the **col3** values:

```
5, 9, 13, 19, 24, 31
```

Because 25 percent of 6 values is 1.5, X_PERCENTILE obtains the list item that has 2 values (1.5 rounded up to the nearest integer) below it. The preceding query returns 13 as the quartile for **col3**.

Suppose you add the following row to the **tab1** table:

```
INSERT INTO tab1 (7, NULL:complexnum_t, NULL);
```

This INSERT statement adds a NULL value to the **col3** column. Because X_PERCENTILE handles NULLs, the database server calls the X_PERCENTILE aggregate on this new row as well. After this seventh row is inserted, X_PERCENTILE would generate the following sorted list for **col3**:

```
(NULL), 5, 9, 13, 19, 24, 31
```

Twenty-five percent of 7 values is 1.75, so X_PERCENTILE obtains the list item that has 2 (1.75 truncated to the nearest integer) values below it. Now the quartile for **col3** would be 9. If X_PERCENTILE was *not* registered with the HANDLESNULLS modifier, however, the database server would not call X_PERCENTILE for this newest row and the quartile for **col3** would have been 13 (the quartile for 6 rows, even though **col3** actually has 7 rows).

If you called the X_PERCENTILE aggregate with an x-percentile that would return the first value in the list (the NULL value), the FINAL support function uses the DataBlade API function **mi_fp_setreturnisnull( )** to set the aggregate result to NULL. For example, suppose you execute the following query on the **col3**:

```
SELECT X_PERCENTILE(col3, 5) FROM tab1;
```

This query asks for the 5th percentile for the seven values in **col3**. Because 5 percent of 7 values is 0.35, X_PERCENTILE obtains the list item that has zero values (0.35 truncated to the nearest integer) below it. The preceding query returns NULL as the quartile for **col3**. The ITER function has stored NULL values as zeros in the sorted **value_array**. For the FINAL support function to determine when a value of zero indicates a NULL and when it indicates zero, it checks the **value_is_null** array. If the zero indicates a NULL value, FINAL calls the DataBlade API function **mi_fp_setreturnisnull( )** to set the aggregate result to NULL.

Providing UDR-Optimization Functions

The DataBlade API provides support for you to create the following kinds of special-purpose UDRs to optimize UDR performance:

- Tasks that optimize execution of UDRs

  When you write a UDR, you can provide the query optimizer with the following information to help it determine the best query path for queries that contain the UDR.

Filter Optimization	More Information
Parallel Execution	"Creating Parallelizable UDRs" on page 15-61
Cost-of-execution	"Query Cost" on page 15-55

- UDRs that optimize query filters

If your UDR returns a BOOLEAN value (**mi_boolean**), it is called a *Boolean function*. Table 15-5 shows the kinds of Boolean functions that are useful as filters in a query.

Table 15-5. Boolean Functions Useful as Query Filters

Comparison Condition	Operator Symbol	Associated User-Defined Function
Relational operator	=, !=, <>	equal( ), notequal( ), notequal( )
	<, <=	lessthan( ), lessthanorequal( )
	>, >=	greaterthan( ), greaterthanorequal( )
LIKE, MATCHES	None	like( ), matches( )
Boolean function	None	Name of a user-defined function that returns a BOOLEAN value

When you write one of the Boolean functions in Table 15-5, you can also provide the query optimizer with information about how to best evaluate a filter that consists of the Boolean function. You can define the following UDR-optimization functions for Boolean functions.

Filter Optimization	More Information
Negator function	"Creating Negator Functions" on page 15-60
Selectivity	"Query Selectivity" on page 15-54

Writing Selectivity and Cost Functions

The query optimizer uses the selectivity and cost of a query to help select the best query plan. To help the optimizer select the best query plan, you can provide the query optimizer with information about the selectivity and cost of your UDR.

This information is extremely useful for an *expensive UDR*, a UDR that requires a lot of execution time or resources to execute. When the query optimizer can obtain the selectivity and cost for an expensive UDR, it can better determine when to execute the UDR in the query plan.

When you write an expensive UDR, you can indicate the following performance-related characteristics of the UDR to assist the query optimizer in developing a query plan:

- The *cost* of the UDR is a measurement of how expensive the UDR is to run.
- The *selectivity* of the UDR is a percentage of the number of rows that you expect it to return.

This section describes how to create selectivity and cost functions for an expensive UDR. For a general description of how the query optimizer uses cost and selectivity for UDRs, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Tip: The IBM Informix BladeSmith development tool automatically generates C source code for selectivity and cost functions as well as the SQL statements to register these functions. For more information, see the *IBM Informix DataBlade Developers Kit User's Guide*.

Query Selectivity

If your UDR is a *Boolean function*, it can be used as a filter in a query. (For a list of Boolean functions that are useful as query filters, see Table 15-5 on page 15-53.) The query optimizer uses the selectivity of a query to estimate the number of rows that the query will return. The selectivity is a floating-point value between zero (0) and one (1) that represents the percentage of rows for which the query (and each filter in the query) is expected to return a true value.

For a Boolean function likely to be used as a query filter, you can use the following routine modifiers to specify a selectivity for the function.

Selectivity	Routine Modifier
Selectivity is *constant* for every invocation of the Boolean function.	SELCONST = *selectivity_value*
	selectivity_value is a floating-point value between 0 and 1.

Selectivity	Routine Modifier
Selectivity *varies* according to some execution conditions.	SELFUNC = *selectivity_func*
	selectivity_func is the name of a selectivity function that returns a floating-point value between 0 and 1 to indicate the selectivity of the Boolean function.

When the query optimizer needs to determine the selectivity of the Boolean function, it either uses the constant selectivity value or calls the selectivity function, depending whether the Boolean function was registered with the SELCONST or SELFUNC routine modifier.

If you need to calculate the selectivity for a Boolean function at runtime, create a *selectivity function*.

To create a selectivity function:
1. Write a C user-defined function to implement the selectivity function.

   The selectivity function has the following coding requirements:
   - The selectivity function must take the same number of arguments as its companion Boolean function.
   - Each argument of the selectivity function must be declared of type **MI_FUNCARG**.
   - The selectivity function must return the selectivity as a floating-point value (**mi_real** or **mi_double_precision**) that is between zero and one.
2. Register the selectivity function with the CREATE FUNCTION statement.

   The SQL selectivity function has the following registration requirements:
   - The selectivity function must take the same number of arguments as its companion Boolean function.
   - Each argument of the selectivity function must be declared of type SELFUNCARG.
   - The selectivity function must return the selectivity as a FLOAT value.
3. Associate the selectivity function with its companion UDR with the SELFUNC routine modifier when you register the companion UDR.

Query Cost

If your UDR requires a lot of system resources (such as a large number of disk accesses or network accesses), you can define a cost-of-execution for the UDR. The query optimizer uses the cost value to determine the total cost of executing a query.

Tip: You can define a cost for either a user-defined procedure or a user-defined function. However, user-defined functions can appear in queries because they return a value. Because user-defined procedures do not appear in queries, the query optimizer is not usually concerned with their cost.

When you register a UDR, you can specify its cost with one of the following routine modifiers.

Cost	Routine Modifier
Cost is *constant* for every invocation of the UDR.	PERCALL_COST = *cost_value*
	cost_value is a floating-point value between 0 and 1.
Cost *varies* according to some execution conditions.	COSTFUNC = *cost_func*
	cost_func is the name of a cost UDR that returns a floating-point value between 0 and 1 to indicate the cost of the UDR.

When the query optimizer needs to determine the cost of the UDR, it either uses the constant cost value or calls the cost function, depending whether the UDR was registered with the PERCALL_COST or COSTFUNC routine modifier.

If you need to calculate the cost for a UDR at runtime, create a *cost function*.

To create a cost function:

1. Write a C user-defined function to implement the cost function.

   The cost function has the following coding requirements:

   - The cost function must take the same number of arguments as its companion UDR.
   - Each argument of the cost function must be declared of type **MI_FUNCARG**.

     For more information, see "MI_FUNCARG Data Type" on page 15-56.
   - The cost function must return the cost as an integer value (**mi_integer** or **mi_smallint**).

2. Register the cost function with the CREATE FUNCTION statement.

   The SQL cost function has the following registration requirements:

   - The cost function must take the same number of arguments as its companion UDR.
   - Each argument of the cost function must be declared of type SELFUNCARG.
   - The cost function must return the cost as an INTEGER or SMALLINT value.

3. Associate the cost function with its companion UDR with the COSTFUNC routine modifier when you register the companion UDR.

MI_FUNCARG Data Type

The **MI_FUNCARG** data type is an Informix-defined opaque type that contains information about the companion UDR of a selectivity or cost function. Selectivity and cost functions both have the same number of arguments as their companion UDRs. To calculate selectivity or cost effectively, however, your user-defined function might need to know additional information about the context in which the UDR was called. The DataBlade API provides this contextual information in the **MI_FUNCARG** structure.

Each argument of a cost or selectivity function is of type **MI_FUNCARG**. The DataBlade API provides accessor functions for the **MI_FUNCARG** structure. You can use any of these functions to extract information about the companion-UDR arguments from the selectivity or cost function. Table 15-6 lists the DataBlade API accessor functions that obtain information from the **MI_FUNCARG** structure.

Table 15-6. Argument Information in the MI_FUNCARG Structure

MI_FUNCARG Information	DataBlade API Function
Information about the companion UDR:	
The identifier of the companion UDR	**mi_funcarg_get_routine_id( )**
The name of the companion UDR	**mi_funcarg_get_routine_name( )**
General companion-UDR argument information:	
Whether the companion-UDR argument is a column, constant, or parameter	**mi_funcarg_get_argtype( )**
The data type of companion-UDR argument	**mi_funcarg_get_datatype( )**
The length of the companion-UDR argument	**mi_funcarg_get_datalen( )**
Constant argument (MI_FUNCARG_CONSTANT):	
The constant value of the companion-UDR argument	**mi_funcarg_get_constant( )**
Whether the value of the companion-UDR argument is the SQL NULL value	**mi_funcarg_isnull( )**
Column-value argument (MI_FUNCARG_COLUMN):	
The column number of the column associated with the companion-UDR argument	**mi_funcarg_get_colno( )**
The table identifier of the table that contains the column associated with the companion-UDR argument	**mi_funcarg_get_tabid( )**
The distribution information for the column associated with the companion-UDR argument	**mi_funcarg_get_distrib( )**

Table 15-7 lists the DataBlade API accessor functions that obtain general information about a companion UDR from the **MI_FUNCARG** structure.

Table 15-7. General Companion-UDR Information in the MI_FUNCARG Structure

MI_FUNCARG Information	DataBlade API Function
Information about the companion UDR:	
The identifier of the companion UDR	**mi_funcarg_get_routine_id( )**
The name of the companion UDR	**mi_funcarg_get_routine_name( )**
General companion-UDR argument information:	
Whether the companion-UDR argument is a column, constant, or parameter	**mi_funcarg_get_argtype( )**
The data type of companion-UDR argument	**mi_funcarg_get_datatype( )**
The length of the companion-UDR argument	**mi_funcarg_get_datalen( )**

Important: To a DataBlade API module, the **MI_FUNCARG** data type is an opaque data type. Do not access its internal fields directly. The internal structure of this opaque data type may change in future releases. Therefore, to create portable code, always use the accessor functions in Table 15-6 to obtain values in this data type.

The **MI_FUNCARG** structure categorizes each argument of the companion UDR arguments. The **MI_FUNCARG** data type identifies the following kinds of arguments in the companion UDR.

Companion-UDR Argument Type	Argument-Type Constant
Argument is a constant value	MI_FUNCARG_CONSTANT
Argument is a column value	MI_FUNCARG_COLUMN
Argument is a parameter	MI_FUNCARG_PARAM

In addition to the general companion-UDR information that the functions in Table 15-7 obtain, you can also obtain information about the arguments themselves. The information that you can obtain depends on the particular category of the companion-UDR argument. Table 15-8 lists the DataBlade API accessor functions that obtain argument information from the **MI_FUNCARG** structure.

Table 15-8. Argument Information in the MI_FUNCARG Structure

MI_FUNCARG Information	DataBlade API Function
Constant argument (MI_FUNCARG_CONSTANT):	
The constant value of the companion-UDR argument	**mi_funcarg_get_constant( )**
Determines if the value of the companion-UDR argument is the SQL NULL value	**mi_funcarg_isnull( )**
Column-value argument (MI_FUNCARG_COLUMN):	
The column number of the column associated with the companion-UDR argument	**mi_funcarg_get_colno( )**
The table identifier of the table that contains the column associated with the companion-UDR argument	**mi_funcarg_get_tabid( )**
The distribution information for the column associated with the companion-UDR argument	**mi_funcarg_get_distrib( )**

For example, you can write the following query:

```
SELECT * FROM tab1 WHERE meets_cost(tab1.int_col, 20) ...;
```

Suppose you register the **meets_cost( )** function with a selectivity function named **meets_cost_selfunc( )**, as follows:

```
CREATE FUNCTION meets_cost(col INTEGER, value INTEGER)
   RETURNS BOOLEAN
   WITH (....SELFUNC=meets_cost_selfunc....)
   EXTERNAL NAME '......'
   LANGUAGE C;
```

Because the **meets_cost( )** function returns a BOOLEAN value, you can write a selectivity function for the function. You write **meets_cost_selfunc( )** so that it expects two arguments of the data type **MI_FUNCARG**. The following table shows what different **MI_FUNCARG** accessor functions return when you invoke them for each of the arguments of the **meets_cost( )** function.

DataBlade API Function	Argument 1	Argument 2
mi_funcarg_get_argtype( )	MI_FUNCARG_COLUMN	MI_FUNCARG_CONSTANT
mi_funcarg_get_datatype( )	Type identifier for data type of **tab1.int_col**	Type identifier for INTEGER data type
mi_funcarg_get_datalen( )	Length of **tab1.int_col**	Length of INTEGER
mi_funcarg_get_tabid( )	Table identifier of **tab1**	Undefined
mi_funcarg_get_colno( )	Column number of **int_col**	Undefined

DataBlade API Function	Argument 1	Argument 2
mi_funcarg_isnull( )	FALSE	FALSE
mi_funcarg_get_constant( )	Undefined	An **MI_DATUM** structure that holds the value of 20

Obtaining Information About Constant Arguments

When the companion UDR receives an argument that is a constant, you can obtain the following information about this constant from within the cost or selectivity function.

MI_FUNCARG Information	DataBlade API Function
The constant value of the companion-UDR argument	**mi_funcarg_get_constant( )**
Determines if the value of the companion-UDR argument is the SQL NULL value	**mi_funcarg_isnull( )**

Obtaining Information About Column Arguments

When the companion UDR receives an argument that is a column, you can obtain the following information about this column from the associated **MI_FUNCARG** argument of the cost or selectivity function.

MI_FUNCARG Information	DataBlade API Function
The column number of the column associated with the companion-UDR argument	**mi_funcarg_get_colno( )**
The table identifier of the table that contains the column associated with the companion-UDR argument	**mi_funcarg_get_tabid( )**
The data-distribution information for the column associated with the companion-UDR argument	**mi_funcarg_get_distrib( )**

The column number and table identifier are useful in a selectivity or cost function to obtain additional information about the column argument from the **syscolumns** or **systables** system catalog tables. The data distribution is useful if the determination of selectivity or cost depends on how the column values are distributed; that is, how many values in each range of values. Data distributions only make sense for data types that can be ordered.

The **mi_funcarg_get_distrib( )** function obtains the contents of the **encdat** column of the **sysdistrib** system catalog table. The **encdat** column stores the data distribution for the column associated with the companion-UDR argument, as follows:

- For columns of built-in data types, the data distribution is stored as an ASCII histogram, with a predetermined number of ordered bins that hold the sorted column values.
- For columns of user-defined data types, this data distribution is in a user-defined statistics structure.

The **mi_funcarg_get_distrib( )** function returns the data distribution in an **mi_bitvarying** structure as an **mi_statret** structure. The **mi_statret** structure can store the data distribution either directly in the structure (in the **statdata.buffer** field) or in a smart large object (in the **statdata.mr** field).

For more information about user-defined statistics, see "Providing Statistics Data for a Column" on page 16-40.

Creating Negator Functions

A *negator function* is a special UDR that is associated with a Boolean user-defined function. It evaluates the Boolean NOT condition for its associated user-defined function. For example, if an expression in a WHERE clause invokes a Boolean user-defined function (*UDR-Boolfunc*), the SQL optimizer can decide whether it is more efficient to replace occurrences of the expression

NOT (*UDR-Boolfunc*)

with a call to the negator function (*UDR-func-negator*).

To implement a negator function with a C user-defined function:

1. Declare the negator function so that its parameters are exactly the same as its associated user-defined function and its return value is BOOLEAN (**mi_boolean**).
2. Within the negator function, perform the tasks to evaluate the NOT condition of the associated Boolean user-defined function.
3. Register the negator function as a user-defined function with the CREATE FUNCTION statement.
4. Associate the Boolean user-defined function and its negator function when you register the user-defined function.

   Specify the name of the negator function with the NEGATOR routine modifier in the CREATE FUNCTION statement that registers the user-defined function.

For more information about Boolean user-defined functions and negator functions, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*. For information on how to determine if a user-defined function has a negator function, see "Checking for a Negator Function" on page 9-26.

Creating Commutator Functions

A *commutator function* is a special UDR that is associated with a user-defined function. A UDR is commutator of another user-defined function if either of the following statements is true:

- The UDR takes the same arguments as its associated user-defined function but in opposite order.
- The UDR returns the same result as the associated user-defined function.

For example, the **lessthan( )** and **greaterthanorequal( )** functions are commutators of one another because the following two expressions yield the same result:

```
a < b
b >= a
```

In the following SELECT statement, the optimizer can choose whether it is more cost effective to execute **lessthan(a, b)** or **greaterthanorequal(b, a)** in the WHERE clause:

```
SELECT * FROM tab1 WHERE lessthan(a, b);
```

The optimizer can choose to invoke the function **greaterthanorequal(b, a)** if there is no index on **lessthan( )** and there exists an index on **greaterthanorequal( )**.

To implement a commutator function with a C user-defined function:

1. Declare the commutator function so that its parameters are in the reverse order as its associated user-defined function and its return value is the same as its user-defined function.

2. Within the commutator function, perform the tasks to evaluate the commutable operation of the associated Boolean user-defined function.

3. Register the commutator function as a user-defined function with the CREATE FUNCTION statement.

4. Associate the user-defined function and its commutator function when you register the user-defined function.

   Specify the name of the commutator function with the COMMUTATOR routine modifier in the CREATE FUNCTION statement that registers the user-defined function.

The following CREATE FUNCTION statements register the **commute_func1( )** and **func1( )** user-defined functions:

```
CREATE FUNCTION commute_func1(b CHAR(20), a INTEGER)
RETURNS INTEGER
EXTERNAL NAME '/usr/local/lib/udrs/udrs.so'
LANGUAGE C;

CREATE FUNCTION func1(a INTEGER, b CHAR(20))
RETURNS INTEGER
WITH (COMMUTATOR = commute_func1)
EXTERNAL NAME '/usr/local/lib/udrs/udrs.so'
LANGUAGE C;
```

Important: The generic B-tree secondary-access method does not check for commutator functions registered with the COMMUTATOR routine modifier. Instead, it performs its own internal optimization for commutable operations. However, commutator functions registered with COMMUTATOR are used by the R-tree secondary-access method when UDRs occur in fragmentation expressions.

For more information about commutator functions, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*. For information on how to determine if a user-defined function has a commutator function, see "Checking for a Commutator Function" on page 9-26.

Creating Parallelizable UDRs

The Parallel Database Query (PDQ) feature allows the database server to run a single SQL statement in parallel. When you send a query to the database server, it breaks your request into a set of discrete subqueries, each of which can be assigned to a different CPU virtual processor. A *parallelizable query* is a query that can be executed in parallel. PDQ is especially effective when your tables are fragmented and your server computer has more than one CPU.

A *parallelizable UDR* is a C UDR that can be executed in parallel when it is invoked within a parallelizable query. If you write your C UDR to be parallelizable, it can be executed in parallel when the query that invokes it is executed in parallel. That is, the C UDR can execute on subsets of table data just as the query itself can. A query that invokes a nonparallelizable UDR can still run in parallel. However, the calls to the UDR do *not* run in parallel. Similarly, prepared queries that invoke a parallelizable query do not run the UDR in parallel.

To create a parallelizable C UDR:

1. Write the C UDR so that it does not call any DataBlade API functions that are non-PDQ-threadsafe.
2. Register the C UDR with the PARALLELIZABLE routine modifier.
3. Execute the parallelized C UDR, once in each scan thread of the parallelized query.
4. Debug the parallelized C UDR.

The following subsections describe these steps in detail.

Writing the Parallelizable UDR

To write a parallelizable C UDR, you must ensure that the UDR does *not* include any calls to the non-PDQ-threadsafe DataBlade API functions that Table 15-9 lists.

Table 15-9. Non-PDQ-Threadsafe DataBlade API Functions

Category of Non-PDQ-Threadsafe Function	DataBlade API Function
Statement processing:	
Statement execution A parallelizable UDR cannot parse an SQL statement.	**mi_exec( )**, **mi_prepare( )**
Current-statement processing No current statement exists in a parallelizable UDR. Therefore, these functions are not useful.	**mi_binary_query( )**, **mi_command_is_finished( )**, **mi_get_result( )**, **mi_get_row_desc_without_row( )**, **mi_next_row( )**, **mi_query_finish( )**, **mi_query_interrupt( )**, **mi_result_command_name( )**, **mi_result_row_count( )**, **mi_value( )**, **mi_value_by_name( )**
Prepared statements No prepared statement exists because you cannot prepare one in a parallelizable UDR. Therefore, these functions are not useful.	**mi_close_statement( )**, **mi_drop_prepared_statement( )**, **mi_exec_prepared_statement( )**, **mi_fetch_statement( )**, **mi_get_statement_row_desc( )**, **mi_open_prepared_statement( )**, **mi_statement_command_name( )** All input-parameter accessor functions: **mi_parameter_*** (see Table 8-5 on page 8-15)
Transfer of data Even though these type-transfer functions are PDQ-threadsafe, they are usually called within the send and receive support functions of an opaque type and are likely to be called during statement processing.	All type-transfer functions: **mi_get_***, **mi_put_*** (see Table 16-4 on page 16-21)
Other	**mi_current_command_name( )**
Save-set handling	All save-set functions: **mi_save_set_*** (see Table 8-8 on page 8-60)
Complex-type (collections and row types) handling:	

Table 15-9. Non-PDQ-Threadsafe DataBlade API Functions (continued)

Category of Non-PDQ-Threadsafe Function	DataBlade API Function
Collection processing	All collection functions: **mi_collection_*** (see "Collections" on page 5-2)
Row-type processing	**mi_get_row_desc( )**, **mi_get_row_desc_from_type_desc( )**, **mi_get_row_desc_without_row( )**, **mi_get_statement_row_desc( )** **mi_row_create( )**, **mi_row_free( )**, **mi_row_desc_create( )**, **mi_row_desc_free( )**
Complex-type processing	Type-descriptor accessor functions if they access a complex type: **mi_type_*** (see Table 2-1 on page 2-3) Column functions if they access a complex type: **mi_column_*** (see Table 5-3 on page 5-30)
Operating-system file access	All file-access functions: **mi_file_*** (see Table 13-7 on page 13-52)
Tracing:	
Even though the files listed here are *not* PDQ-threadsafe, you can include most statements that generate trace output in a parallelizable UDR.	**mi_tracefile_set( )**, **mi_tracelevel_set( )** GL_DPRINTF
Miscellaneous	**mi_get_connection_option( )**, **mi_get_database_info( )**, **mi_get_session_connection( )**, **mi_get_type_source_type( )**

A parallelizable C UDR cannot call (either explicitly or implicitly) any of the DataBlade API functions in Table 15-9. If you attempt to run a UDR that contains a non-PDQ-threadsafe function in parallel, the database server generates an error. If your UDR must call one of the functions in Table 15-9, it cannot be parallelizable.

Keep in mind the following considerations when you write a UDR to be parallelizable:

- For a UDR that operates on an opaque type to be parallelizable, all support functions of the opaque type must be parallelizable.
- A UDR that operates on complex data types *cannot* be parallelizable.
- A UDR can be parallelizable whether it runs in the CPU VP or a user-defined VP.
- A UDR that acts as a functional index *cannot* be parallelizable.
- A UDR that is parallelizable cannot call a UDR that is *not* parallelizable (either explicitly or with the Fastpath interface).

Registering the Parallelizable UDR

When you register a UDR with the PARALLELIZABLE routine modifier, you tell the database server that the UDR was written according to the guidelines in "Writing the Parallelizable UDR" on page 15-62. That is, the UDR does not call any DataBlade API functions that are non-PDQ-threadsafe. However, registering the UDR with the PARALLELIZABLE modifier does *not* guarantee that every

invocation of the UDR executes in parallel. The decision whether to parallelize a query and any accompanying UDRs is made when the query is parsed and optimized.

Executing the Parallelizable UDR

When a query with a parallelizable UDR executes in parallel, each routine instance might have more than one routine sequence. For a parallelized UDR, the routine manager creates a routine sequence for each scan thread of the query.

For example, suppose you have the following query:

```
SELECT a_func(x)
FROM table1
WHERE a_func(y) > 7;
```

Suppose also that the **table1** table in the preceding query is fragmented into four fragments and the **a_func( )** user-defined function was registered with the PARALLELIZABLE routine modifier. When this query executes in serial, it contains two routine instances (one in the select list and one in the WHERE clause) and two routine sequences. However, when this query executes in parallel over **table1**, it still contains two routine instances but it now has six routine sequences:

- One routine sequence for the primary thread to execute **a_func( )** in the select list.
- Five routine sequences for **a_func( )** in the WHERE clause:
  - One routine sequence for the primary thread
  - Four routine sequences for secondary PDQ threads, one for each fragment in the table

The **MI_FPARAM** structure holds the information of the routine sequence. Therefore, the routine manager allocates an **MI_FPARAM** structure for each scan thread of the parallelized query. All invocations of the UDR that execute in the scan thread can share the information in an **MI_FPARAM** structure. However, UDRs in different scan threads cannot share **MI_FPARAM** information across scan threads.

Tip: The DataBlade API also supports memory locking for a parallelizable UDR that shares data with other UDRs or with multiple instances of the same routine. Memory locking allows the UDR to implement concurrency control on its data; however, the memory-locking feature is an advanced feature of the DataBlade API. For more information on the memory-locking feature, see "Handling Concurrency Issues" on page 14-27.

For more information about how the routine manager creates a routine sequence, see "Creating the Routine Sequence" on page 12-22.

Debugging the Parallelizable UDR

You can use the SQL statement SET EXPLAIN to determine whether a parallelizable query is actually being executed in parallel. The SET EXPLAIN statement executes when the database server optimizes a statement. It creates a file that contains:

- A copy of the SQL statement
- The plan of execution that the optimizer has chosen
- An estimate of the amount of work

For more information on SET EXPLAIN, see its description in the *IBM Informix Guide to SQL: Syntax* and your *IBM Informix Performance Guide*.

The following **onstat** options are useful to track execution of parallel activities:

- The **-g ath** option shows the session thread and any additional scan threads for each fragment that is scanned for a statement that is running in parallel. You can use the **-g ses** option to help find the relationship between the threads.
- The **-g stk** option dumps the stack of a specified thread. This option can be helpful in tracing exactly what the thread is doing.

For more information on the **onstat** utility, see the *IBM Informix Administrator's Reference*.

Chapter 16. Extending Data Types

In This Chapter

This chapter describes the following ways to extend data types with C user-defined routines (UDRs):

- Create an opaque data type with the C language
- Create a distinct data type
- Write operator-class support functions
- Write optimization functions, including selectivity functions, cost functions, negator functions, and user-defined statistics functions

Tip: For general information about the creation of an opaque type and its support routines, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Creating an Opaque Data Type

This section describes how to design and write an opaque data type.

To create an opaque data type:

1. Design the opaque data type, including its external and internal representations.
2. Write the opaque-type support functions.
3. Take special measures if the opaque type is for multirepresentational data.

Tip: The IBM Informix BladeSmith development tool, which is part of the Informix
  DataBlade Developers Kit, automatically generates C source code for the
  support routines of an opaque type as well as the SQL statements to register
  the opaque type. For more information, see the *IBM Informix DataBlade
  Developers Kit User's Guide*.

Designing an Opaque Data Type

As with most data types, an opaque data type can have two representations for its
data:

- The *external* representation, which is a text or binary representation of the
  opaque-type data
- The *internal* representation, which is the internal structure stored on disk

To design an opaque data type, you must determine these representations for the
opaque-type data.

Determining External Representation

The external representation of an opaque data type is a character string. This string
is the *literal value* for the opaque-type data. A literal value can appear in SQL
statements most anywhere that the binary value can appear. For your opaque-type
data to be valid as a literal value in SQL statements, you must define its external
representation. It is important that the external representation be reasonably
intuitive and easy to enter.

Tip: The external representation of an opaque data type is its ASCII representation.

Suppose you need to create an opaque type that holds information about a circle.
You could create the external representation that Figure 16-1 shows for this circle.
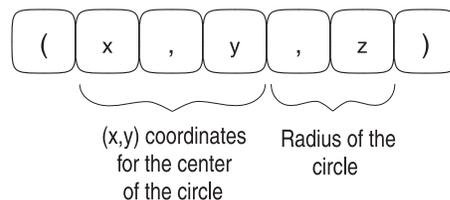
Figure 16-1. External Representation of the circle Opaque Data Type

With the external representation in Figure 16-1, an INSERT statement can specify a
literal value for a column of type **circle** with the following format:

```
INSERT INTO tab1 (id_col, circle_col) VALUES (1, "(2, 3, 9)");
```

Similarly, when an opaque type has an external representation, a client application
such as DB–Access (which displays results as character data) can display a
retrieved opaque-type value as part of the output of the following query:

```
SELECT circle_col FROM tab1 WHERE id_col = 1;
```

In DB–Access, the results of this query would display as follows:

```
(2, 3, 9)
```

Tip: The external representation of an opaque data type is handled by its input
  and output support functions. For more information, see "Input and Output
  Support Functions" on page 16-11.

Determining Internal Representation

The internal representation of an opaque data type is a C data structure that holds the information that the opaque type needs. The internal representation of an opaque type that is stored in a database is called its *server internal representation*. Inside this internal C structure, use the platform-independent DataBlade API data types (such as **mi_integer** and **mi_real**) to improve the portability of the opaque data type.

Tip: The internal representation of an opaque data type is a binary format that might not match the external binary format surfaced to the client.

For example, Figure 16-2 shows the **circle_t** data structure, which holds the values for the **circle** opaque data type.

```
typedef struct
   {
   mi_double_precision    x;
   mi_double_precision    y;
   } point_t;

typedef struct
   {
   point_t                center;
   mi_double_precision    radius;
   } circle_t;
```

Figure 16-2. Internal Representation of the circle Opaque Data Type

The CREATE OPAQUE TYPE statement uniquely names the opaque data type. It is recommended that you develop a unique prefix for the name of an opaque data type. If your DataBlade module uses a prefix, such as **USR**, you could begin the names of opaque types with this prefix. For example, you might use the prefix **USR** on all database objects that your DataBlade module creates. The preceding **circle_t** opaque type could be named **USR_circle_t** to ensure that it does not conflict with opaque types that other DataBlade modules might create.

You register the opaque data type with the CREATE OPAQUE TYPE statement, which stores information about the opaque type in the **sysxtdtypes** system catalog table. When you register an opaque data type, you provide the following information about the internal representation of an opaque type:

- The final size of the new opaque data type
- How the opaque data type should be aligned in memory
- How the opaque data type should be passed in an **MI_DATUM** structure

Determining the Size of an Opaque Type: To save space in the database, you should lay out the internal representation of the opaque type as compactly as possible. The database server stores values in its internal representation, so any C-language structure with padding between entries consumes unnecessary space. You must also decide whether your opaque data type is to be of fixed length or varying length. The following sections briefly describe each of these kinds of opaque types.

Fixed-Length Opaque Data Type: If the C structure that holds your opaque type is *always* the same size, regardless of the data it holds, you can declare the opaque type as a *fixed-length opaque type*. You tell the database server that an opaque type is fixed length when you register the opaque type. In the CREATE OPAQUE TYPE

statement, you must include the INTERNALLENGTH modifier to specify the fixed size of the C structure. The database server stores the value of the INTERNALLENGTH modifier in the **length** column of the **sysxtdtypes** system catalog table.

The **circle_t** C structure (which Figure 16-2 on page 16-3 defines) is a fixed-length structure because all of its member fields have a constant size. Therefore, the following CREATE OPAQUE TYPE statement registers a fixed-length opaque type named **circle** for the **circle_t** structure:

```
CREATE OPAQUE TYPE circle (INTERNALLENGTH = 24);
```

The size of a fixed-length opaque data type must match the value that the C-language **sizeof** directive returns for the C structure. On most compilers, the **sizeof( )** directive performs cast promotion to the nearest four-byte size to ensure that the pointer match on arrays of structures works correctly. However, you do not need to round up for the size of a fixed-length opaque data type. Instead you can specify alignment for the opaque data type with the ALIGNMENT modifier. For more information, see "Specifying the Memory Alignment of an Opaque Type" on page 16-6.

Important: The routine manager does perform cast promotion on argument values smaller than the size of the **MI_DATUM** data type when it pushes routine arguments onto the stack. On some platforms, small values can create problems with pointer matching. For more information, see "Pushing Arguments Onto the Stack" on page 12-22.

The size of the fixed-length opaque type determines the passing mechanism for the opaque type. For more information, see "Determining the Passing Mechanism for an Opaque Type" on page 16-7.

You can obtain information about support functions for the **circle** fixed-length opaque type in "Writing Opaque-Type Support Functions" on page 16-8. The following table lists the **circle** support functions that this section declares.

Support Function for circle Opaque Type	Where to Find Declaration
Input	Figure 16-6 on page 16-13
Output	Figure 16-9 on page 16-14
Receive	Figure 16-12 on page 16-18
Send	Figure 16-15 on page 16-20
Import	Figure 16-18 on page 16-24
Export	Figure 16-21 on page 16-27
Importbin	Figure 16-24 on page 16-30
Exportbin	Figure 16-27 on page 16-32

Varying-Length Opaque Data Type: If the C structure that holds your opaque type can vary in size depending on the data it holds, you must declare the opaque type as a *varying-length opaque type*. The opaque type can contain character strings. Each instance of the opaque type can contain a character string with a different size. When you define the internal representation of a varying-length opaque, make sure that only the *last* member of the C structure is of varying size.

Figure 16-3 shows the internal representation for a varying-length opaque data type named **image**.

```
typedef struct
   {
   mi_integer        img_id;
   mi_integer        img_thresh_trck;
   mi_integer        img_thresh;
   mi_date           img_date;
   mi_integer        img_flags;
   mi_lvarchar    img_data;
   } image_t;
```

Figure 16-3. Internal Representation for the image Opaque Data Type

You tell the database server that an opaque type is varying length when you register the opaque type. In the CREATE OPAQUE TYPE statement, you must include the INTERNALLENGTH modifier with the VARIABLE keyword.

The CREATE OPAQUE TYPE statement in Figure 16-4 registers the **image** opaque type (which Figure 16-3 on page 16-5 defines) as a varying-length opaque type.

```
CREATE OPAQUE TYPE image
   (INTERNALLENGTH = VARIABLE);
```

Figure 16-4. Registration of the image Opaque Data Type

The database server stores the value of the INTERNALLENGTH modifier in the **length** column of the **sysxtdtypes** system catalog table. For varying-length opaque types, this column holds a value of zero (0).

You can obtain information about support functions for the **image** varying-length opaque type in "Writing Opaque-Type Support Functions" on page 16-8. The following table lists the **image** support functions that this section declares.

Support Function for image Opaque Type	Where to Find Declaration
Input	Figure 16-8 on page 16-13
Output	Figure 16-11 on page 16-15
Receive	Figure 16-14 on page 16-18
Send	Figure 16-17 on page 16-20
Import	Figure 16-20 on page 16-25
Export	Figure 16-23 on page 16-27
Importbin	Figure 16-26 on page 16-31
Exportbin	Figure 16-29 on page 16-33

The database server requires you to store the C data structure for a varying-length opaque type in an **mi_lvarchar** structure. To store varying-length data in the **mi_lvarchar** structure, you need to code support functions. The size limitations of a varying-length structure apply to a varying-length opaque type as follows:

- By default, the maximum size for a varying-length opaque type is two kilobytes.
- You specify a different maximum size for a varying-length opaque type when you register the opaque type.

In the CREATE OPAQUE TYPE statement, use the MAXLEN modifier. You can specify a maximum length of up to 32 kilobytes. The database server stores the value of the MAXLEN modifier in the **maxlen** column of the **sysxtdtypes** system catalog table.

For example, the following CREATE OPAQUE TYPE statement defines a varying-length opaque type named **var_type** whose maximum size is four kilobytes:

```
CREATE OPAQUE TYPE var_type
    (INTERNALLENGTH=VARIABLE, MAXLEN=4096);
```

Because the database server uses **mi_lvarchar** to transfer varying-length data, the passing mechanism for a varying-length opaque type is always by reference. For more information, see "Determining the Passing Mechanism for an Opaque Type" on page 16-7.

Specifying the Memory Alignment of an Opaque Type: When the database server passes an opaque data type to a UDR, it aligns the data on a certain byte boundary. By default, the database server uses a four-byte alignment for the internal representation of an opaque type. Four bytes is the standard alignment for 32-bit platforms.

64-bit

On 64-bit platforms, alignment should usually be eight bytes.

End of 64-bit

You can specify a different memory-alignment requirement for your opaque type with the ALIGNMENT modifier of the CREATE OPAQUE TYPE statement. The database server stores the value of the ALIGNMENT modifier in the **align** column of the **sysxtdtypes** system catalog table.

Actual alignment requirements depend on the C definition of the opaque type and on the system (hardware and compiler) on which the opaque data type is compiled. The following table summarizes valid alignment values for some C data types.

Value for ALIGNMENT Modifier	Meaning	Purpose
1	Align structure on one-byte boundary	Structures that begin with one-byte quantities
2	Align structure on two-byte boundary	Structures that begin with two-byte quantities, such as **mi_unsigned_smallint**
4 (default)	Align structure on four-byte boundary	Structures that begin with four-byte quantities, such as **mi_real** or **mi_unsigned_integer**
8	Align structure on eight-byte boundary	Structures that contain members of the **mi_double_precision** data type

Arrays of a data type must follow the same alignment restrictions as the data type itself. However, structures that begin with single-byte characters (such as **mi_boolean** or **mi_char**) can be aligned anywhere.

When you obtain aligned data for an opaque data type from a varying-length structure, use the **mi_get_vardata_align( )** function. Make sure that the *align* argument of **mi_get_vardata_align( )** matches the value of the **align** column in the **sysxtdtypes** system catalog table for the opaque type. For example, the **mi_double_precision** data type is aligned on an eight-byte boundary. If an opaque type contains an array of **mi_double_precision** values, use **mi_get_vardata_align( )** with an *align* value of 8 to access the data portion of the **mi_double_precision** array.

The following call to **mi_get_vardata_align( )** obtains data that is aligned on eight-byte boundaries from the **var_struc** varying-length structure:

```
opaque_type_t *buff;
mi_lvarchar *var_struc;
...
buff = (opaque_type_t *)mi_get_vardata_align(var_struc, 8);
```

Determining the Passing Mechanism for an Opaque Type: The way that the DataBlade API passes the internal representation of an opaque type in an **MI_DATUM** structure depends on the kind of opaque type, as follows:

- For fixed-length opaque types, the contents of the **MI_DATUM** structure depends on the size of the internal representation for the opaque type:
  - Most fixed-length opaque types have an internal representation that *cannot* fit into an **MI_DATUM** structure. These fixed-length opaque types must be passed *by reference*. The **MI_DATUM** structure contains a pointer to the internal C structure of the opaque type.
  - If your fixed-length opaque type is *always smaller* than the size of the **MI_DATUM** data type, the opaque type can be passed *by value*. The **MI_DATUM** structure contains the actual internal representing of the opaque type.

    For such fixed-length opaque types, you must include the PASSEDBYVALUE modifier in the CREATE OPAQUE TYPE statement when you register the opaque type. The database server stores the value of the PASSEDBYVALUE modifier in the **byvalue** column of the **sysxtdtypes** system catalog table.
- For varying-length opaque types, the **MI_DATUM** structure *always* contains a pointer to an **mi_lvarchar** structure.

    Varying-length opaque types must be passed *by reference*. The actual varying-length data is in the data portion of this **mi_lvarchar** structure.

If the internal representation of a fixed-length opaque type can fit into an **MI_DATUM** structure, the routine manager can pass the internal representation by value. Suppose you have the declaration in Figure 16-5 for a fixed-length opaque type named **two_bytes**.

```
typedef two_bytes_t mi_smallint;
```

Figure 16-5. Internal Representation for the two_bytes Opaque Data Type

The following CREATE OPAQUE TYPE statement specifies that the **two_bytes** fixed-length opaque type can be passed by value:

```
CREATE OPAQUE TYPE two_bytes (INTERNALLENGTH=2,
   ALIGNMENT=2, PASSEDBYVALUE);
```

Figure 16-10 on page 16-15 declares the output support function for the **two_bytes** fixed-length opaque type. The **intrnl_format** parameter in this declaration is

passed *by value*. In contrast, the **circle** fixed-length opaque type (which Figure 16-2 on page 16-3 declares) cannot fit into an **MI_DATUM** structure. Therefore, its output support function must declare its **intrnl_format** parameter as passed by reference, as Figure 16-9 on page 16-14 shows.

When the routine manager receives data from a varying-length opaque type, it passes the data to the C UDR in an **mi_lvarchar** varying-length structure that the UDR allocates. The routine manager also passes a pointer to this **mi_lvarchar** structure as the **MI_DATUM** structure for the UDR argument. Therefore, a C UDR must have its parameter declared as a pointer to an **mi_lvarchar** structure when the parameter accepts data from varying-length opaque types. Figure 16-11 on page 16-15 shows the declaration of the output support function for the **image** varying-length opaque type.

Writing Opaque-Type Support Functions

The database server does not know the internal representation of an opaque type. To handle the internal representation, you write *opaque-type support functions*. These support functions tell the database server how to interact with the opaque type. The following table summarizes the opaque-type support functions.

Category of Support Function	Opaque-Type Support Functions	More Information
Support functions as casts	input, output receive, send import, export importbin, exportbin	"Support Functions as Casts" on page 16-8
Stream support functions	**streamwrite( )**, **streamread( )**	"Stream Support Functions" on page 16-34
Disk-storage support functions	**assign( )**, **destroy( )**	"Disk-Storage Support Functions" on page 16-37
Other support functions	**compare( )**, **deepcopy( )**, **update( )**	The *IBM Informix User-Defined Routines and Data Types Developer's Guide*

Tip: The IBM Informix BladeSmith development tool, which is part of the Informix DataBlade Developers Kit, automatically generates some of the C source code for the support routines of an opaque type. For more information, see the *IBM Informix DataBlade Developers Kit User's Guide*.

The following sections provide information specific to the development of the opaque-type support functions as C UDRs. For a general discussion of opaque-type support functions, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Support Functions as Casts

The internal (binary) representation of an opaque type is a C structure that encapsulates the opaque-type information. The database server does *not* know about the structure of this internal representation. To be able to transfer opaque-type data to various locations, the database server assumes that cast functions exist between the internal representation of the opaque type (which the database server does not know) and some known representation of the opaque-type data.

Many of the opaque-type support functions serve as casts between some known representation of opaque-type data and the internal representation of the opaque type. Each known representation of an opaque type has an associated SQL data type, which you use when you register the support function. Each of these SQL data types has a corresponding DataBlade API data type, which you use when you declare the C function that implements the support function. Table 16-1 shows the opaque-type representations and the corresponding SQL and DataBlade API data types that implement them.

Table 16-1. SQL and DataBlade API Data Types for Opaque-Type Representations

Opaque-Type Representation	SQL Data Type	DataBlade API Data Type	Opaque-Type Support Functions
External (text) representation	LVARCHAR	**mi_lvarchar**	input, output
External binary representation on the client	SENDRECV	**mi_sendrecv**	receive, send
Text load file representation	IMPEXP	**mi_impexp**	import, export
Binary load file representation	IMPEXPBIN	**mi_impexpbin**	importbin, exportbin

When the database server *receives* some known representation of the opaque type, it receives it in one of the SQL data types that Table 16-1 lists. To locate the appropriate opaque-type support function, the database server looks in the **syscasts** system catalog table for a cast function that performs a cast from one of these SQL data types to the opaque type. Table 16-2 shows the opaque-type support functions cast from each of the SQL data types in Table 16-1 to the internal representation of the opaque type.

Table 16-2. Opaque-Type Support Functions That Cast from SQL to Opaque Data Types

Cast		Opaque-Type Support Function
From	To	
LVARCHAR	opaque data type	input
SENDRECV	opaque data type	receive
IMPEXP	opaque data type	import
IMPEXPBIN	opaque data type	importbinary

For example, when the database server receives from a client application an LVARCHAR value for a column of type **circle**, it looks for a cast function that casts this value to the internal representation of the **circle** opaque type. This cast function is the input support function for **circle**, which takes as an argument an **mi_lvarchar** value and returns the **circle_t** structure (which contains the internal representation of **circle**):

```
circle_t *circle_input(external_rep)
   mi_lvarchar *external_rep;
```

The database server then saves the return value of the **circle_input( )** support function in the column whose data type is **circle**. In this way, the database server does *not* need to know about the internal representation of circle. The **circle_input( )** support function handles the details of filling the C structure.

Tip: All of the opaque-type support functions in Table 16-2 must be registered as *implicit* casts in the database. For more information, see "Registering an Opaque Data Type" on page 16-39.

Similarly, when the database server *sends* some known representation of the opaque type, it sends it in one of the SQL data types that Table 16-1 on page 16-9 lists. To locate the appropriate opaque-type support function, the database server looks for a cast function that performs a cast from the opaque type to one of these SQL data types. Table 16-3 shows the opaque-type support functions that cast from the internal representation of the opaque type to each of the SQL data types in Table 16-1.

Table 16-3. Opaque-Type Support Functions That Cast from Opaque to SQL Data Types

| Cast | | Opaque-Type |
From	To	Support Function
opaque data type	LVARCHAR	output
opaque data type	SENDRECV	send
opaque data type	IMPEXP	export
opaque data type	IMPEXPBIN	exportbin

All the opaque-type support functions in Table 16-3 must be registered as *explicit* casts in the database. For more information, see "Registering an Opaque Data Type" on page 16-39.

Important: For the database server to locate one of the opaque-type support functions in Table 16-1, you must register these support functions as cast functions with the CREATE CAST statement. Otherwise, the database server will not find the function to perform the cast when it checks the **syscasts** system catalog table. For more information, see the description of how to create casts for support functions in the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

The DataBlade API data types in Table 16-1 on page 16-9 are *all* implemented as varying-length structures. Therefore, all these data types have the *same* internal format. Any DataBlade API function that is declared to handle the **mi_lvarchar** data type can also handle these other varying-length data types. However, you might need to cast between these types to avoid compilation warnings. If you are using a varying-length data type *other than* **mi_lvarchar**, you can cast between the varying-length type you are using and **mi_lvarchar**.

For example, the **mi_string_to_lvarchar( )** function converts a null-terminated string to an **mi_lvarchar** varying-length data type. You can use casting to have this function convert a null-terminated string to an **mi_impexp** varying-length data type, as follows:

```
mi_impexp *new_impexp;
...
new_impexp = (mi_impexp *)mi_string_to_lvarchar(strng);
```

This casting is *not* strictly required, but many compilers recommend it and it does improve clarity of purpose.

Any size of data can fit into a varying-length structure. When a varying-length data type holds a value for an opaque-type column, this two-kilobyte size restriction for LVARCHAR columns does *not* apply. You can write the appropriate support functions of the opaque data type to handle more than two kilobytes. For more information on how to manage these varying-length structures, see "Varying-Length Data Type Structures" on page 2-13.

Subsequent sections describe each of the opaque-type support functions, grouped by the opaque-type representation that they handle, as the following table shows.

Opaque-Type Support Functions	Description	More Information
input, output	• Convert the opaque-type data between its external and internal representation. • Serve as casts between the LVARCHAR and opaque data types.	"Input and Output Support Functions" on page 16-11
send, receive	• Convert the opaque-type data between its internal representations on the client and server computers. • Serve as casts between the SENDRECV and opaque data types.	"Send and Receive Support Functions" on page 16-17
import, export	• Convert the opaque-type data between its external unload representation and its server internal representation. • Serve as casts between the IMPEXP and opaque data types.	"External Unload Representation" on page 16-22
importbin, exportbin	• Convert the opaque-type data between its internal unload representation and its server internal representation. • Serve as casts between the IMPEXPBIN and opaque data types.	"Internal Unload Representation" on page 16-29

Input and Output Support Functions: To handle opaque-type data in its external text representation, the database server calls the input and output support functions for the opaque type. The external representation is the text version of the opaque-type data. (For more information, see "Determining External Representation" on page 16-2.) The external representation is often what end users enter for the opaque-type value. When a client application sends or receives opaque-type data in its external representation, the database server must find a support function to handle the conversion of this data between its server internal representation (in the database) and the external representation. The input and output support functions are the cast functions for an opaque type between its external (text) representation and its internal (binary) representation (on the server computer). The server internal representation is the C structure that holds the opaque-type data in the database. For more information, see "Determining Internal Representation" on page 16-3.

The database server stores the external representation of an opaque type in an **mi_lvarchar** structure. The **mi_lvarchar** structure is a varying-length structure that encapsulates the external representation of the opaque type. The **mi_lvarchar** structure is *always* passed by reference. Therefore, the input and output support routines cast the data as follows.

Opaque-Type Support Function	Cast From	Cast To
Input	**mi_lvarchar** *	Server internal representation of the opaque data type
Output	Server internal representation of the opaque data type	**mi_lvarchar** *

There is no limitation on the size of an **mi_lvarchar** structure. The DataBlade API can transport **mi_lvarchar** data to and from the database server. However, to conform to the storage limit of a database table (32 kilobytes for a table, two kilobytes for an LVARCHAR column), the input support function might need to handle extra data and the output support function might need to generate the extra data.

The two-kilobyte restriction does not apply to an **mi_lvarchar** structure that holds the external representation of an opaque-type column. If the input and output support functions of the opaque data type can handle more than two kilobytes, the **mi_lvarchar** structure can hold more than two kilobytes. For more information, see "The mi_lvarchar Data Type" on page 2-9.

Global Language Support

For your opaque data type to accept an external representation in non-default locales, you must internationalize the input and output support functions. For more information, see "Internationalization of DataBlade API Modules (GLS)" on page 1-19.

End of Global Language Support

Input Support Function: When an application performs some operation that passes the external representation of an opaque type to the database server (such as INSERT or UPDATE with an opaque-type value as a literal string), the database server calls the input support function. The input support function accepts the external representation of the opaque type, which is encapsulated in an **mi_lvarchar** structure, and returns the appropriate server internal representation for that type, as the following signature shows:

```
srvr_internal_rep input(external_rep)
   mi_lvarchar *external_rep;
```

external_rep   is a pointer to an **mi_lvarchar** structure that holds the external representation of the opaque type.

An **mi_lvarchar** is *always* passed by reference. Therefore, the *external_rep* argument must always be a pointer to the **mi_lvarchar** data type. For information on how to obtain information from this varying-length structure, see "Information About Varying-Length Data" on page 2-24.

input   is the name of the C-language function that implements the input support function for the opaque type. It is recommended that you include the name of the opaque type in its input function.

srvr_internal_rep

is the appropriate format for the server internal representation of the opaque data type. The passing mechanism of this return value depends on the kind of opaque type, as Figures 16-6 through 16-13 show. Most opaque types are passed by reference.

Figure 16-6 declares a sample input support function for a fixed-length opaque type named **circle** (which Figure 16-2 on page 16-3 declares).

```
/* Input support function: circle */
circle_t *circle_input(extrnl_rep)
   mi_lvarchar *extrnl_rep;
```

Figure 16-6. Input Support Function for circle Opaque Type

The **circle_input( )** function is a cast function from the **mi_lvarchar** data type (which contains the external representation for the **circle** opaque type) to the **circle_t** internal representation (on the server computer). The database server executes **circle_input( )** when it needs a cast function to convert from the SQL data type LVARCHAR to the server internal representation of the **circle** opaque type. For more information, see "Support Functions as Casts" on page 16-8.

The **circle_input( )** function returns a pointer to the **circle_t** data type. Because **circle** *cannot* fit into an **MI_DATUM** structure, it must be passed by reference. If your fixed-length opaque type *can* fit into an **MI_DATUM** structure, the input support function can return the internal representation by value. Figure 16-7 declares a sample input function for a fixed-length opaque type named **two_bytes** (which Figure 16-5 on page 16-7 declares).

```
/* Input support function: two_bytes */
two_bytes_t two_bytes_input(extrnl_rep)
   mi_lvarchar *extrnl_rep;
```

Figure 16-7. Input Support Function for two_bytes Opaque Type

The **two_bytes** opaque type must be registered as PASSEDBYVALUE to tell the database server that it can be passed by value.

Figure 16-8 declares a sample input support function for a varying-length opaque type named **image** (which Figure 16-3 on page 16-5 declares).

```
/* Input support function: image */
mi_lvarchar *image_input(extrnl_rep)
   mi_lvarchar *extrnl_rep;
```

Figure 16-8. Input Support Function for image Opaque Type

The **image** opaque type stores its data inside an **mi_lvarchar** structure, which must be passed by reference. The **image_input( )** function is a cast function from the external representation of **image** to the server internal representation of **image**.

The input support function performs the following tasks:
- Accepts as an argument a pointer to the external representation of the opaque type

  The external representation is in the data portion of an **mi_lvarchar** structure, which is passed by reference.
- Allocates enough space to hold the server internal representation of the opaque type

  The input function can use the **mi_alloc( )** DataBlade API function to allocate the space for the internal representation, or the **mi_new_var( )** function if the opaque type is varying length. For more information on memory management, see "Managing User Memory" on page 14-20.
- Parses the input string of the external representation

The input function must obtain the individual members from the input string and store them into the appropriate fields of the server internal representation. The DataBlade API provides functions to convert various DataBlade API data types from their external to internal representations. For example, to convert a date string in an external representation to its internal representation (the **mi_date** value in the **image_t** structure), the **image_input( )** function can call the **mi_string_to_date( )** function. For a list of these DataBlade API functions, see "Conversion of Opaque-Type Data Between Text and Binary Representations" on page 16-16.

- Returns the appropriate server internal representation for the opaque type

  If the opaque data type is passed by reference, the input function returns a pointer to the server internal representation. If the opaque data type is passed by value, the input function returns the actual value of the server internal representation instead of a pointer to this representation. For more information, see "Determining the Passing Mechanism for an Opaque Type" on page 16-7.

Output Support Function: When an application performs some operation that requests the external representation of an opaque type (such as a SELECT operation that requests data in its text representation), the database server calls the output support function. The output support function accepts the appropriate server internal representation of the opaque type and returns the external representation of that type, which is encapsulated in an **mi_lvarchar** structure, as the following signature shows:

```
mi_lvarchar *output(srvr_internal_rep)
```

output   is the name of the C-language function that implements the output support function for the opaque type. It is recommended that you include the name of the opaque type in the name of its output function. For example, if the UDT name is **image**, the name of the output function would be **image_output( )**.

srvr_internal_rep

is the appropriate format for the server internal representation of the opaque data type. The passing mechanism of this argument value depends on the kind of opaque type, as Figures 16-9 through 16-11 show. Most opaque types are passed by reference.

An **mi_lvarchar** value is *always* passed by reference. Therefore, the return value of the output support function must always be a pointer to the **mi_lvarchar** data type. For information on how to obtain information from this varying-length structure, see "Information About Varying-Length Data" on page 2-24.

Figure 16-9 declares a sample output support function for a fixed-length opaque type named **circle** (which Figure 16-2 on page 16-3 declares).

```
/* Output support function: circle */
mi_lvarchar *circle_output(srvr_intrnl_rep)
   circle_t *srvr_intrnl_rep;
```

Figure 16-9. Output Support Function for circle Opaque Type

The **circle_output( )** function is a cast function from the **circle_t** internal representation (on the server computer) to the **mi_lvarchar** data type (which contains the external representation for **circle**). The database server executes **circle_output( )** when it needs a cast function to convert from the server internal

representation of the **circle** opaque type to the SQL data type LVARCHAR. For more information, see "Support Functions as Casts" on page 16-8.

The **circle_output( )** function accepts as an argument a pointer to the **circle_t** data type. Because **circle** *cannot* fit into an **MI_DATUM** structure, it must be passed by reference. If your fixed-length opaque type *can* fit into an **MI_DATUM** structure, the output support function can pass the server internal representation by value. Figure 16-10 declares a sample output function for a fixed-length opaque type named **two_bytes** (which Figure 16-5 on page 16-7 declares).

```
/* Output support function: two_bytes */
mi_lvarchar *two_bytes_output(srvr_intrnl_rep)
   two_bytes_t srvr_intrnl_rep;
```

Figure 16-10. Output Support Function for two_bytes Opaque Type

The **two_bytes** opaque type must be registered as PASSEDBYVALUE to tell the database server that it can be passed by value.

Figure 16-11 declares a sample output support function for a varying-length opaque type named **image** (which Figure 16-3 on page 16-5 declares).

```
/* Output support function: image */
mi_lvarchar *image_output(srvr_intrnl_rep)
   mi_lvarchar *srvr_intrnl_rep;
```

Figure 16-11. Output Support Function for image Opaque Type

The **image** opaque type stores its data inside an **mi_lvarchar** structure, which must be passed by reference. The **image_output( )** function is a cast function from the internal representation of **image** to the external representation of **image**.

The output support function performs the following tasks:

- Accepts as an argument a pointer to the appropriate server internal representation of the opaque type

  If the opaque data type is passed by reference, the output support function accepts a pointer to the server internal representation. If the opaque data type is passed by value, the output function returns the actual value of the internal representation instead of a pointer to this representation. For more information, see "Determining the Passing Mechanism for an Opaque Type" on page 16-7.

- Allocates enough space to hold the external representation of the opaque type

  The output function can use the **mi_alloc( )** DataBlade API function to allocate the space for the character string. For more information on memory management, see "Managing User Memory" on page 14-20.

- Creates the output string of the external representation from the individual members of the server internal representation

  The DataBlade API provides functions to convert various DataBlade API data types from their internal to external representations. For example, to convert the **mi_date** value in the **image_t** structure to its appropriate external representation, the **image_output( )** function can call the **mi_date_to_string( )** function. For a list of these DataBlade API functions, see "Conversion of Opaque-Type Data Between Text and Binary Representations" on page 16-16.

- Copies the external representation into an **mi_lvarchar** structure

You must use the **mi_new_var( )** function to create a new **mi_lvarchar** structure. You can use **mi_set_vardata( )** to copy data into the **mi_lvarchar** structure or **mi_set_varptr( )** to store a pointer to storage allocated by **mi_alloc( )**.

- Returns a pointer to the external representation for the opaque type

  This character string must reside in the data portion of an **mi_lvarchar** structure. Therefore, the output support function returns a pointer to this **mi_lvarchar** structure.

Conversion of Opaque-Type Data Between Text and Binary Representations: The input and output support functions can call the following DataBlade API functions to convert the atomic C data types within the server internal representation of the opaque data type between their external (text) and internal (binary) representations.

	DataBlade API Function	
Type of Data	**In Input Support Function**	**In Output Support Function**
Date and Date/time data		
DATE data	**mi_string_to_date( )**	**mi_date_to_string( )**
DATETIME data	**mi_string_to_datetime( )**	**mi_datetime_to_string( )**
INTERVAL data	**mi_string_to_interval( )**	**mi_interval_to_string( )**
Integer data		
SMALLINT data (two-byte integers)	**rstoi( )**, **atoi( )**	
INTEGER data (four-byte integers)	**rstol( )**, **atol( )**	
INT8 data (eight-byte integers)	**ifx_int8cvasc( )**	**ifx_int8toasc( )**
Fixed-point and Floating-point data		
DECIMAL data (fixed-point and floating-point)	**mi_string_to_decimal( )**	**mi_decimal_to_string( )**
MONEY data	**mi_string_to_money( )**	**mi_money_to_string( )**
SMALLFLOAT data	**atof( )**	
FLOAT data	**rstod( )**	
Other data		
Character data, Varying-length data	**mi_string_to_lvarchar( )**	**mi_lvarchar_to_string( )**
LO handle (smart large objects)	**mi_lo_from_string( )**	**mi_lo_to_string( )**

┌──────────────── **Global Language Support** ────────────────┐

Most DataBlade API functions that convert between text and binary representations recognize the end-user formats for data in a locale-specific format. For more information about how to internationalize a C UDR, see "Internationalization of DataBlade API Modules (GLS)" on page 1-19.

└──────────── **End of Global Language Support** ────────────┘

Send and Receive Support Functions: To handle opaque-type data in its external binary representation, the database server calls the send and receive support functions for the opaque type. When a client application sends or receives opaque-type data in its internal representation, the database server must find a support function to handle the possibility that the client computer uses a different byte ordering than the server computer. The send and receive support functions are the cast functions for an opaque type between its internal representation on a client computer and its internal representation on the server computer.

The database server stores the client internal representation of an opaque type in an **mi_sendrecv** structure. The **mi_sendrecv** structure is a varying-length structure that encapsulates the client internal representation. Its ability to store varying-length data enables it to handle any possible changes in the size of the opaque-type data when it is converted between these two internal representations. For example, the client and server computers might have different packing rules for structures. Because the **mi_sendrecv** data type is a varying-length structure (like **mi_lvarchar**), it is *always* passed by reference. Therefore, the send and receive support routines cast the data as follows.

Opaque-Type Support Function	Cast From	Cast To
Send	Server internal representation of the opaque data type	**mi_sendrecv** *
Receive	**mi_sendrecv** *	Server internal representation of the opaque data type

The database server receives a description of the client computer when the client application establishes a connection. The DataBlade API provides several functions that access this information for use in send and receive functions. For more information, see "Conversion of Opaque-Type Data with Computer-Specific Data Types" on page 16-21.

Receive Support Function: When an application executes a query, such as INSERT or UPDATE, and specifies binary transfer of data, the database server calls the receive support function. The way to specify binary transfer (for fetch or send) depends on the client API:

- ODBC uses an **SQLBindCol( )** call.
- The DataBlade API **mi_exec_prepared_statement( )** call takes a PARAMS_ARE_BINARY flag.
- Informix ESQL/C uses the host-variable data type to specify if the transfer is binary or text.

The receive support function accepts the client internal representation of the opaque type, which is encapsulated in an **mi_sendrecv** structure, and returns the appropriate server internal representation of that type, as the following signature shows:

```
srvr_internal_rep receive(client_internal_rep)
   mi_sendrecv *client_internal_rep;
```

client_internal_rep

is a pointer to an **mi_sendrecv** structure that holds the client internal representation of the opaque type.

An **mi_sendrecv** is *always* passed by reference. Therefore, the *client_internal_rep* argument must always be a pointer to the

mi_sendrecv data type. For more information, see "Information About Varying-Length Data" on page 2-24.

receive   is the name of the C-language function that implements the receive support function for the opaque type. It is recommended that you include the name of the opaque type in its receive function.

srvr_internal_rep

is the appropriate format for the server internal representation of the opaque data type. The passing mechanism of this return value depends on the kind of opaque type, as Figures 16-12 through 16-14 show. Most opaque types are passed by reference.

Figure 16-12 declares a sample receive support function for a fixed-length opaque type named **circle** (which Figure 16-2 on page 16-3 declares).

```
/* Receive support function: circle */
circle_t *circle_recv(client_intrnl_rep)
   mi_sendrecv *client_intrnl_rep;
```

Figure 16-12. Receive Support Function for circle Opaque Type

The **circle_recv( )** function is a cast function from the **mi_sendrecv** data type (which contains the client internal representation for the **circle** opaque type) to the **circle_t** internal representation (on the server computer). The database server executes **circle_recv( )** when it needs a cast function to convert from the SQL data type SENDRECV to the server internal representation of the **circle** opaque type. For more information, see "Support Functions as Casts" on page 16-8.

The **circle_recv( )** function returns a pointer to the **circle_t** data type. Because **circle** *cannot* fit into an **MI_DATUM** structure, it must be passed by reference. If your fixed-length opaque type *can* fit into an **MI_DATUM** structure, the receive support function can return the server internal representation by value. Figure 16-13 declares a sample receive function for a fixed-length opaque type named **two_bytes** (which Figure 16-5 on page 16-7 declares).

```
/* Receive support function: two_bytes */
two_bytes_t two_bytes_recv(client_intrnl_rep)
   mi_sendrecv *client_intrnl_rep;
```

Figure 16-13. Receive Support Function for two_bytes Opaque Type

The **two_bytes** opaque type must be registered as PASSEDBYVALUE to tell the database server that it can be passed by value.

Figure 16-14 declares a sample receive support function for a varying-length opaque type named **image** (which Figure 16-3 on page 16-5 declares).

```
/* Receive support function: image */
mi_lvarchar *image_recv(client_intrnl_rep)
   mi_sendrecv *client_intrnl_rep;
```

Figure 16-14. Receive Support Function for image Opaque Type

The **image** opaque type stores its data inside an **mi_lvarchar** structure, which must be passed by reference. The **image_recv( )** function is a cast function from the

mi_sendrecv data type (which contains the *client* internal representation of **image**) to the **mi_lvarchar** data type (which contains the *server* internal representation of **image**).

The receive support function performs the following tasks:

- Accepts as an argument a pointer to the client internal representation of the opaque type

  The client internal representation is in the data portion of an **mi_sendrecv** structure, which is passed by reference.

- Allocates enough space to hold the server internal representation of the opaque type

  The receive function can use the **mi_alloc( )** DataBlade API function to allocate the space for the internal representation, or the **mi_new_var( )** function if the opaque type is varying length. For more information on memory management, see "Managing User Memory" on page 14-20.

- Creates the server internal representation from the individual members of the client internal representation

  The DataBlade API provides functions to convert simple C data types from their client to server binary representations. For example, to convert the double-precision values in the **circle_t** structure to their binary representation on the server computer, the **circle_recv( )** function can call the **mi_get_double_precision( )** function. For a list of these DataBlade API functions, see "Conversion of Opaque-Type Data with Computer-Specific Data Types" on page 16-21.

- Returns the appropriate server internal representation for the opaque type

  If the opaque data type is passed by reference, the receive function returns a pointer to the server internal representation. If the opaque data type is passed by value, the receive function returns the actual value of the internal representation instead of a pointer to this representation. For more information, see "Determining the Passing Mechanism for an Opaque Type" on page 16-7.

Send Support Function: When an application performs some operation that requests the binary representation of an opaque type (such as a SELECT that requests data in its binary representation), the database server calls the send support function. The send support function takes the appropriate server internal representation of the opaque data type and returns the client internal representation of that type, encapsulated in an **mi_sendrecv** structure, as the following signature shows:

```
mi_sendrecv *send(srvr_internal_rep)
```

send   is the name of the C-language function that implements the send support function for the opaque type. It is recommended that you include the name of the opaque type in its send function.

srvr_internal_rep
                  is the appropriate format for the server internal representation of the opaque data type. The passing mechanism of this argument value depends on the kind of opaque type, as Figures 16-15 through 16-17 show. Most opaque types are passed by reference.

An **mi_sendrecv** is *always* passed by reference. Therefore, the return value of the send support function must always be a pointer to the **mi_sendrecv** data type. For information on how to obtain information from this varying-length structure, see "Information About Varying-Length Data" on page 2-24.

Figure 16-15 declares a sample send support function for a fixed-length opaque type named **circle** (which Figure 16-2 on page 16-3 declares).

```
/* Send support function: circle */
mi_sendrecv *circle_send(srvr_intrnl_rep)
    circle_t *srvr_intrnl_rep;
```

Figure 16-15. Send Support Function for circle Opaque Type

The **circle_send( )** function is a cast function from the **circle_t** internal representation (on the server computer) to the **mi_sendrecv** data type (which contains the client internal representation for **circle**). The database server executes **circle_send( )** when it needs a cast function to convert from the internal representation of the **circle** opaque type to the SQL data type SENDRECV. For more information, see "Support Functions as Casts" on page 16-8.

The **circle_send( )** function accepts as an argument a pointer to the **circle_t** data type. Because **circle** *cannot* fit into an **MI_DATUM** structure, it must be passed by reference. If your fixed-length opaque type *can* fit into an **MI_DATUM** structure, the send support function can pass the server internal representation by value. Figure 16-16 declares a sample send function for a fixed-length opaque type named **two_bytes** (which Figure 16-5 on page 16-7 declares)

```
/* Send support function: two_bytes */
mi_sendrecv *two_bytes_send(srvr_intrnl_rep)
    two_bytes_t srvr_intrnl_rep;
```

Figure 16-16. Send Support Function for two_bytes Opaque Type

The **two_bytes** opaque type must be registered as PASSEDBYVALUE to tell the database server that it can be passed by value.

Figure 16-17 declares a sample send support function for a varying-length opaque type named **image** (which Figure 16-3 on page 16-5 declares)

```
/* Send support function: image */
mi_sendrecv *image_send(srvr_intrnl_rep)
    mi_lvarchar *srvr_intrnl_rep;
```

Figure 16-17. Send Support Function for image Opaque Type

The **image** opaque type stores its data inside an **mi_lvarchar** structure, which must be passed by reference. The **image_send( )** function is a cast function from the **mi_lvarchar** data type (which contains the *server* internal representation of **image**) to the **mi_sendrecv** data type (which contains the *client* internal representation of **image**).

The send support function performs the following tasks:

• Accepts as an argument a pointer to the appropriate server internal representation of the opaque type

If the opaque data type is passed by reference, the send function accepts a pointer to the server internal representation. If the opaque data type is passed by value, the send function accepts the actual value of the internal representation instead of a pointer to this representation. For more information, see "Determining the Passing Mechanism for an Opaque Type" on page 16-7.

- Allocates enough space to hold the client internal representation

  The send function can use the **mi_alloc( )** DataBlade API function to allocate the space for the internal representation. For more information on memory management, see "Managing User Memory" on page 14-20.
- Creates the client internal representation from the individual members of the server internal representation

  The DataBlade API provides functions to convert simple C data types from server to client binary representations. For example, to convert the double-precision values in the **circle_t** structure to their binary representation on the client computer, the **circle_send( )** function can call the **mi_put_double_precision( )** function. For a list of these DataBlade API functions, see "Conversion of Opaque-Type Data with Computer-Specific Data Types" on page 16-21.
- Copies the client internal representation into an **mi_sendrecv** structure

  You must use the **mi_new_var( )** function to create a new **mi_sendrecv** structure. You can use **mi_set_vardata( )** to copy the data into the **mi_sendrecv** structure or **mi_set_varptr( )** to store the pointer to storage allocated by **mi_alloc( )**.
- Returns a pointer to the client internal representation for the opaque type

  This client internal representation must reside in the data portion of an **mi_sendrecv** structure. Therefore, the send support function returns a pointer to this **mi_sendrecv** structure.

Conversion of Opaque-Type Data with Computer-Specific Data Types: The send and receive support functions can call DataBlade API functions to convert data of the atomic C data types within the internal (binary) representation of an opaque data type. Table 16-4 shows the DataBlade API functions that can convert a difference in alignment or byte order between the client computer and the server computer.

Table 16-4. Type-Transfer Functions of the DataBlade API

	DataBlade API Function	
Type of Data	**In Send Support Function**	**In Receive Support Function**
Byte data	**mi_put_bytes( )**	**mi_get_bytes( )**
Date and Date/time data		
DATE data	**mi_put_date( )**	**mi_get_date( )**
DATETIME data	**mi_put_datetime( )**	**mi_get_datetime( )**
INTERVAL data	**mi_put_interval( )**	**mi_get_interval( )**
Integer data		
SMALLINT data (two-byte integers)	**mi_put_smallint( )**, **mi_fix_smallint( )**	**mi_get_smallint( )**, **mi_fix_smallint( )**
INTEGER data (four-byte integers)	**mi_put_integer( )**, **mi_fix_integer( )**	**mi_get_integer( )**, **mi_fix_integer( )**
INT8 data (eight-byte integers)	**mi_put_int8( )**	**mi_get_int8( )**
Fixed-point and Floating-point data		
DECIMAL data (fixed-point and floating-point)	**mi_put_decimal( )**	**mi_get_decimal( )**
MONEY data	**mi_put_money( )**	**mi_get_money( )**
SMALLFLOAT data	**mi_put_real( )**	**mi_get_real( )**

Table 16-4. Type-Transfer Functions of the DataBlade API (continued)

Type of Data	DataBlade API Function	
	In Send Support Function	In Receive Support Function
FLOAT data	**mi_put_double_precision( )**	**mi_get_double_precision( )**
Other data		
Character data	**mi_put_string( )**	**mi_get_string( )**
LO handle (smart large objects)	**mi_put_lo_handle( )**	**mi_get_lo_handle( )**

Global Language Support

Characters have the same binary representation on all architectures, so they do not need to be converted. However, if the code sets of the server-processing locale (in which the UDR executes) and the client locale differ, the **mi_get_string( )** and **mi_put_string( )** functions automatically perform the appropriate code-set conversion (provided that the two code sets are compatible). For more information about how to internationalize a C UDR, see "Internationalization of DataBlade API Modules (GLS)" on page 1-19.

End of Global Language Support

Bulk-Copy Support Functions: The database server can copy data in and out of a database with a bulk copy operation. In a bulk copy, the database server reads or sends large numbers of column values in a copy file, rather than handling each column value individually. IBM Informix utilities such as DB–Access, the **dbimport** and **dbexport** utilities, and the High Performance Loader (HPL) can perform bulk copies.

The format of the opaque-type data in the copy file is called its *unload representation*. This unload representation might be different from the server internal representation of the opaque-type data (which is stored in the database). You can create the following opaque-type support functions to handle the unload representations of the opaque-type data.

Unload Representation	Description	Opaque-Type Support Functions
External unload representation	The *text* format of the opaque type, as it resides in a copy file	import, export
Internal unload representation	The *binary* format of the opaque type, as it resides in a copy file	importbin, exportbin

External Unload Representation: To handle opaque-type data in its external unload representation, the database server calls the import and export support functions of the opaque type. The external unload representation is the text version of the opaque-type data when it resides in a copy file. Usually, the external unload and external representations of an opaque type are the same. When a bulk-copy utility sends or receives opaque-type data in its external unload representation, the database server must find a support function to handle any conversion between this text in the copy file and the individual field values of the server internal representation. The import and export support functions are the cast functions for an opaque type between its external unload representation (its text format in a copy file) and its server internal (binary) representation.

Important: An opaque data type only requires import and export support functions if its external unload representation is different from its external representation (which the input and output support functions handle). For most opaque data types, the database server can use the input and output support functions for import and export, respectively, to handle bulk copies of the opaque-type columns to and from their text representation.

The database server stores the external unload representation of an opaque type in an **mi_impexp** structure. The **mi_impexp** structure is a varying-length structure that encapsulates the external unload representation. Its ability to store varying-length data enables it to handle any possible changes in the size of the opaque-type data when it is converted between its server internal and its external unload representations. For example, opaque data types that contain smart large objects might have a filename in their external unload representation rather than storing all the smart-large-object data in the copy file.

Because the **mi_impexp** data type is a varying-length structure (like **mi_lvarchar**), it is *always* passed by reference. Therefore, the import and export support routines have the following basic signatures.

Opaque-Type Support Function	Cast From	Cast To
Import	**mi_impexp** *	Server internal representation of the opaque data type
Export	Server internal representation of the opaque data type	**mi_impexp** *

Global Language Support

For your opaque data type to accept an external representation in nondefault locales, you must internationalize the import and export support functions. For more information, see "Internationalization of DataBlade API Modules (GLS)" on page 1-19.

End of Global Language Support

For most opaque types, the import support function can be the same as the input support function because the external representation and the external unload representation are usually the same. For such opaque types, you can handle the import support function in either of the following ways:

- Call the input function inside the import function.

  The import functions for the **circle** opaque type (Figure 16-18 on page 16-24) and the **two_bytes** opaque type (Figure 16-19 on page 16-25) use this method.

- Omit the import function from the definition of the opaque type.

  You must still create the implicit cast from the IMPBIN data type to the opaque data type with the CREATE CAST statement. However, instead of listing an import support function as the cast function, list the input support function. The database server would then automatically call the appropriate input support function to load the opaque type when it is in its external unload representation.

Import Support Function: When a bulk-copy utility performs a load of opaque-type data in its external unload representation, the database server calls the import support function. For example, when DB–Access performs a bulk load of an

opaque-type column with the LOAD statement, the database server calls the import support function for the opaque type.

The import support function takes the external unload representation of the opaque type, which is encapsulated in an **mi_impexp** structure, and returns the appropriate server internal representation of that type, as the following signature shows:

```
srvr_internal_rep import(external_unload_rep)
   mi_impexp *external_unload_rep;
```

external_unload_rep

is a pointer to an **mi_impexp** structure that holds the external unload representation of the opaque type.

An **mi_impexp** is *always* passed by reference. Therefore, the *external_unload_rep* argument must always be a pointer to the **mi_impexp** data type. For information on how to obtain information from this varying-length structure, see "Information About Varying-Length Data" on page 2-24.

import

is the name of the C-language function that implements the import support function for the opaque type. It is recommended that you include the name of the opaque type in its import function.

srvr_internal_rep

is the appropriate format for the server internal representation of the opaque data type. The passing mechanism of this return value depends on the kind of opaque type, as Figures 16-18 through 16-20 show. Most opaque types are passed by reference.

Figure 16-18 declares a sample import support function for a fixed-length opaque type named **circle** (which Figure 16-2 on page 16-3 declares).

```
/* Import support function: circle */
circle_t *circle_imp(extrnl_unload_rep)
   mi_impexp *extrnl_unload_rep;
{
   return (circle_input((mi_lvarchar *)extrnl_unload_rep));
}
```

Figure 16-18. Import Support Function for circle Opaque Type

The **circle_imp( )** function is a cast function from the **mi_impexp** data type (which contains the external unload representation for the **circle** opaque type) to the **circle_t** internal representation (on the server computer). The database server executes **circle_imp( )** when it needs a cast function to convert from the SQL data type IMPEXP to the server internal representation of the **circle** opaque type. For more information, see "Support Functions as Casts" on page 16-8.

The **circle_imp( )** function returns a pointer to the **circle_t** data type. Because **circle** *cannot* fit into an **MI_DATUM** structure, it must be passed by reference. If your fixed-length opaque type *can* fit into an **MI_DATUM** structure, the import support function can return the server internal representation by value. Figure 16-19 declares a sample import function for a fixed-length opaque type named **two_bytes** (which Figure 16-5 on page 16-7 declares).

```
/* Import support function: two_bytes */
two_bytes_t two_bytes_imp(extrnl_unload_rep)
   mi_impexp *extrnl_unload_rep;
{
   return ( two_bytes_input( (mi_lvarchar *)extrnl_unload_rep) );
}
```

Figure 16-19. Import Support Function for two_bytes Opaque Type

The **two_bytes** opaque type must be registered as PASSEDBYVALUE to tell the database server that it can be passed by value.

Figure 16-20 declares a sample import support function for a varying-length opaque type named **image** (which Figure 16-3 on page 16-5 declares).

```
/* Import support function: image */
mi_lvarchar *image_imp(extrnl_unload_rep)
   mi_impexp *extrnl_unload_rep;
```

Figure 16-20. Import Support Function for image Opaque Type

The **image** opaque type stores its data inside an **mi_lvarchar** structure, which must be passed by reference. The **image_imp( )** function is a cast function from the **mi_impexp** data type (which contains the external unload representation of **image**) to the **mi_lvarchar** data type (which contains the server internal representation of **image**).

Typically, only opaque data types that contain smart large objects have import and export functions defined. The external unload representation can include a client filename (which contains the smart-large-object data), a length, and an offset. The import support function can use the **mi_lo_from_file( )** function (with the MI_O_CLIENT_FILE file-mode constant) to:

- Open the specified client file.
- Load the data from the client file into a new smart large object, starting at the specified offset, and ending when the specified length is reached.

Finally, the import function must save the LO handle for the new smart large object in the server internal representation of the opaque type.

Tip: For opaque types with smart large objects, you can choose whether to provide support for an external representation (a client filename, length, and offset) in the input and output support functions or the import and export support functions. When you define the input and output support functions to handle this external representation, applications can use this representation as a literal value for opaque-type data.

For an opaque type that *does* require an import support function, the import function performs the following tasks:

- Accepts as an argument a pointer to the external unload representation of the opaque type

  The external unload representation is in the data portion of an **mi_impbin** structure, which is passed by reference.
- Allocates enough space to hold the server internal representation of the opaque type

The import function can use the **mi_alloc( )** DataBlade API function to allocate the space for the internal representation. For more information on memory management, see "Managing User Memory" on page 14-20.

- Parses the input string of the external unload representation

  Obtain the individual members from the input string and store them into the appropriate fields of the server internal representation. The DataBlade API provides functions to convert various DataBlade API data types from their external to internal representations. For example, to convert a date string in an external unload representation to its internal representation (the **mi_date** value in the **image_t** structure), the **image_imp( )** function can call the **mi_string_to_date( )** function. For a list of these DataBlade API functions, see "Conversion of Opaque-Type Data Between Text and Binary Representations" on page 16-16.

- Returns the appropriate server internal representation for the opaque type

  If the opaque data type is passed by reference, the import function returns a pointer to the server internal representation. If the opaque data type is passed by value, the import function returns the actual value of the internal representation instead of a pointer to this representation. For more information, see "Determining the Passing Mechanism for an Opaque Type" on page 16-7.

Because the **image** opaque type contains a smart large object, it would require an import function. From the external unload representation that is read from the copy file, the import function could obtain the name of the client file that contains the smart-large-object data.

Export Support Function: When a bulk-copy utility performs an unload of opaque-type data to its external unload representation, the database server calls the export support function. For example, when DB–Access performs a bulk unload of an opaque-type column with the UNLOAD statement, the database server calls the export support function for the opaque type.

The export support function takes the appropriate server internal representation of the opaque data type and returns the external unload representation of that type, encapsulated in an **mi_impexp** structure, as the following signature shows:

```
mi_impexp *export(srvr_internal_rep)
```

export   is the name of the C-language function that implements the export support function for the opaque type. It is recommended that you include the name of the opaque type in its export function.

srvr_internal_rep

        is the appropriate format for the server internal representation of the opaque data type. The passing mechanism of this argument value depends on the kind of opaque type, as Figures 16-15 through 16-17 show. Most opaque types are passed by reference.

An **mi_impexp** is *always* passed by reference. Therefore, the return value of the export support function must always be a pointer to the **mi_impexp** data type. For information on how to obtain information from this varying-length structure, see "Information About Varying-Length Data" on page 2-24.

Figure 16-21 declares a sample export support function for a fixed-length opaque type named **circle** (which Figure 16-2 on page 16-3 declares).

```
/* Export support function: circle */
mi_impexp *circle_exp(srvr_intrnl_rep)
   circle_t *srvr_intrnl_rep;
{
   return ((mi_impexp *)circle_output(srvr_intrnl_rep));
}
```

Figure 16-21. Export Support Function for circle Opaque Type

The **circle_exp( )** function is a cast function from the **circle_t** internal representation (on the server computer) to the **mi_impexp** data type (which contains the external unload representation for **circle**). The database server executes **circle_exp( )** when it needs a cast function to convert from the server internal representation of the **circle** opaque type to the SQL data type IMPEXP. For more information, see "Support Functions as Casts" on page 16-8.

The **circle_exp( )** function accepts as an argument a pointer to the **circle_t** data type. Because **circle** *cannot* fit into an **MI_DATUM** structure, it must be passed by reference. If your fixed-length opaque type *can* fit into an **MI_DATUM** structure, the export support function can pass the server internal representation by value. Figure 16-22 declares a sample export function for a fixed-length opaque type named **two_bytes** (which Figure 16-5 on page 16-7 declares).

```
/* Export support function: two_bytes */
mi_impexp *two_bytes_exp(srvr_intrnl_rep)
   two_bytes_t srvr_intrnl_rep;
{
   return ((mi_impexp *)two_bytes_output(srvr_intrnl_rep));
}
```

Figure 16-22. Export Support Function for two_bytes Opaque Type

The **two_bytes** opaque type must be registered as PASSEDBYVALUE to tell the database server that it can be passed by value.

Figure 16-23 declares a sample export support function for a varying-length opaque type named **image** (which Figure 16-3 on page 16-5 declares).

```
/* Export support function: image */
mi_impexp *image_exp(srvr_intrnl_rep)
   mi_lvarchar *srvr_intrnl_rep;
```

Figure 16-23. Export Support Function for image Opaque Type

The **image** opaque type stores its data inside an **mi_lvarchar** structure, which must be passed by reference. The **image_exp( )** function is a cast function from the **mi_lvarchar** data type (which contains the server internal representation of **image**) to the **mi_impexp** data type (which contains the external unload representation of **image**).

In most cases, the export support function can be the same as the output support function, because the external representation and the external unload representation are usually the same. For such opaque types, you can handle the export functions in either of the following ways:

• Call the output function inside the export function.

The export functions for the **circle** opaque type (Figure 16-21 on page 16-27) and the **two_bytes** opaque type (Figure 16-22 on page 16-27) use this method.

- Omit the export function from the definition of the opaque type.

   You must still create the explicit cast from the opaque data type to the IMPBIN data type with the CREATE CAST statement. However, instead of listing an export support function as the cast function, list the output support function. The database server would then automatically call the appropriate output support function to unload the opaque type to its external unload representation.

Typically, only opaque data types that contain smart large objects have import and export functions defined. The external unload representation can include a client filename (which contains the smart-large-object data), a length, and an offset. The export support function can obtain the LO handle of the smart large object from the server internal representation of the opaque type. With this LO handle, export can use the **mi_lo_to_file( )** function (with the MI_O_CLIENT_FILE file-mode constant) to:

- Create the specified file on the client computer.
- Write the smart-large-object data into this file at the specified offset and for the number of bytes that the length specifies.

Finally, the export function can put the client filename, length of data, and starting offset into the external unload representation that is to be written to the copy file.

Tip: For opaque types with smart large objects, you can choose whether to provide support for an external representation (a client filename, length, and offset) in the input and output support functions or the import and export support functions. When you define the input and output support functions to handle this external representation, applications can use this representation as a literal value for opaque-type data.

For an opaque type that *does* require an export support function, the export function performs the following tasks:

- Accepts as an argument a pointer to the appropriate server internal representation of the opaque type

   If the opaque data type is passed by reference, the export function accepts a pointer to the server internal representation. If the opaque data type is passed by value, the export function accepts the actual value of the internal representation instead of a pointer to this representation. For more information, see "Determining the Passing Mechanism for an Opaque Type" on page 16-7.

- Allocates enough space to hold the external unload representation of the opaque type

   The export function can use the **mi_alloc( )** DataBlade API function to allocate the space for the character string. For more information on memory management, see "Managing User Memory" on page 14-20.

- Creates the external unload representation from the individual members of the server internal representation

   The DataBlade API provides functions to convert various DataBlade API data types from their internal to external representations. For example, to convert the **mi_date** value in the **image_t** structure to its appropriate external representation, the **image_exp( )** function can call the **mi_date_to_string( )** function. For a list of these DataBlade API functions, see "Conversion of Opaque-Type Data Between Text and Binary Representations" on page 16-16.

- Copies the external unload representation into an **mi_impexp** structure

  You can use the **mi_new_var( )** function to create a new **mi_impexp** structure and the **mi_get_vardata( )** or **mi_get_vardata_align( )** function to obtain a pointer to the data portion of this structure.
- Returns a pointer to the external unload representation for the opaque type

  This character string must reside in the data portion of an **mi_impexp** structure. Therefore, the export support function returns a pointer to this **mi_impexp** structure.

Because the **image** opaque type contains a smart large object, it would require an export function, which could save in the external unload representation that is written to the copy file the name of the client file that contains the smart-large-object data.

Internal Unload Representation: To handle opaque-type data in its internal unload representation, the database server calls the importbin and exportbin support functions of the opaque type. The internal unload representation is the binary version of the opaque-type data when it resides in a copy file. Usually, the internal unload and server internal representations of an opaque type are the same. When a bulk-copy utility sends or receives opaque-type data in its internal unload representation, the database server must find a support function to handle the possibility that the client computer uses a different byte ordering than the server computer. The importbin and exportbin support functions are the cast functions for an opaque type between its internal (binary) unload representation (its binary format in a copy file) and its server internal (binary) representation.

Important: An opaque data type only requires importbin and exportbin support functions if its internal unload representation is different from its server internal representation (which the send and receive support functions handle). For most opaque data types, the database server can use the send and receive support functions for importbin and exportbin, respectively, to handle bulk copies of the opaque-type columns to and from their binary representation.

The database server stores the internal unload representation of an opaque type in an **mi_impexpbin** structure, which is a varying-length structure. Its ability to store varying-length data enables it to handle any possible changes in the size of the opaque-type data when it is converted between these two internal representations. For example, the client and server computers might have different packing rules for structures.

Because the **mi_impexpbin** data type is a varying-length structure (like **mi_lvarchar**), it is *always* passed by reference. Therefore, the importbin and exportbin support routines have the following basic signatures.

Opaque-Type Support Function	Cast From	Cast To
Importbin	**mi_impexpbin** *	Server internal representation of the opaque data type
Exportbin	Server internal representation of the opaque data type	**mi_impexpbin** *

Importbin Support Function: When a bulk-copy utility performs a load of opaque-type data in its internal unload representation, the database server calls the

importbin support function. The importbin support function takes the internal
unload representation of the opaque type, which is encapsulated in an
mi_impexpbin structure, and returns the appropriate server internal representation
of that type, as the following signature shows:

```
srvr_internal_rep importbin(internal_unload_rep)
   mi_impexpbin *internal_unload_rep;
```

importbin   is the name of the C-language function that implements the
importbin support function for the opaque type. It is recommended
that you include the name of the opaque type in its importbin
function.

internal_unload_rep

is a pointer to an **mi_impexpbin** structure that holds the internal
unload representation of the opaque type.

An **mi_impexpbin** is *always* passed by reference. Therefore, the
internal_unload_rep argument must always be a pointer to the
mi_impexpbin data type. For information on how to obtain
information from this varying-length structure, see "Information
About Varying-Length Data" on page 2-24.

srvr_internal_rep

is the appropriate format for the server internal representation of
the opaque data type. The passing mechanism of this return value
depends on the kind of opaque type, as Figures 16-24 through
16-26 show. Most opaque types are passed by reference.

Figure 16-24 declares a sample importbin support function for a fixed-length
opaque type named **circle** (which Figure 16-2 on page 16-3 declares).

```
/* Importbin support function: circle */
circle_t *circle_impbin(intrnl_unload_rep)
   mi_impexpbin *intrnl_unload_rep;
{
   return ( circle_recv((mi_sendrecv *)intrnl_unload_rep) );
}
```

Figure 16-24. Importbin Support Function for circle Opaque Type

The **circle_impbin( )** function is a cast function from the **mi_impexpbin** data type
(which contains the internal unload representation for the **circle** opaque type) to
the **circle_t** internal representation (on the server computer). The database server
executes **circle_impbin( )** when it needs a cast function to convert from the SQL
data type IMPEXPBIN to the server internal representation of the **circle** opaque
type. For more information, see "Support Functions as Casts" on page 16-8.

The **circle_impbin( )** function returns a pointer to the **circle_t** data type. Because
circle *cannot* fit into an **MI_DATUM** structure, it must be passed by reference. If
your fixed-length opaque type *can* fit into an **MI_DATUM** structure, the importbin
support function can return the server internal representation by value.
Figure 16-25 declares a sample importbin function for a fixed-length opaque type
named **two_bytes** (which Figure 16-5 on page 16-7 declares).

```
/* Importbin support function: two_bytes */
two_bytes_t two_bytes_impbin(intrnl_unload_rep)
   mi_impexpbin *intrnl_unload_rep;
{
   return ( two_bytes_recv( (mi_sendrecv *)intrnl_unload_rep) );
}
```

Figure 16-25. Importbin Support Function for two_bytes Opaque Type

The **two_bytes** opaque type must be registered as PASSEDBYVALUE to tell the database server that it can be passed by value.

Figure 16-26 declares a sample importbin support function for a varying-length opaque type named **image** (which Figure 16-3 on page 16-5 declares).

```
/* Importbin support function: image */
mi_lvarchar *image_impbin(intrnl_unload_rep)
   mi_impexpbin *intrnl_unload_rep;
{
   return ( image_recv( (mi_sendrecv *)intrnl_unload_rep) );
}
```

Figure 16-26. Importbin Support Function for image Opaque Type

The **image** opaque type stores its data inside an **mi_lvarchar** structure, which must be passed by reference. The **image_impbin( )** function is a cast function from the **mi_impexpbin** data type (which contains the internal unload representation of **image**) to the **mi_lvarchar** data type (which contains the server internal representation of **image**).

For most opaque types, the importbin support function can be the same as the receive support function, because the client internal representation and the internal unload representation are the same. For such opaque types, you can handle the importbin function in either of the following ways:

- Call the receive function inside the importbin function

  The importbin functions for the **circle** opaque type (Figure 16-24 on page 16-30), the **two_bytes** opaque type (Figure 16-25 on page 16-31), and the **image** opaque type (Figure 16-26 on page 16-31) use this method.

- Omit the importbin function from the definition of the opaque type

  You must still create the implicit cast from the IMPEXPBIN data type to the opaque data type with the CREATE CAST statement. However, instead of listing an importbin support function as the cast function, list the receive support function. The database server would then automatically call the appropriate receive support function to load the opaque type when it is in its internal unload representation.

For an opaque type that *does* require an importbin support function, the importbin function performs the following tasks:

- Accepts as an argument a pointer to the internal unload representation of the opaque type

  The internal unload representation is in the data portion of an **mi_impexpbin** structure, which is passed by reference.

- Allocates enough space to hold the server internal representation of the opaque type

The importbin function can use the **mi_alloc( )** DataBlade API function to allocate the space for the internal representation. For more information on memory management, see "Managing User Memory" on page 14-20.

- Creates the server internal representation from the individual members of the internal unload representation

  The DataBlade API provides functions to convert simple C data types from their client to server binary representations. For example, to convert the double-precision values in the **circle_t** structure to their binary representation on the server computer, the **circle_impbin( )** function can call the **mi_get_double_precision( )** function. For a list of these DataBlade API functions, see "Conversion of Opaque-Type Data with Computer-Specific Data Types" on page 16-21.

- Returns the appropriate server internal representation for the opaque type

  If the opaque data type is passed by reference, the importbin function returns a pointer to the server internal representation. If the opaque data type is passed by value, the importbin function returns the actual value of the internal representation instead of a pointer to this representation. For more information, see "Determining the Passing Mechanism for an Opaque Type" on page 16-7.

Exportbin Support Function: When a bulk-copy utility performs an unload of opaque-type data to its internal unload representation, the database server calls the exportbin support function. The exportbin support function takes the appropriate server internal representation of the opaque data type and returns the internal unload representation of that type, encapsulated in an **mi_impexpbin** structure, as the following signature shows:

```
mi_impexpbin *exportbin(srvr_internal_rep)
```

exportbin   is the name of the C-language function that implements the exportbin support function for the opaque type. It is recommended that you include the name of the opaque type in its exportbin function.

srvr_internal_rep
      is the appropriate format for the server internal representation of the opaque data type. The passing mechanism of this argument value depends on the kind of opaque type, as Figures 16-27 through 16-29 show. Most opaque types are passed by reference.

An **mi_impexpbin** is *always* passed by reference. Therefore, the return value of the exportbin support function must always be a pointer to the **mi_impexpbin** data type. For information on how to obtain information from this varying-length structure, see "Information About Varying-Length Data" on page 2-24.

Figure 16-27 declares a sample exportbin support function for a fixed-length opaque type named **circle** (which Figure 16-2 on page 16-3 declares).

```
/* Exportbin support function: circle */
mi_impexpbin *circle_expbin(srvr_intrnl_rep)
   circle_t *srvr_intrnl_rep;
{
   return ((mi_impexpbin *)circle_send(srvr_intrnl_rep));
}
```

Figure 16-27. Exportbin Support Function for circle Opaque Type

The **circle_expbin( )** function is a cast function from the **circle_t** internal representation (on the server computer) to the **mi_impexpbin** data type (which contains the internal unload representation for **circle**). The database server executes **circle_expbin( )** when it needs a cast function to convert from the server internal representation of the **circle** opaque type to the SQL data type IMPEXPBIN. For more information, see "Support Functions as Casts" on page 16-8.

The **circle_expbin( )** function accepts as an argument a pointer to the **circle_t** data type. Because **circle** *cannot* fit into an **MI_DATUM** structure, it must be passed by reference. If your fixed-length opaque type *can* fit into an **MI_DATUM** structure, the exportbin support function can pass the server internal representation by value. Figure 16-28 declares a sample exportbin function for a fixed-length opaque type named **two_bytes** (which Figure 16-5 on page 16-7 declares).

```
/* Exportbin support function: two_bytes */
mi_impexpbin *two_bytes_expbin(srvr_intrnl_rep)
   two_bytes_t srvr_intrnl_rep;
{
   return ( (mi_impexpbin *)two_bytes_send( srvr_intrnl_rep) );
}
```

Figure 16-28. Exportbin Support Function for two_bytes Opaque Type

The **two_bytes** opaque type must be registered as PASSEDBYVALUE to tell the database server that it can be passed by value.

Figure 16-29 declares a sample exportbin support function for a varying-length opaque type named **image** (which Figure 16-3 on page 16-5 declares).

```
/* Exportbin support function: image */
mi_impexpbin *image_expbin(srvr_intrnl_rep)
   mi_lvarchar *srvr_intrnl_rep;
{
   return ((mi_impexpbin *)image_send(srvr_intrnl_rep));
}
```

Figure 16-29. Exportbin Support Function for image Opaque Type

The **image** opaque type stores its data inside an **mi_lvarchar** structure, which must be passed by reference. The **image_expbin( )** function is a cast function from the **mi_lvarchar** data type (which contains the server internal representation of **image**) to the **mi_impexpbin** data type (which contains the internal unload representation of **image**).

For most opaque types, the exportbin function can be the same as the send support function, because the client internal representation and the internal unload representation are the same. For such opaque types, you can handle the exportbin support function in either of the following ways:

• Call the send function inside the exportbin function.

  The **circle** opaque type (Figure 16-27 on page 16-32), the **two_bytes** opaque type (Figure 16-28 on page 16-33), and the **image** opaque type (Figure 16-29 on page 16-33) use this method.

• Omit the exportbin function from the definition of the opaque type.

  You must still create the explicit cast from the opaque data type to the IMPEXPBIN data type with the CREATE CAST statement. However, instead of listing an exportbin support function as the cast function, list the send support

function. The database server would then automatically call the appropriate send support function to unload the opaque type to its internal unload representation.

For an opaque type that *does* require an exportbin support function, the exportbin function performs the following tasks:

- Accepts as an argument a pointer to the appropriate server internal representation of the opaque type

   If the opaque data type is passed by reference, the exportbin function accepts a pointer to the server internal representation. If the opaque data type is passed by value, the exportbin function returns the actual value of the internal representation instead of a pointer to this representation. For more information, see "Determining the Passing Mechanism for an Opaque Type" on page 16-7.

- Allocates enough space to hold the internal unload representation of the opaque type

   The exportbin function can use the **mi_alloc( )** DataBlade API function to allocate the space for the internal representation. For more information on memory management, see "Managing User Memory" on page 14-20.

- Creates the internal unload representation from the individual members of the server internal representation

   The DataBlade API provides functions to convert simple C data types from server to client binary representations. For example, to convert the double-precision values in the **circle_t** structure to their binary representation on the client computer, the **circle_expbin( )** function can call the **mi_put_double_precision( )** function. For a list of these DataBlade API functions, see "Conversion of Opaque-Type Data with Computer-Specific Data Types" on page 16-21.

- Copies the internal unload representation into an **mi_impexpbin** structure

   You can use the **mi_new_var( )** function to create a new **mi_impexpbin** structure and the **mi_get_vardata( )** or **mi_get_vardata_align( )** function to obtain a pointer to the data portion of this structure.

- Returns a pointer to the internal unload representation for the opaque type

   This internal unload representation must reside in the data portion of an **mi_impexpbin** structure. Therefore, the exportbin support function returns a pointer to this **mi_impexpbin** structure.

Stream Support Functions

The following support functions convert a UDT to or from a stream representation while reading the UDT from a stream or writing the UDT to a stream.

Support Function	Purpose
streamwrite( )	Conversion of opaque-type data from its binary representation to its stream representation
streamread( )	Conversion of opaque-type data from its stream representation to its binary representation

The stream representation is self-contained and includes enough information to enable the **streamread( )** function to re-create the UDT instance.

Important: If the UDT includes out-of-row data, the stream representation should normally include that data.

Enterprise Replication invokes the **streamwrite( )** and **streamread( )** support functions when replicating UDT columns. The streams it passes to these functions are Enterprise Replication streams, a write-only stream for **streamwrite()** and a read-only stream for **streamread()**. You cannot open or close an Enterprise Replication stream or use the **mi_stream_setpos( )** or **mi_stream_seek( )** function on it. For more information about Enterprise Replication, see the *IBM Informix Dynamic Server Enterprise Replication Guide*.

Important: If a column that includes out-of-row data is to be replicated, avoid placing a NOT NULL constraint on the column. Enterprise Replication collects out-of-row data for transmission after the user transaction has committed. Due to activity on the replicated row, the data might not exist at the time Enterprise Replication collects it for replication. In such cases, Enterprise Replication normally applies a NULL on the target system.

The streamwrite( ) Support Function: On a destination database server, the **streamwrite( )** support function converts opaque-type data from its binary representation to its stream representation. The **streamwrite( )** support function accepts a stream descriptor and the address of opaque-type data to write, as the following signature shows:

```
mi_integer streamwrite(strm_desc, binary_rep)
   MI_STREAM *strm_desc;
   my_opq_type *binary_rep;
```

strm_desc   is a pointer to a stream descriptor for an open stream. For more information, see "Access to a Stream (Server)" on page 13-42.

binary_rep   is a pointer to the binary representation of the opaque-type data, which is written to the stream.

The binary representation is the appropriate format for the opaque-type data. The passing mechanism for this data depends on the kind of opaque type, as Figures 16-6 through 16-8 show. Most opaque-type values are passed by reference to the **streamwrite( )** function as single pointers.

The **streamwrite( )** function returns the number of bytes written to the stream or MI_ERROR. This function can also return the errors that **mi_stream_write( )** returns. To convert the individual fields of the opaque-type internal representation to their stream representation, **streamwrite( )** can call the stream-write functions of the DataBlade API (see Table 16-5 on page 16-36).

A sample SQL declaration for the **streamwrite()** function follows:

```
CREATE FUNCTION streamwrite(STREAM, MyUdt)
RETURNS INTEGER
EXTERNAL NAME '/usr/local/udrs/stream/myudt.so(MyUdtStreamWrite)'
LANGUAGE C;
```

Tip: Unlike most opaque-type support functions, the **streamwrite( )** function for an opaque type must have the explicit name "**streamwrite**" when you register it with the CREATE FUNCTION statement. It is recommended that you include the name of the opaque type in the C-language version of its **streamwrite( )** function.

The streamread( ) Support Function: On a target database server, the **streamread( )** support function converts opaque-type data from its stream representation to its binary representation, which is stored in the target database.

The **streamread( )** support function accepts a stream descriptor and the address of a buffer into which to read the opaque-type data, as the following signature shows:

```
mi_integer streamread(strm_desc, binary_rep)
   MI_STREAM  *strm_desc;
   my_opq_type **binary_rep;
```

strm_desc   is a pointer to a stream descriptor for an open stream. For more information, see "Access to a Stream (Server)" on page 13-42.

binary_rep is a pointer to the buffer into which the function is to copy the binary representation of the opaque-type data.

The stream buffer is declared with the appropriate format for the binary representation of the opaque-type data. The passing mechanism for this buffer depends on the kind of opaque type, as Figures 16-6 through 16-8 show. Most buffers for opaque-type data are passed by reference to **streamread( )** as double pointers.

The **streamread( )** function returns the number of bytes read from the stream or MI_ERROR. This function can also return the errors that **mi_stream_read( )** returns. To convert the individual fields of the opaque-type internal representation to their binary representation, **streamwrite( )** can call the stream-read functions of the DataBlade API (see Table 16-5).

A sample SQL declaration for the **streamread()** function follows:

```
CREATE FUNCTION streamread(STREAM, OUT MyUdt)
RETURNS INTEGER
EXTERNAL NAME '/usr/local/udrs/stream/myudt.so(MyUdtStreamRead)'
LANGUAGE C;
```

Tip: Unlike most opaque-type support functions, the **streamread( )** function for an opaque type must have the explicit name "**streamread**" when you register it with the CREATE FUNCTION statement. It is recommended that you include the name of the opaque type in the C-language version of its **streamread( )** function.

Converting Opaque-Type Data Between Stream and Binary Representations:
The DataBlade API provides several functions to convert built-in data types between binary and stream representations. The **streamwrite( )** and **streamread( )** support functions can use these DataBlade API functions to convert a UDT between its binary representation and its stream representation.

Important: Writing a collection or row to a stream opened for Enterprise Replication is not supported. Likewise, reading a collection or row from a stream opened for Enterprise Replication is not supported.

Table 16-5 shows the DataBlade API stream-conversion functions.

Table 16-5. Stream-Conversion Functions of the DataBlade API

	DataBlade API Function	
Type of Data	The streamwrite( ) Support Function	The streamread( ) Support Function
Byte data	mi_stream_write( )	mi_stream_read( )
Date and Date/time data		
DATE data	mi_streamwrite_date( )	mi_streamread_date( )
DATETIME data	mi_streamwrite_datetime( )	mi_streamread_datetime( )

Type of Data	DataBlade API Function	
	The streamwrite( ) Support Function	The streamread( ) Support Function
INTERVAL data	mi_streamwrite_interval( )	mi_streamread_interval( )
Integer data		
SMALLINT data (two-byte integers)	mi_streamwrite_smallint( )	mi_streamread_smallint( )
INTEGER data (four-byte integers)	mi_streamwrite_integer( )	mi_streamread_integer( )
INT8 data (eight-byte integers)	mi_streamwrite_int8( )	mi_streamread_int8( )
Fixed-point and Floating-point data		
DECIMAL data (fixed- and floating-point)	mi_streamwrite_decimal( )	mi_streamread_decimal( )
MONEY data	mi_streamwrite_money( )	mi_streamread_money( )
SMALLFLOAT data	mi_streamwrite_real( )	mi_streamread_real( )
FLOAT data	mi_streamwrite_double( )	mi_streamread_double( )
Other data		
Character data	mi_streamwrite_string( )	mi_streamread_string( )
Smart large objects	mi_streamwrite_lo( )	mi_streamread_lo( ) mi_streamread_lo_by_lofd( )
Boolean data	mi_streamwrite_boolean( )	mi_streamread_boolean( )
Collection structures	mi_streamwrite_collection( )	mi_streamread_collection( )
Row structures	mi_streamwrite_row( )	mi_streamread_row( )
Varying-length structures	mi_streamwrite_lvarchar( )	mi_streamread_lvarchar( )

The DataBlade API function converts the corresponding data type to a machine-independent stream representation.

Important: The **mistrmutil.h** header file declares the stream-conversion functions of the DataBlade API; however, the **mi.h** header file does not include **mistrmutil.h**. You must explicitly include **mistrmutil.h** in files that use these stream-conversion functions.

Disk-Storage Support Functions

To provide the ability to perform special processing on the internal representation of an opaque type that it is stored on disk, you can define the following disk-storage support functions for an opaque type.

Support Function	Purpose
assign( )	Special processing required just before a row that contains the opaque-type column is *inserted* into the table (written to disk)

destroy( )	Special processing required just before a row that contains the opaque-type column is *deleted* from a table (removed from disk)

The disk internal representation is the contents of the C structure that is actually written to disk for the opaque-type column. The **assign( )** and **destroy( )** support functions are useful for opaque types that contain smart large objects. For such data types, **assign( )** and **destroy( )** can provide management of the associated smart large object as well as any necessary modification of the internal representation.

Important: An opaque data type requires **assign( )** and **destroy( )** support functions only if its disk internal representation is different from its server internal representation. For most opaque types, these two representations are the same.

The assign( ) Support Function: The database server calls the **assign( )** support function for an opaque type when a value is ready to be inserted into an opaque-type column (INSERT, UPDATE, or LOAD). The **assign( )** support function accepts the server internal representation of the opaque type and returns the appropriate disk internal representation for that type, as the following signature shows:

```
disk_internal_rep assign(internal_rep);
```

disk_internal_rep

> is the appropriate format for the disk internal representation of the opaque data type. The passing mechanism of this return value depends on the kind of opaque type. For more information, see "Determining the Passing Mechanism for an Opaque Type" on page 16-7. The disk internal representation is the internal format as modified by the **assign( )** support function. This format is what the database server writes to the database table.

internal_rep is the appropriate format for the server internal representation of the opaque data type. The passing mechanism of this return value depends on the kind of opaque type. For more information, see "Determining the Passing Mechanism for an Opaque Type" on page 16-7. The server internal representation is the representation that the input support function returns.

> **Tip:** Unlike most opaque-type support functions, the **assign( )** function for an opaque type must have the explicit name "**assign**" when you register it with the CREATE FUNCTION statement. No implicit casting occurs when the database server resolves this function. However, it is recommended that you include the name of the opaque type in the C-language version of its **assign( )** function.

The destroy( ) Support Function: The database server calls the **destroy( )** support function for an opaque type when a value is ready to be deleted from an opaque-type column (DELETE or DROP TABLE). The **destroy( )** support function accepts the disk internal representation of the opaque data type and does not return a value, as the following signature shows:

```
void destroy(disk_internal_rep);
```

disk_internal_rep

> is the appropriate format for the disk internal representation of the

opaque data type. The passing mechanism of this return value depends on the kind of opaque type. For more information, see "Determining the Passing Mechanism for an Opaque Type" on page 16-7.

> **Tip:** Unlike most opaque-type support functions, the **destroy( )** function for an opaque type must have the explicit name "**destroy**" when you register it with the CREATE FUNCTION statement. No implicit casting occurs when the database server resolves this function. However, it is recommended that you include the name of the opaque type in the C-language version of its **destroy( )** function.

Handling Locale-Specific Opaque-Type Data (GLS)

To internationalize your opaque type, you must ensure that the following support functions handle data in a nondefault locale:

- The input and output support functions provide the ability to transfer the external representation of the opaque type.
- The send and receive support functions provide the ability to transfer the binary representation of the opaque type.

For a description of the internationalization support that the DataBlade API provides, see "Internationalization of DataBlade API Modules (GLS)" on page 1-19. For general information on internationalized support that the opaque-type can provide, see the chapter on support functions in the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Registering an Opaque Data Type

This section explains how to register an opaque data type.

To register an opaque type in a database:

1. Register the opaque type as an extended data type.
2. Register the opaque-type support functions.
3. Register the opaque-type casts.

For more information on how to register an opaque type and grant the associated privileges, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Registering an Opaque Type in a Database

Use the CREATE OPAQUE TYPE statement to register an opaque data type in a database. For more information, see "Determining Internal Representation" on page 16-3. You can assign privileges to the opaque type with the GRANT USAGE ON TYPE statement. Type privileges for user-defined types (including opaque types) are stored in the **sysxtdtypeauth** system catalog table. By default, Usage privilege is granted to the person who registered the user-defined type. For more information on the syntax of the GRANT statement, see the *IBM Informix Guide to SQL: Syntax*.

Registering Opaque-Type Support Functions

To have the database server able to locate the opaque-type support functions, you must register them with the following actions:

- Use the CREATE FUNCTION statement to register the opaque-type support functions as C UDRs.

For more information, see "Registering a C UDR" on page 12-14.
- Use the GRANT EXECUTE ON statement to grant the Execute privilege to the opaque-type support functions.

  For more information, see "Privileges for the UDR" on page 12-18.
- Use the GRANT USAGE ON LANGUAGE statement to ensure that users have the Usage privilege in the C language for UDRs.

  For more information, see "The UDR Language" on page 12-16.

Registering the Opaque-Type Casts

Use the CREATE CAST statement to register the input, output, receive, send, import, export, importbin, and exportbin support functions as cast functions in the **syscasts** system catalog table. The input, receive, import, and importbin support functions must be registered as implicit casts. The output, send, export, and exportbin support functions must be registered as explicit casts. For more information, see "Support Functions as Casts" on page 16-8.

Providing Statistics Data for a Column

The database server can provide statistics data for the columns of a table. This statistics data describes the distribution of the values within a column. The query optimizer uses this statistics data to determine the best path for an SQL statement. With this information, the optimizer can estimate the effect of a WHERE clause by examining, for each column included in the WHERE clause, the proportionate occurrence of data values contained in the column. (For more information about statistics data and the optimizer, see your *IBM Informix Performance Guide*.)

The database server provides the following support for column statistics data:
- The UPDATE STATISTICS statement collects statistics data for the columns of a table.
- The **dbschema -hd** command displays statistics data for columns in a table.

However, the database server can only provide this support for columns with built-in data types. For the database server to support statistics data for a column with a user-defined data type, you must write special UDRs that collect and print the statistics data.

DBDK

BladeSmith can automatically generate user-defined statistics for an opaque data type in a **statistics.c** file. This file contains the following user-defined functions.

Statistics Function	Purpose
*Opaque*StatCollect( )	The **statcollect( )** function for the *Opaque* data type
*Opaque*StatPrint( )	The **statprint( )** function for the *Opaque* data type

These functions are *not* complete. You must add code to handle the statistics data to these functions for them to compile and execute.

End of DBDK

Collecting Statistics Data

The UPDATE STATISTICS statement collects statistics about the tables in your database. It automatically collects statistics for all columns with built-in data types

(except TEXT and BYTE). However, it cannot automatically collect statistics for columns with user-defined data types because it does not know the structure of these data types.

For UPDATE STATISTICS to collect statistics for a column with a user-defined data type, you must write a user-defined function named **statcollect( )** that collects statistics data for your user-defined data type. The UPDATE STATISTICS statement takes the following steps for columns of user-defined data types:

- Calls the **statcollect( )** function that handles the user-defined data type

  This **statcollect( )** function gathers the statistics data for the column and stores it as the **stat** opaque data type.
- Stores this **stat** data type in the **sysdistrib** system catalog table, where the statistics data can be accessed by the query optimizer

  UPDATE STATISTICS stores the following information in the row of the **sysdistrib** table that corresponds to the user-defined-type column:
  - In the **encdat** column of the **sysdistrib** row: the **stat** data type that **statcollect( )** returns
  - In the **type** column of the **sysdistrib** row: an 'S' to indicate that the **encdat** column contains user-defined statistics

To have the UPDATE STATISTICS statement collect statistics for your user-defined data type, you must:

- Design the statistics information that is appropriate for your user-defined data type.
- Define a C statistics-collection function to implement the statistics collection.
- Collect the statistics for the column within this statistics-collection function.
- Register this C function as a **statcollect( )** user-defined function.

If a **statcollect( )** function does not exist for your user-defined data type, UPDATE STATISTICS does *not* collect statistics data for any column of that type.

Designing the User-Defined Statistics

Before you begin to code a **statcollect( )** function for a particular user-defined data type, you need to decide what it means to collect statistics on this data type. For example, consider the following issues:

- Do the values of the user-defined type have some ordering?

  To be able to group the values into bins of related values, the data must have some kind of implied sequence. A common use of statistics information is within a selectivity function for a query filter such as "less than" or "greater than". If the values of the user-defined data type do not have ordering, they would not logically be used in such filters. For more info, see "Query Selectivity" on page 15-54.
- How does the distribution handle SQL NULL values?

  For example, the distribution can ignore NULL values or it could aggregate them. However, the handling of the NULL values should make sense to the user-defined data type.

Defining the Statistics-Collection Function

When you declare your statistics-collection function, it must have the following C signature:

```
mi_statret *statcollect(udt_arg, num_rows, resolution,
    fparam_ptr)
  udt_type *udt_arg;
  mi_double_precision *num_rows;
  mi_double_precision *resolution;
  MI_FPARAM *fparam_ptr;
```

udt_arg   is a pointer to the internal structure of the user-defined data type. The database server uses this argument to resolve the function and to pass in column values.

num_rows   is a pointer to a floating-point value that indicates the number of rows that the database server must scan to gather the statistics.

resolution   is a pointer to a floating-point value that is the resolution specified by the UPDATE STATISTICS statement. The resolution value specifies the bucket size for the distribution. However, you might choose to ignore this parameter if it does not make sense for your user-defined data type.

fparam_ptr   is a pointer to the **MI_FPARAM** structure that holds the iterator-status constant for each iteration of the **statcollect( )** function.

> **Tip:** The statistics-collection function can have any name. It does not have to be named **statcollect( )**. It is recommended that you include the name of your user-defined data type in the name of the statistics-collection function to help distinguish the function from the statistics-collection functions of other user-defined data type.

Figure 16-30 shows a C declaration of the statistics-collection function for the **longlong** opaque type.

```
mi_statret *statcollect_ll(ll_arg, num_rows, resolution, fparam_ptr)
  longlong_t *ll_arg;
  mi_double_precision *num_rows;
  mi_double_precision *resolution;
  MI_FPARAM *fparam_ptr;
```

Figure 16-30. Sample Declaration of a Statistics-Collection Function

─── **DBDK** ───

BladeSmith automatically generates an *Opaque***StatCollect( )** function (in which *Opaque* is the name of your opaque data type) with the following declaration:

```
mi_lvarchar *OpaqueStatCollect(Gen_pColValue,
    Gen_Resolution, Gen_RowCount, Gen_fparam)
  Opaque *Gen_pColValue;
  mi_double_precision *Gen_Resolution;
  mi_double_precision *Gen_RowCount;
  MI_FPARAM *Gen_fparam;
```

If this declaration is not appropriate for your opaque type, you must customize the *Opaque***StatCollect( )** function.

─── **End of DBDK** ───

Collecting the Statistics

The **statcollect( )** user-defined function is an iterator function; that is, the database server calls **statcollect( )** for *each* of the rows on whose column of a user-defined data type UPDATE STATISTICS is collecting statistics. As with other iterator functions, the database server uses an iterator-status constant to indicate when the statistics-collection function is called.

Important: The database server passes the value of the iterator-status constant within the **MI_FPARAM** structure. Therefore, your statistics-collection function must declare an **MI_FPARAM** structure as its last parameter. Otherwise, it cannot access the value of the iterator-status constant with the **mi_fp_request( )** function.

The following table summarizes the values of the iterator-status constant for the **statcollect( )** function.

When Is the statcollect( ) Function Called?	What Does statcollect( ) Need To Do?	Iterator-Status Constant in MI_FPARAM
The *first* time that **statcollect( )** is called	Perform any initialization operations, such as allocating memory for a statistics-collection structure and initializing values First argument (*udt_arg*) is a NULL value.	SET_INIT
Once for each row for which statistics are being collected	Return one item of the active set Read the column value from the first argument (*udt_arg*) and place it in your statistics-collection structure.	SET_RETONE
After all rows have been processed	Release iteration resources Put the statistics in the **stat** data type and perform any memory deallocation	SET_END

To obtain the iterator-status constant in each iteration, your **statcollect( )** function can use a **switch** statement on the return value of the **mi_fp_request( )** function, as follows:

```
switch ( mi_fp_request(fparam_ptr) )
{
   case SET_INIT:
      ...
   case SET_RETONE:
      ...
   case SET_END:
      ...
}
```

If **statcollect( )** raises an error, UPDATE STATISTICS terminates the statistics collection for that column.

The following sections summarize the steps that **statcollect( )** must take for each of these iterator-status constants. For general information about iterator-status constants, see Table 15-1 on page 15-3.

SET_INIT in statcollect( ): When the iterator-status constant is SET_INIT, the database server has invoked the initial call to **statcollect( )**. Usually, in this initial

call, your **statcollect( )** function allocates and initializes an internal C structure, called a *statistics-collection structure*. The statistics-collection structure is a holding area for the statistics data that **statcollect( )** gathers on a row-by-row basis.

DBDK

BladeSmith generates the *Opaque***StatCollect( )** function (in which *Opaque* is the name of your opaque data type), which allocates a statistics-collection structure *Opaque*_**stat_t** (declared in a file with the **.h** extension). This structure contains the following information.

Element of Statistics-Collection Structure	Description	Data Type
count	Current number of rows	**mi_integer**
max	Maximum value	**mi_integer**
min	Minimum value	**mi_integer**
distribution[]	An array to hold the "in-progress" statistics data	An array of **mi_integer** values whose size is the number of elements that can fit into the text distribution area (usually 256 bytes)

BladeSmith generates statistics code under the assumption that the minimum, maximum, and distribution of values are appropriate for your opaque data type. The SET_INIT case in the *Opaque***StatCollect( )** function calls the *Opaque*_**SetMaxValue( )** and *Opaque*_**SetMaxValue( )** functions (which you must implement) to initialize maximum and minimum values, respectively. It initializes the current row count and the elements of the **distribution** array to zero (0).

If this statistics data is not appropriate for your opaque type, take the following actions:

- Define your own statistics-collection structure to hold statistics data.
- Allocate and initialize this statistics-collection structure within the SET_INIT case of your **statcollect( )** function.

End of DBDK

Your **statcollect( )** function can use the **MI_FPARAM** structure to store this statistics-collection structure (and any other state information) between iterations of **statcollect( )**. Allocate any memory used across multiple iterations of statcollect( ) from the PER_COMMAND pool and free it as soon as possible. Allocate any memory *not* used across multiple invocations of **statcollect( )** from the PER_ROUTINE memory pool.

Use the **mi_fp_setfuncstate( )** function to save a pointer to the user-state memory in the **MI_FPARAM** structure of your **statcollect( )** function. For more information, see "Saving a User State" on page 9-8.

SET_RETONE in statcollect( ): For each row of a table, the **statcollect( )** function collects the statistics data for the column that has the user-defined data type. When the iterator-status constant is SET_RETONE, the database server has invoked the **statcollect( )** function on a single row of the table on which statistics is being

gathered. At this point, statcollect( ) reads the column value from the first argument and places it into the statistics-collection structure.

The **statcollect( )** function processes the statistics on a row-by-row basis; that is, for each iterator status of SET_RETONE, **statcollect( )** merges the current column value into the statistics data in the internal statistics-collection structure. Therefore, the **statcollect( )** function must perform the following tasks:

- Obtain the address of the statistics-collection structure from the user state of the **MI_FPARAM** structure with the **mi_fp_funcstate( )** function.
- Compare the current column value with the current maximum and minimum values (if maximum and minimum are desired).
- Merge the current column value into the distribution data in the statistics-collection structure.
- Handle a NULL column value as appropriate for your distribution.

--- DBDK ---

The SET_RETONE case in the *Opaque***StatCollect( )** function (where *Opaque* is the name of your opaque data type) that BladeSmith generates automatically calls the *Opaque*_**SetMinValue( )** and *Opaque*_**SetMinValue( )** functions to compare the current column value with the existing minimum and maximum. It then calls the *Opaque*_**Histogram( )** function to merge the column value into the distribution array of the *Opaque*_**stat_t** statistics-collection structure. However, you must provide this code within the *Opaque*_**SetMinValue( )** and *Opaque*_**Histogram( )** functions to perform the actual comparisons and distribution for your *Opaque* data type.

--- End of DBDK ---

SET_END in statcollect( ):   After all rows are processed, **statcollect( )** must transfer the statistics data from its statistics-collection structure into the predefined opaque type, **stat**. It is **stat** data that the UPDATE STATISTICS statement stores in the **encdat** column of the **sysdistrib** system catalog table.

The **stat** data type is a multirepresentational opaque data type; that is, it holds statistics data within its internal structure until the data reaches a predefined threshold. If the statistics data exceeds this threshold, the **stat** data type stores the data in a smart large object. In support of the multirepresentational data, the **stat** data type provides the following functions:

- An **assign( )** support function, which is responsible for determining whether or not the statistics data is to be stored in a smart large object

  If the data exceeds the predefined threshold, this **assign( )** function creates the smart large object and increments its reference count. The database server calls this **assign( )** function just before it inserts the **mi_statret** structure into the **encdat** column of the **sysdistrib** table.

- A **destroy( )** support function, which is responsible for deleting any smart large object that might exist to hold the statistics data

  The database server calls this **destroy( )** function just before it deletes a row from the **sysdistrib** system catalog table in response to the DROP DISTRIBUTION clause of the UPDATE STATISTICS statement.

For UPDATE STATISTICS to be able to store the distribution data in the **encdat** column, the **statcollect( )** function must copy its statistics-collection structure into the **stat** data type.

The internal structure of the **stat** opaque type is a C language structure named mi_statret. The **stat** support functions handle most of the interaction with the **mi_statret** structure; however, your **statcollect( )** function must fill in the **mi_statret** multirepresentational fields.

For an exact declaration of mi_statret, see the milo.h header file. This header file also provides the following useful declarations.

Declaration	Purpose
mi_stat_buf	#define for the **statdata.buffer** field
mi_stat_mr	#define for the **statdata.mr** field
MI_STATMAXLEN	Constant for the size of the **statdata.buffer** field
mi_stat_hdrsize	Size of the information in the **mi_statret** structure that is *not* holding the statistics data (size of all fields *except* the **statdata** field)

The **assign( )** and **destroy( )** support functions of the **stat** opaque type determine whether to store the distribution data directly in the **encdat** column or in a smart large object. In the latter case, the **encdat** column stores the LO handle of the smart large object. Your **statcollect( )** function can use the MI_STATMAXLEN constant to determine whether it needs to handle multirepresentational data.

The MI_STATMAXLEN constant is the maximum size that the **encdat** column of **sysdistrib** can hold. Therefore, it is the maximum size of the **statdata.buffer** array. If your distribution data has a size less than MI_STATMAXLEN, you can take the following actions:

- Copy the data from the statistics-collection structure directly into the **statdata.buffer** field.
- Set the **statdata.szind** field to MI_MULTIREP_SMALL to indicate that the multirepresentational data is not stored in a smart large object but is in the **mi_statret** structure.

The **assign( )** and **destroy( )** support functions of the **stat** opaque type take care of determining whether to store the distribution data directly in the **encdat** column or in a smart large object whose LO handle is stored in the **encdat** column.

If your distribution data *exceeds* MI_STATMAXLEN, your **statcollect( )** function must handle the multirepresentational data itself, with the following steps:

1. Create a new smart large object.
2. Copy the data from the statistics-collection structure into the new smart large object.
3. Copy the LO handle of this smart large object into the **statdata.mr.mr_lo_struct.mr_s_lo** field.
4. Set the **statdata.szind** field to MI_MULTIREP_LARGE to indicate that the multirepresentational data is stored in a smart large object.

Registering the statcollect( ) Function

As with any user-defined function, you register the statistics-collection function with the CREATE FUNCTION statement. The registration of this function has the following requirements:

- You *must* name this user-defined function **statcollect**.

The database server handles routine resolution based on the data type of the first argument to **statcollect( )**. If the name of your C statistics-collection function is not **statcollect( )**, specify the C function name in the EXTERNAL NAME clause.

- You *must* declare the **statcollect( )** function with HANDLESNULLS routine modifier.

  Your statistics-collection function can choose whether to include the NULL value in the statistics data that it generates.

- The data types of the parameters *must* be as follows.

Parameter Number	Parameter Data Type
1	SQL name for the user-defined data type
2	FLOAT
3	FLOAT

- Do *not* include the declaration of the **MI_FPARAM** structure in the SQL registration.
- The function must return a value of type **stat**.

The following CREATE FUNCTION statement registers the statistics-collection function that Figure 16-30 on page 16-42 declares:

```
CREATE FUNCTION statcollect(ll_arg longlong, num_rows FLOAT,
   resolution FLOAT)
RETURNING stat
WITH (HANDLESNULLS)
EXTERNAL NAME '/usr/udrs/bin/longlong.so(stat_collect_ll)'
LANGUAGE C;
```

After you register the **statcollect( )** function, make sure those with the DBA privilege or the table owner has the Execute privilege on the function.

Executing the UPDATE STATISTICS Statement

To collect user-defined statistics, run the UPDATE STATISTICS statement in HIGH or MEDIUM mode. The syntax of UPDATE STATISTICS is the same for user-defined data types as for built-in data types. However, when UPDATE STATISTICS collects statistics for a user-defined type, it does not automatically determine the minimum and maximum column values (stored in the **colmin** and **colmax** columns of the **syscolumns** system catalog table). Your **statcollect( )** function can explicitly calculate these values if desired.

The **statcollect( )** function executes once for every row that the database server scans during UPDATE STATISTICS. Therefore, a database table must contain *more than one row* before the database server calls any **statcollect( )** functions.

The number of rows that the database server scans depends on the mode and the confidence level. Executing UPDATE STATISTICS in HIGH mode causes the database server to scan all rows in the table. In MEDIUM mode the database server chooses the number of rows to scan based on the confidence level. The higher the confidence level, the higher the number of rows that the database server scans. For general information about UPDATE STATISTICS, see the *IBM Informix Guide to SQL: Syntax*.

For example, if the **mytable** table contains a column of type **Box**, the following UPDATE STATISTICS statement collects user-defined statistics for all columns of **mytable**, including any columns with user-defined statistics defined:

```
UPDATE STATISTICS HIGH FOR TABLE mytable;
```

If the **mytable** table contains columns with any data types that require
user-defined statistics and you do *not* define this statistics collection, the UPDATE
STATISTICS statement does not collect statistics for the column.

Important: The statistics that the database server collects might require a smart
large object for storage. For the database server to use user-defined
statistics, the configuration parameter SYSSBSPACENAME must be set
in the ONCONFIG file *before* the database server is initialized. This
configuration parameter must specify the name of an existing sbspace.
If SBSSPACENAME is not set, the database server might not be able to
collect the specified statistics.

Using User-Defined Statistics

The user-defined statistics information in **sysdistrib** system catalog table is used in
the following ways:

- The **dbschema** utility with its **-hd** option uses these statistics to display statistics
data for tables.
- The query optimizer uses statistics to obtain a best guess for queries on the
user-defined data type column.

Displaying Statistics Data

The **dbschema** utility with its **-hd** option displays statistics data for tables in your
database. It can automatically display statistics for all columns with built-in data
types (except TEXT and BYTE). It cannot automatically collect statistics for
columns with user-defined data types because it does not know the structure of
these data types.

For **dbschema -hd** to display statistics for a column with a user-defined data type,
you must write a user-defined function named **statprint( )** that generates text
output of the statistics collected for your user-defined data type. The **dbschema
-hd** command obtains the user-defined statistics from the **encdat** column of the
sysdistrib system catalog table. The **encdat** column stores the statistics data in the
stat opaque type. Therefore, **dbschema** must call the **statprint( )** function for your
user-defined data type to convert the statistics data from the **stat** data type to an
LVARCHAR value that can be displayed.

To provide display statistics for your user-defined data type, you must:

- Define a C statistics-display function to implement the statistics display.
- Convert the user-defined statistics for the column to text output within
statistics-display function.
- Register this C function as a **statprint( )** user-defined function.

Defining a Statistics-Display Function: When you declare your statistics-display
function, it must have the following signature:

```
mi_lvarchar *statprint(udt_arg, stat_arg)
    udt_type *udt_arg;
    mi_statret *stat_arg;
```

udt_arg   is a pointer to a dummy argument. The database server uses this
argument to resolve the function and to pass in column values.

stat_arg   is a pointer to the **mi_statret** structure that contains the statistics
information for the user-defined data type.

BladeSmith automatically generates an *Opaque*StatPrint( ) function (in which
Opaque is the name of your opaque data type) with the following declaration:

```
mi_lvarchar *OpaqueStatCollect(Gen_dummy, Gen_bvin)
   void *Gen_dummy;
   mi_lvarchar *Gen_bvin;
```

If this declaration is not appropriate for your opaque type, you must customize the
*Opaque*StatPrint( ) function.

Creating the ASCII Histogram: The **statprint( )** function converts the statistics
data stored in the **stat** data type to an LVARCHAR value that the database server
can use to display information. The **stat** data type is a multirepresentational data
type that the database server uses to store statistics data in the **encdat** column of
the **sysdistrib** system catalog table.

Registering the statprint( ) Function: As with any user-defined function, you
register the statistics-display function with the CREATE FUNCTION statement.
The registration of this function has the following requirements:

- You *must* name this user-defined function **statprint**.

  The database server handles routine resolution based on the data type of the
  first argument to **statprint( )**. If the name of your C statistics-collection function
  is not **statprint( )**, specify the C function name in the EXTERNAL NAME clause.

- The data types of the parameters must be as follows.

Parameter Number	Parameter Data Type
1	SQL name for the user-defined data type
2	stat

- The function must return a value of type LVARCHAR.

The following CREATE FUNCTION statement registers a statistics-display
function:

```
CREATE FUNCTION statprint(ll_arg longlong, num_rows stat)
RETURNING LVARCHAR
EXTERNAL NAME '/usr/udrs/bin/longlong.so(stat_print_ll)'
LANGUAGE C;
```

After you register the **statprint( )** function, make sure those with the DBA
privilege and the table owner have the Execute privilege for the function.

Using User-Defined Statistics in a Query

For SQL statements that use user-defined data types, the optimizer can call custom
selectivity and cost functions. Selectivity and cost functions might need to use
statistics about the nature of the data in a column. When you create the
statcollect( ) function that collects statistics for a UDT, the database server executes
this function automatically when a user runs the UPDATE STATISTICS statement
with the MEDIUM or HIGH keyword.

The statistics that the database server collects might require a smart large object for
storage. The configuration parameter SBSSPACENAME specifies an sbspace for
storing this information. If SBSSPACENAME is not set, the database server might
not be able to collect the specified statistics.

The query optimizer can use data distributions when it assesses the selectivity of a query filter. The *selectivity* is the number of rows that the filter will return. For queries that involve columns with built-in data types, the database server uses data distributions to automatically determine selectivity for the following kinds of filters:

- Relational-operator functions (lessthan( ), ...)
- Boolean built-in operator functions: like( ), matches( )

Important: The query optimizer can only use data distributions if the UPDATE STATISTICS statement has collected these distributions in the **sysdistrib** system catalog table.

However, if the query involves columns with user-defined data types, you must provide the following information for the query optimizer to be able to determine the filter selectivity:

1. Write a user-defined function to implement the appropriate operator function.

   For user-defined types, these built-in operator functions do not automatically exist. You must write versions of these functions that handle your user-defined type.

2. Write a selectivity function for the operator function to provide the optimizer with a selectivity value.

   Selectivity and cost functions might need to use statistics about the nature of the data in a column. If you want these selectivity functions to use data distributions, take the following actions:

   - Provide user-defined statistics so that the UPDATE STATISTICS statement saves the data distributions in the **sysdistrib** system catalog table.

   - Access the **sysdistrib** table from within the selectivity function to obtain the data distributions for the column.

   For more information on how to write and register selectivity functions, see "Writing Selectivity and Cost Functions" on page 15-54.

Optimizing Queries

The WHERE clause of the SELECT statement controls the amount of information that the query evaluates. This clause can consist of a comparison condition, which evaluates to a BOOLEAN value. Therefore, a comparison condition can contain a Boolean function; that is, it can contain a user-defined function that returns a BOOLEAN value. Boolean functions can act as filters in queries, as Table 16-6 shows.

Table 16-6. Boolean Functions Valid in a Comparison Condition

Comparison Condition	Operator Symbol	Associated User-Defined Function
Relational operator	=, !=, <> <, <= >, >=	equal( ), notequal( ), notequal( ) lessthan( ), lessthanorequal( ) greaterthan( ), greaterthanorequal( )
LIKE, MATCHES	None	like( ), matches( )
Boolean function	None	Name of a user-defined function that returns a BOOLEAN value

The Boolean functions in Table 16-6 can act as filters in queries. To optimize queries that use these functions as filters, you can define the following UDR-optimization functions.

Type of Optimization	Description
Negator function	Calculate the NOT condition of the Boolean expression
Selectivity and cost functions	Provide an estimate of the number of rows that the filter will return

Tip: A WHERE clause can also consist of a condition with a subquery. However, conditions with subqueries do not evaluate to a Boolean function. Therefore, they do not require UDR-optimization functions. For more information on conditions with subqueries, see your *IBM Informix Performance Guide* and the Condition segment of the *IBM Informix Guide to SQL: Syntax*.

Query Plans

The optimizer uses the cost and selectivity information to help determine the best query plan for a query. In particular, the optimizer uses this information to obtain the following query and cost estimates:

- Number of rows to retrieve from a table

  This estimated number of rows is based on the *selectivity* of each filter within the WHERE clause of the query.

- Amount of resources that the query requires

  The *cost* is an estimate of the total cost of resource usage for executing the query filter.

The following kinds of user-defined functions are Boolean expressions:

- Built-in operator functions:
  - relational-operator functions, such as lessthan( )
  - Boolean built-in operator functions, such as like( ) and matches( )
- End-user functions that return a BOOLEAN value

Because these user-defined functions are Boolean expressions, they can act as filters in queries. You can optimize these Boolean-expression functions as follows.

Type of Optimization	Description
Negator function	Calculate the NOT condition of the Boolean expression
Selectivity and cost functions	Provide an estimate of the number of rows that the filter will return

Both the cost and selectivity of a UDR can dramatically affect the performance of a particular query plan. For example, in a join between tables, it is often advantageous to have the tables with the most selective filters as the outer tables to reduce the number of rows that flow through the intermediate parts of the query plan.

Selectivity Functions

The optimizer bases query-cost estimates on the number of rows to be retrieved from each table. In turn, the estimated number of rows is based on the *selectivity* of each conditional expression that is used within the WHERE clause. A conditional expression that is used to select rows is termed a *filter*.

The optimizer can use data distributions to calculate selectivities for the filters in a query. However, in the absence of data distributions, the database server calculates

selectivities for filters of different types based on table indexes. The following table lists some of the selectivities that the optimizer assigns to filters of different types.

Filter Expression	Selectivity (F)
any-col IS NULL	F = 1/10
any-col = any-expression	F = 1/10
any-col > any-expression	F = 1/3
any-col < any-expression	F = 1/3
any-col MATCHES any-expression	F = 1/5
any-col LIKE any-expression	F = 1/5
...	

Selectivities calculated using data distributions are even more accurate than the ones that the preceding table shows, as follows:

- Your *IBM Informix Performance Guide* describes the filter expressions that can appear in WHERE clauses with their selectivities *when no data distributions exist* for a column (*any-col*). These selectivities are those that the database server calculates by default.

- The UPDATE STATISTICS statement can generate statistics (data distributions) for columns of built-in data types. However, it cannot generate data distributions for columns of user-defined data types.

- Columns of user-defined types require implementation of *user-defined statistics* for UPDATE STATISTICS to generate statistics (for example, for it to store data distributions in **sysdistrib**).

Query filters can include user-defined functions. You can improve selectivity of filters that include user-defined functions with the following features:

- Functional indexes

  You can create a functional index on the resulting values of a user-defined function on one or more columns. The function can be a built-in function or a user-defined function. When you create a functional index, the database server computes the return values of the function and stores them in the index. The database server can locate the return value of the function in an appropriate index without executing the function for each qualifying column.

- User-defined selectivity functions

  You can write a user-defined selectivity function that calculates the expected fraction of rows that qualify for a particular user-defined function that acts as a filter.

- An end-user function

  For queries that use an end-user function as a filter, you can improve performance by writing a selectivity function for this end-user function.

- An operator function

  For queries that use relational operators (<,>, ...) as filters, you can improve performance by writing a selectivity function for the associated operator function (**lessthan( )**, **greaterthan( )**, ...). *For built-in types*, the relational-operator functions are built-in functions. They have selectivity functions that can use data distributions, which the UPDATE STATISTICS statement can automatically generate.

  For user-defined types, relational-operator functions do not automatically exist. You must write versions of these functions that handle your user-defined type.

In addition, you must write any selectivity functions. If you want these selectivity functions to use data distributions, you must take the following actions:

– Provide user-defined statistics so that UPDATE STATISTICS saves the data distributions in the sysdistrib system catalog table.

– Access the sysdistrib system catalog table from within the selectivity function to obtain the data distributions for the column.

Part 5. Appendixes

Appendix A. Writing a Client LIBMI Application

This appendix outlines the following implementation issues for writing a client LIBMI application:

- How to manage memory with DataBlade API memory-management functions
- How to access operating-system files

```
┌──────────────────────────── Server Only ────────────────────────────┐

This appendix covers topics specific to the creation of a client LIBMI application.
This material does not necessarily apply to the creation of C user-defined routines
(UDRs). For information specific to the creation of C UDRs, see Chapter 13,
"Writing a User-Defined Routine," on page 13-1.

└──────────────────────────── End of Server Only ─────────────────────┘
```

Managing Memory in Client LIBMI Applications

When a DataBlade API module needs to perform dynamic memory allocation, it must do so from user memory. The following table shows the memory-management functions that the DataBlade API provides for memory operations for user memory.

Memory Duration	Memory Operation	Function Name
Not applicable	Constructor	**mi_alloc( )**, **mi_dalloc( )**, **mi_realloc( )**, **mi_zalloc( )**
	Destructor	**mi_free( )**

A client LIBMI application allocates user memory from the process of the client LIBMI application. In a client LIBMI application, the DataBlade API memory-management functions perform the same type of allocation as operating-system memory functions such as **malloc( )** and **free( )**. Therefore, use of the DataBlade API memory-management functions is optional in a client LIBMI application. However, use of the DataBlade API memory-management functions to ensure consistency and portability of code between client and server DataBlade API modules is recommended.

Tip: To use these DataBlade API memory-management functions, be sure to include the **mi.h** header file in the appropriate source files of your client LIBMI application.

Allocating User Memory

To handle dynamic memory allocation of user memory, use one of the following DataBlade API memory-management functions.

Memory-Allocation Task	DataBlade API Function
To allocate user memory	**mi_alloc( )**

Memory-Allocation Task	DataBlade API Function
To allocate user memory with a specified memory duration (memory duration is ignored)	**mi_dalloc( )**
To allocate user memory that is filled with zeros	**mi_zalloc( )**
To change the size of existing memory or allocate new user memory	**mi_realloc( )**

In client LIBMI applications, **mi_dalloc( )** works exactly like **malloc( )**: storage is allocated on the heap of the client process. However, this memory has *no* memory duration associated with it; that is, the database server does *not* automatically free this memory. Therefore, the client LIBMI application *must* use **mi_free** to free explicitly all allocations that **mi_dalloc( )** makes.

The **mi_alloc( )** and **mi_zalloc( )** functions return a pointer to the newly allocated memory. Cast this pointer to match the structure of the user-defined buffer or structure that you allocate. For example, the following call to **mi_dalloc( )** casts the pointer to the allocated memory as a pointer to a structure named **func_info** and uses this pointer to access the **count_fld** of the **func_info** structure:

```
#include mitypes.h
...
struct func_info *fi_ptr;
mi_integer count;
...
fi_ptr = (func_info *)mi_dalloc(sizeof(func_info),
    PER_COMMAND);
fi_ptr->count_fld = 3;
```

The **mi_realloc( )** function accepts a pointer to existing memory and a parameter specifying the number of bytes reallocate to that memory. The function returns a pointer to the reallocated memory. If the pointer to existing memory is NULL, then **mi_realloc( )** allocates new memory in the same way as **mi_alloc( )**.

The **mi_switch_mem_duration( )** function has no effect when it is invoked in a client LIBMI application. Client LIBMI applications ignore memory duration.

Deallocating User Memory

The database server does *not* perform any automatic reclamation of user memory in a client LIBMI application. Therefore, the client LIBMI application *must* use **mi_free** to explicitly free all allocations that **mi_alloc( )** makes.

User memory remains valid until whichever of the following events occurs first:
- The **mi_free( )** function frees the memory.
- The **mi_close( )** function closes the current connection.
- The client LIBMI application ends.

To conserve resources, use the **mi_free( )** function to explicitly deallocate the user memory once your DataBlade API module no longer needs it. The **mi_free( )** function is the destructor function for user memory.

Important: Use **mi_free( )** only for user memory that you have explicitly allocated with **mi_alloc( )**, **mi_dalloc( )**, or **mi_zalloc( )**. Do not use this function to free structures that other DataBlade API functions allocate.

Keep the following restrictions in mind about memory deallocation:

- Do *not* free user memory that you allocate for the return value of a UDR.
- Do *not* free memory until you are finished accessing the memory.
- Do *not* use **mi_free( )** to deallocate memory that you have not explicitly allocated.
- Do *not* use **mi_free( )** for data type structures that other DataBlade API constructor functions allocate.
- Do *not* attempt to free user memory after its memory duration expires.
- Reuse memory whenever possible. Do not repeat calls to allocation functions if you can reuse the memory for another task.

Accessing Operating-System Files in Client LIBMI Applications

In a client LIBMI application, the DataBlade API file-access functions perform the same type of task as operating-system file-management functions such as **open( )** and **close( )**. Table 13-7 on page 13-52 shows the basic file operations with the DataBlade API file-access functions that perform them and the analogous operating-system calls for these file operations.

Use of the DataBlade API file-access functions is optional in a client LIBMI application. However, use of the DataBlade API file-access functions to ensure consistency and portability of code between client and server DataBlade API modules is recommended.

Tip: To use these DataBlade API file-access functions, be sure to include the **mi.h** header file in the appropriate source files of your client LIBMI application.

For DataBlade API modules that you design to run in both client LIBMI applications and UDRs, use the DataBlade API file-access functions. The behavior of these functions in client LIBMI applications is basically the same as in C UDRs. For a description of these files, see "Access to Operating-System Files" on page 13-52.

The main difference in behavior of the DataBlade API file-access functions is that the **mi_open( )** function opens files on the client computer, not the server computer. The filename that you specify to **mi_open( )** is relative to the client computer.

Handling Transactions

For databases that use logging, a client LIBMI application specifies the start and end of each transaction. For these databases, an SQL statement is always part of a transaction. The type of transaction that the SQL statement is part of is based on the type of database and whether it uses transaction logging, as Table 12-1 on page 12-7 shows.

Appendix B. Accessibility

IBM strives to provide products with usable access for everyone, regardless of age or ability.

Accessibility features for IBM Informix Dynamic Server

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility Features

The following list includes the major accessibility features in IBM Informix Dynamic Server. These features support:

- Keyboard-only operation.
- Interfaces that are commonly used by screen readers.
- The attachment of alternative input and output devices.

Tip: The IBM Informix Dynamic Server Information Center and its related publications are accessibility-enabled for the IBM Home Page Reader. You can operate all features using the keyboard instead of the mouse.

Keyboard Navigation

This product uses standard Microsoft Windows navigation keys.

Related Accessibility Information

IBM is committed to making our documentation accessible to persons with disabilities. Our publications are available in HTML format so that they can be accessed with assistive technology such as screen reader software. The syntax diagrams in our publications are available in dotted decimal format.

You can view the publications for IBM Informix Dynamic Server in Adobe Portable Document Format (PDF) using the Adobe Acrobat Reader.

IBM and Accessibility

See the *IBM Accessibility Center* at http://www.ibm.com/able for more information about the commitment that IBM has to accessibility.

B-1

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

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

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol ($^®$ or ^{TM}), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml.

Adobe, Acrobat, Portable Document Format (PDF), and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

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

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Index

Special characters

__myErrors__ trace class   12-30
_open(Windows) system call   13-54
-, (hyphen), as formatting character   3-21
, (comma symbol)   3-2, 3-9, 3-17, 3-21
; (semicolon symbol)   8-7, 8-32
? (question mark), input-parameter indicator   8-12
. (period symbol)   3-9, 3-17, 3-21
.bld file extension   12-11
.dll file extension   12-13
.dsw file extension   12-11
.mak file extension   12-11
.o file extension   12-13
.so file extension   12-12
.trc file extension   12-34
$ (dollar sign)   3-9, 3-22
* (asterisk symbol), as formatting character   3-21
& (ampersand symbol)   3-21
# (pound sign)   3-21
+ (plus sign)   3-21

A

accept() system call   13-21
accessibility   B-1
   keyboard   B-1
   shortcut keys   B-1
Account name.
   See User account.
Aggregate algorithm   15-16
Aggregate functions
   See also Built-in aggregate function; User-defined aggregate.
   built-in   15-12
   creating   15-11
   overloaded   15-12
   user-defined   15-16
Aggregate state
   See also User-defined aggregate.
   allocating a new   15-33
   deallocating   15-33
   defined   15-16, 15-17
   determining   15-17
   nonsimple   15-28, 15-29
   opaque-type   15-31
   simple   15-27, 15-28
   single-valued   15-30
Aggregate support function
   defined   15-16, 15-18
   determining required   15-25
   summary of   15-18
   writing   15-18
AIO VP.
   See Asynchronous I/O virtual-processor (AIO VP) class.
alarm() system call   13-27
ALIGNMENT opaque-type modifier   16-6
Alignment.
   See Type alignment.
All-events callback   10-25, 10-56
Allocation extent size   6-36
ALTER FUNCTION statement   12-36

ALTER PROCEDURE statement   12-36
ALTER ROUTINE statement   12-36
ALTER TABLE statement   6-51, 8-35
Ampersand symbol (&)   3-21
ANSI SQL standards
   ANSI-compliant database   8-23
   date and/or time-string format   4-13
   interval-string format   4-13
   runtime-error values   10-23
   SQLSTATE class values   10-22
   warning values   10-23
Arithmetic operations
   See also Nonarithmetic operations.
   date and/or time values   4-15
   decimal values   3-16
   fixed-point values   2-12
   INT8 values   3-8
assign() support function
   defined   16-37, 16-38
   incrementing the reference count   6-43, 6-57
Asterisk symbol (*), as formatting character   3-21
Asynchronous I/O virtual-processor (AIO VP) class   13-16, 13-19, 13-21
Availability   13-17
ax_reg() function   11-12
ax_unreg() function   11-13

B

BEGIN WORK statement   6-12, 12-7, 12-8, 14-15
Binary operator   15-29, 15-40
Binary representation
   Boolean data   2-30, 8-9
   character data   2-9, 2-11, 8-8
   collection   5-2, 8-10, 8-53
   column values in   8-44, 8-49, 8-50, 8-52
   date and/or time data   4-7, 4-13, 8-9
   date data   4-2, 4-3, 8-9
   decimal data   3-10, 3-12, 3-14, 8-9
   defined   8-8
   distinct data type   8-10
   fixed-length opaque type   8-10
   fixed-point data   3-10
   floating-point data   3-17, 8-9
   input parameters   8-28
   INT8 (mi_int8)   3-6, 3-7, 8-9
   INTEGER (mi_integer)   3-4, 8-9
   integer data   3-2, 8-9
   interval data   4-8, 4-13, 8-9
   LO handle   6-4, 6-60, 8-9, 8-48
   mi_exec_prepared_statement() results   8-30
   mi_exec() results   8-10
   mi_open_prepared_statement() results   8-30
   monetary data   3-11, 3-12, 3-14, 8-9
   opaque type   16-3, 16-11, 16-16, 16-17, 16-21, 16-22, 16-29, 16-36
   row type   5-29, 8-9
   SMALLINT (mi_smallint)   3-3, 8-9
   varying-length opaque type   8-10
bind() system call   13-21
BITVARYING data type   1-10, 2-28

M

User-defined aggregates *(continued)*
   sample   15-37
   set-up argument   15-34
   support functions.
      See Aggregate support function.
User-defined error structure
   allocating   10-33
   defined   10-32
   defining   10-33
   memory duration of   10-33, 10-35
   sample   10-33
User-defined function
   See also Noncursor function; Routine return value;
     Special-purpose function; User-defined routine (UDR).
   Boolean   15-53, 15-60
   cast function.
      See Cast function.
   commutator   15-60
   cost   15-54
   defined   1-2
   defining a return value   13-11
   handling NULL return value   13-13
   iterator function.
      See Iterator function.
   multiple return values   13-14
   negator   15-60
   nonvariant   9-25
   obtaining return-value data   9-6
   registering   12-14, 13-15
   return value.
      See Routine return value.
   routine identifier.
      See Routine identifier.
   selectivity   15-54
   variant   8-2, 9-25
User-defined procedure   1-2, 12-14, 13-14
   See User-defined routine (UDR).
User-defined routine (UDR)
   See also DataBlade API module; Routine argument; Routine
     identifier; Routine return value; User-defined function;
     User-defined procedure.
   aborting   10-11, 10-14, 10-26, 10-31, 10-43
   altering   12-36
   argument.
      See Routine argument.
   as calling module   10-11, 10-26
   callback return value   10-13
   calling   9-14
   calling directly   9-13, 12-18, 14-36
   calling implicitly   12-18, 12-20
   calling sequence of   10-14, 10-30, 10-31
   changing   12-36
   character data handling   2-10, 13-6, 13-13
   choosing a virtual-processor class   12-24, 13-16, 13-38
   coding considerations   12-5, 13-2
   column values in   8-45
   commutator   15-60
   compiling   12-11, 12-26
   connection descriptor in registration   10-6
   cost   15-54
   current VP   13-40
   debugging   12-25, 15-64
   defined   1-2
   defining a return value   13-11
   determining stack space of   14-35
   developing   1-4
   development process   12-2

User-defined routine (UDR) *(continued)*
   development tools   12-2
   dropping   12-36
   entry point in shared-object file   12-16
   event handling in   10-3, 10-11, 12-11
   exception handling in   10-11, 10-25
   executing   12-18, 12-23, 13-17, 14-36, 15-64
   executing with Fastpath   9-14, 9-27
   expensive.
      See Expensive UDR.
   file management in   9-25, 13-21, 13-52
   foreign   9-15, 9-16
   function descriptor for   9-17
   generic   9-37
   global variable   13-23
   granting Execute privilege   12-18, 16-40
   granting language privilege   12-16, 16-40
   handling events.
      See Event handling; Exception handling.
   handling NULL argument   9-5, 9-24, 9-27, 12-24, 13-8
   handling NULL return value   9-8, 13-13
   identifier for.
      See Routine identifier.
   ill-behaved   13-17, 13-18
   information about   9-12
   instance.
      See Routine instance.
   invocation.
      See Routine invocation.
   invoking through SQL   9-13
   iterator function.
      See Iterator function.
   local variable   14-35, 14-36
   locking   13-41
   locking in memory   13-42
   locking to VP   13-41
   looking up with Fastpath   9-18
   memory context   14-4
   memory management in   14-1, 14-19, 14-23
   migrating to another VP   14-2
   multiple return values   13-14
   name of.
      See Routine name.
   negator   15-60
   nonvariant   9-25
   obtaining argument data type   9-3
   obtaining argument values   13-5
   obtaining return-value data type   9-6
   opaque-type data handling   2-29, 13-9, 13-14
   optimizing   15-53
   OUT parameters with   13-14
   overloaded routine.
      See Routine overloading.
   parallelizable   9-10, 15-61
   passing mechanisms for.
      See Passing mechanism.
   programming rules   13-31, 13-32
   recursive   14-36
   reexecuting   9-31
   registering   12-14, 15-63, 16-39
   resource-intensive   13-20
   return value.
      See Routine return value.
   routine argument.
      See Routine argument.
   routine identifier.
      See Routine identifier.

Y

IBM ®

Printed in USA

Spine information:

IBM Informix   Version 11.50   IBM Informix DataBlade API Programmer's Guide

IBM