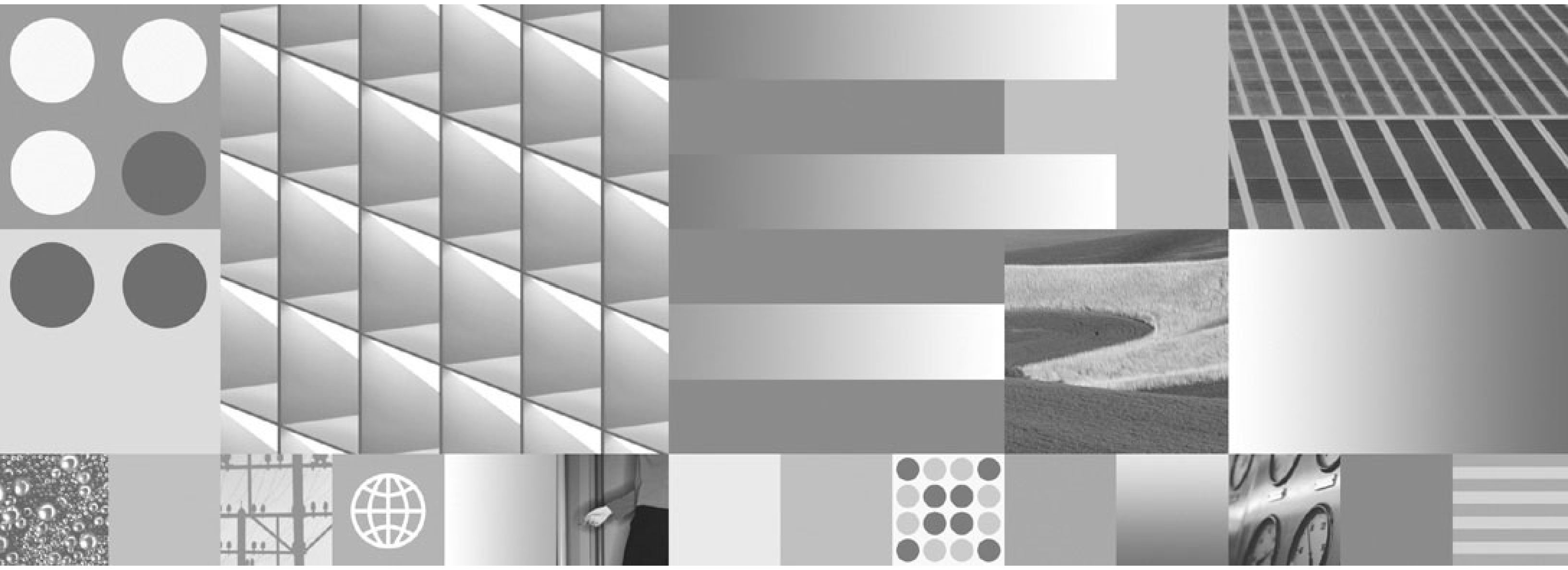
IBM Informix

Version 11.50



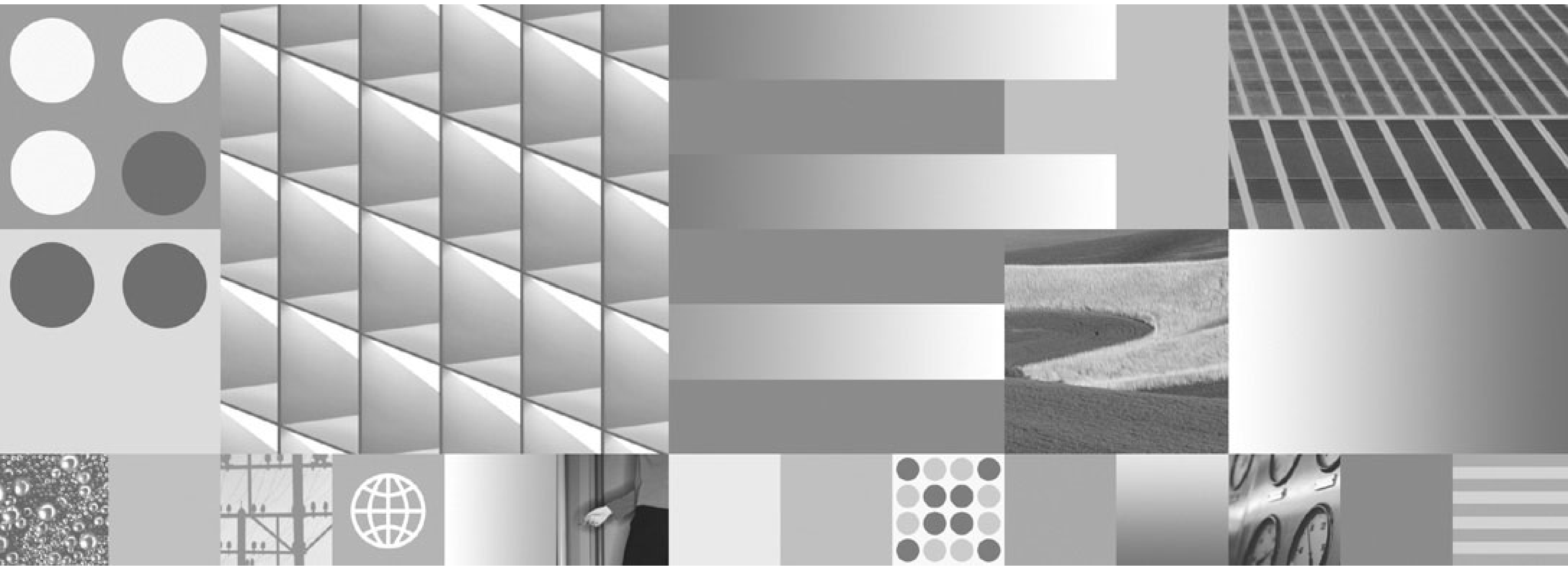
IBM Informix DataBlade API Function Reference

SC23-9428-01

IBM Informix

IBM

Version 11.50



IBM Informix DataBlade API Function Reference

SC23-9428-01

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

This edition replaces SC23-9428-00.

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

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

© Copyright International Business Machines Corporation 1996, 2008.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

© Copyright IBM Corp. 1996, 2008

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 describes the DataBlade® API functions and the subset of Informix ESQL/C functions that the DataBlade API supports. This C-language application programming interface is provided with IBM® Informix® Dynamic Server (IDS). You can use the DataBlade API to develop client LIBMI applications and C user-defined routines that access data in an IDS database.

The *IBM Informix DataBlade API Programmer's Guide*, a companion document to the function reference, explains how to use the functions in client LIBMI applications and user-defined routines.

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 user-defined routines

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

© Copyright IBM Corp. 1996, 2008

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

To use all the functions that this publication describes, you need IBM Informix Dynamic Server (IDS), Version 11.1 or later.

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 GLS (Global Language Support) 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 IBM 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.

What's New in the IBM Informix DataBlade API Function Reference for Dynamic Server, Version 11.50

For a comprehensive list of new features for this release, see the *IBM Informix Dynamic Server Getting Started Guide*. The following changes and enhancements are relevant to this publication.

Table 1. What's New in the IBM Informix DataBlade API Function Reference

Overview	Reference
New return codes for the **mi_hdr_status()** function. The **mi_hdr_status()** function now reports MI_RSS_SECONDARY, MI_SDS_SECONDARY, and MI_HDR_SEC_NODE if the current server is an RSS node, SDS node, and HDR secondary node respectively.	"mi_hdr_status( )" on page 2-328

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.

Convention	Meaning
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.

Chapter 1. Categories of DataBlade API Functions

In This Chapter

This chapter summarizes by categories the functions that the DataBlade API provides. It divides functions that are valid within DataBlade API modules into several functional categories and lists the functions within each category.

The DataBlade API supports functions in the DataBlade API library as well as a subset of functions in the Informix ESQL/C library. This chapter also lists the Informix ESQL/C functions that the DataBlade API supports.

The DataBlade API Function Library

This section lists the functions in the DataBlade API function library by categories, in the following subsections:

- Data Handling
- Session, Thread, and Transaction Management
- SQL Statement Processing
- User-Defined Routine Execution
- Selectivity and Cost Functions
- Memory Management
- Exception Handling
- Smart-Large-Object Interface
- Input and Output Operations
- Tracing (Server)
- Control of the Virtual-Processor Environment (Server)
- Database Management (Client)
- Miscellaneous Information

© Copyright IBM Corp. 1996, 2008

Data Handling

The DataBlade API provides the following categories of functions for handling data.

Function Category	DataBlade API Function	
Obtaining type information	*Type-descriptor accessor functions*:	
	mi_type_align( )	**mi_type_owner( )**
	mi_type_byvalue( )	**mi_type_precision( )**
	mi_type_constructor_typedesc( )	**mi_type_qualifier( )**
	mi_type_element_typedesc( )	**mi_type_scale( )**
	mi_type_full_name( )	**mi_type_typename( )**
	mi_type_length( )	**mi_typedesc_typeid( )**
	mi_type_maxlength( )	
	Type-identifier accessor functions:	
	mi_typeid_equals( )	**mi_typeid_is_list( )**
	mi_typeid_is_builtin( )	**mi_typeid_is_multiset( )**
	mi_typeid_is_collection( )	**mi_typeid_is_row( )**
	mi_typeid_is_complex( )	**mi_typeid_is_set( )**
	mi_typeid_is_distinct( )	
	Other type functions:	
	mi_get_transaction_id( )	**mi_typename_to_typedesc( )**
	mi_type_typedesc( )	**mi_typestring_to_id( )**
	mi_typename_to_id( )	**mi_typestring_to_typedesc( )**
Transferring data types between computers (*server side only*)	**mi_fix_integer( )**	**mi_get_string( )**
	mi_fix_smallint( )	**mi_put_bytes( )**
	mi_get_bytes( )	**mi_put_date( )**
	mi_get_date( )	**mi_put_datetime( )**
	mi_get_datetime( )	**mi_put_decimal( )**
	mi_get_decimal( )	**mi_put_double_precision( )**
	mi_get_double_precision( )	**mi_put_int8( )**
	mi_get_int8( )	**mi_put_integer( )**
	mi_get_integer( )	**mi_put_interval( )**
	mi_get_interval( )	**mi_put_lo_handle( )**
	mi_get_lo_handle( )	**mi_put_money( )**
	mi_get_money( )	**mi_put_real( )**
	mi_get_real( )	**mi_put_smallint( )**
	mi_get_smallint( )	**mi_put_string( )**
Converting data types	**mi_date_to_string( )**	**mi_string_to_date( )**
	mi_datetime_to_string( )	**mi_string_to_datetime( )**
	mi_decimal_to_string( )	**mi_string_to_decimal( )**
	mi_interval_to_string( )	**mi_string_to_interval( )**
	mi_lvarchar_to_string( )	**mi_string_to_lvarchar( )**
	mi_money_to_string( )	**mi_string_to_money( )**
	Deprecated functions:	
	mi_binary_to_date( )	**mi_date_to_binary( )**
	mi_binary_to_datetime( )	**mi_datetime_to_binary( )**
	mi_binary_to_decimal( )	**mi_decimal_to_binary( )**
	mi_binary_to_money( )	**mi_money_to_binary( )**
Converting data between code sets	*Server side*	
	mi_get_string( )	**mi_put_string( )**
	Client side	
	mi_convert_from_codeset( )	**mi_convert_to_codeset( )**

Function Category	DataBlade API Function	

Handling collections	**mi_collection_card( )**
	mi_collection_close( )
	mi_collection_copy( )
	mi_collection_create( )
	mi_collection_delete( )
	mi_collection_fetch( )
		mi_collection_free( )
		mi_collection_insert( )
		mi_collection_open( )
		mi_collection_open_with_options( )
		mi_collection_update( )

Managing varying-length structures	**mi_get_vardata( )**
	mi_get_vardata_align( )
	mi_get_varlen( )
	mi_lvarchar_to_string( )
	mi_new_var( )
	mi_set_vardata( )
	mi_set_vardata_align( )
		mi_set_varlen( )
		mi_set_varptr( )
		mi_string_to_lvarchar( )
		mi_var_copy( )
		mi_var_free( )
		mi_var_to_buffer( )

Obtaining SERIAL values	**mi_last_serial( )**
	mi_last_serial8( )	

Accessing multirepresentational data types	**mi_lo_expand( )**	
	Macros:	
	mi_issmall_data( )	**mi_set_large( )**

Handling NULL values	**mi_column_nullable( )**
	mi_fp_argisnull( )
	mi_fp_returnisnull( )
	mi_fp_setargisnull( )
		mi_fp_setreturnisnull( )
		mi_func_handlesnulls( )
		mi_funcarg_isnull( )
		mi_parameter_nullable( )

Obtaining trigger information	**mi_trigger_event( )**
	mi_trigger_get_new_row( )
	mi_trigger_get_old_row( )
		mi_trigger_level( )
		mi_trigger_name( )
		mi_trigger_tabname( )

Obtaining High-Availability Data Replication information	**mi_hdr_status( )**	

Session, Thread, and Transaction Management

The DataBlade API provides the following categories of functions for managing sessions, threads, and transactions.

Function Category	DataBlade API Function	
Obtaining connection information	*Connection-parameter functions*:	

	mi_get_connection_info( )
	mi_get_default_connection_info( )
		mi_set_default_connection_info( )

	Database-parameter functions:	

	mi_get_database_info( )
	mi_get_default_database_info( )
		mi_set_default_database_info( )

	Session-parameter functions:	
	mi_get_parameter_info( )	**mi_set_parameter_info( )**
	Connection user-data functions:	
	mi_get_connection_user_data( )	**mi_set_connection_user_data( )**
	Other connection functions:	

	mi_get_connection_option( )
	mi_get_id( )
	mi_get_next_sysname( )
		mi_get_serverenv( )
		mi_sysname( )

Function Category	DataBlade API Function	
Establishing a connection	*Server side:*	
	mi_open( )	
	Server side, Advanced function:	
	mi_get_session_connection( )	
	Client side:	

	mi_open( )
	mi_server_connect( )
		mi_server_reconnect( )

Closing a connection	**mi_close( )**	

Initializing the DataBlade API	**mi_init_library( )**
	mi_client_locale( )
	mi_get_default_connection_info( )
	mi_get_default_database_info( )
	mi_get_next_sysname( )
	mi_get_parameter_info( )
		mi_open( )
		mi_register_callback( )
		mi_server_connect( )
		mi_set_parameter_info( )
		mi_sysname( )

Managing Informix threads (*server side only*)	**mi_call( )**
	mi_interrupt_check( )
		mi_yield( )

	Advanced function:	
	mi_call_on_vp( )	

Obtaining transaction and server-processing state changes	**mi_transaction_state( ) (Server)**
	mi_transition_type( )
	mi_get_transaction_id( )	

SQL Statement Processing

The DataBlade API provides the following categories of functions for processing SQL statements.

Function Category	DataBlade API Function	
Sending SQL statements	*Executable-statement functions:*	

	mi_exec( )
	mi_query_finish( )
		mi_query_interrupt( )

	Prepared-statement functions:	

	mi_close_statement( )
	mi_drop_prepared_statement( )
	mi_exec_prepared_statement( )
	mi_fetch_statement( )
		mi_get_cursor_table( )
		mi_open_prepared_statement( )
		mi_prepare( )

Obtaining statement information	*Input-parameter functions:*	

	mi_parameter_count( )
	mi_parameter_nullable( )
	mi_parameter_precision( )
		mi_parameter_scale( )
		mi_parameter_type_id( )
		mi_parameter_type_name( )

	Other statement functions:	

	mi_binary_query( )
	mi_command_is_finished( )
	mi_current_command_name( )
		mi_get_id( )
		mi_get_statement_row_desc( )
		mi_statement_command_name( )

Obtaining result information	**mi_get_result( )**
	mi_result_command_name( )
		mi_result_reference( )
		mi_result_row_count( )

Function Category	DataBlade API Function	
Retrieving rows and row data as well as row types and row-type data	*Row-descriptor functions:*	
	mi_get_row_desc( )	**mi_row_desc_create( )**
	mi_get_row_desc_from_type_desc( )	**mi_row_desc_free( )**
	mi_get_row_desc_without_row( )	
	Row-structure functions:	
	mi_next_row( )	**mi_row_free( )**
	mi_row_create( )	
Retrieving columns	*Column-information functions:*	
	mi_column_count( )	**mi_column_precision( )**
	mi_column_id( )	**mi_column_scale( )**
	mi_column_name( )	**mi_column_type_id( )**
	mi_column_nullable( )	**mi_column_typedesc( )**
	Column-value functions:	
	mi_value( )	**mi_value_by_name( )**
Using save sets	**mi_save_set_count( )**	**mi_save_set_get_last( )**
	mi_save_set_create( )	**mi_save_set_get_next( )**
	mi_save_set_delete( )	**mi_save_set_get_previous( )**
	mi_save_set_destroy( )	**mi_save_set_insert( )**
	mi_save_set_get_first( )	**mi_save_set_member( )**

User-Defined Routine Execution

The DataBlade API provides the following routines for executing user-defined routines (UDRs).

Function Category	DataBlade API Function	
Accessing the **MI_FPARAM** structure	**mi_fp_argisnull( )**	**mi_fp_returnisnull( )**
	mi_fp_arglen( )	**mi_fp_setargisnull( )**
	mi_fp_argprec( )	**mi_fp_setarglen( )**
	mi_fp_argscale( )	**mi_fp_setargprec( )**
	mi_fp_argtype( )	**mi_fp_setargscale( )**
	mi_fp_funcname( )	**mi_fp_setargtype( )**
	mi_fp_funcstate( )	**mi_fp_setfuncid( )**
	mi_fp_getcolid( )	**mi_fp_setfuncstate( )**
	mi_fp_getfuncid( )	**mi_fp_setisdone( )**
	mi_fp_getrow( )	**mi_fp_setnargs( )**
	mi_fp_nargs( )	**mi_fp_setnrets( )**
	mi_fp_nrets( )	**mi_fp_setretlen( )**
	mi_fp_request( )	**mi_fp_setretprec( )**
	mi_fp_retlen( )	**mi_fp_setretscale( )**
	mi_fp_retprec( )	**mi_fp_setrettype( )**
	mi_fp_retscale( )	**mi_fp_setreturnisnull( )**
	mi_fp_rettype( )	
	Advanced functions:	
	mi_fp_setcolid( )	**mi_fparam_get_current( )**
	mi_fp_setrow( )	
Allocating an **MI_FPARAM** structure	**mi_fp_usr_fparam( )**	**mi_fparam_copy( )**
	mi_fparam_allocate( )	**mi_fparam_free( )**
Using the Fastpath interface	**mi_cast_get( )**	**mi_routine_get( )**
	mi_func_desc_by_typeid( )	**mi_routine_get_by_typeid( )**
	mi_routine_end( )	**mi_td_cast_get( )**
	mi_routine_exec( )	

Function Category	DataBlade API Function	

Accessing a function descriptor	**mi_fparam_get( )**
	mi_func_commutator( )
	mi_func_handlesnulls( )
		mi_func_isvariant( )
		mi_func_negator( )
		mi_routine_id_get( )

Execute an operating system command	**mi_system( )**	

Selectivity and Cost Functions

The DataBlade API provides the following routines for accessing the **MI_FUNCARG** data type, which is the data type of all parameters of selectivity and cost functions.

Function Category	DataBlade API Function	

Accessing the **MI_FUNCARG** data type	**mi_funcarg_get_argtype( )**
	mi_funcarg_get_colno( )
	mi_funcarg_get_constant( )
	mi_funcarg_get_datalen( )
	mi_funcarg_get_datatype( )
		mi_funcarg_get_distrib( )
		mi_funcarg_get_routine_id( )
		mi_funcarg_get_routine_name( )
		mi_funcarg_get_tabid( )
		mi_funcarg_isnull( )

Memory Management

The following DataBlade API functions manage memory.

Function Category	DataBlade API Function	

Managing user memory	**mi_alloc( )**
	mi_dalloc( )
	mi_free( )
		mi_realloc( )
		mi_switch_mem_duration( )
		mi_zalloc( )

Managing named memory	**mi_lock_memory( )**
	mi_named_alloc( )
	mi_named_free( )
	mi_named_get( )
		mi_named_zalloc( )
		mi_try_lock_memory( )
		mi_unlock_memory( )

Managing stack memory	**mi_call( )**
	mi_stack_limit( )	

Memory duration	**mi_get_duration_size( )**	**mi_get_memptr_duration( )**

Exception Handling

The DataBlade API provides the following routines for handling exceptions.

Function Category	DataBlade API Function	
Raising a database exception	**mi_db_error_raise( )**	

Accessing an error descriptor	**mi_errmsg( )**
	mi_error_desc_copy( )
	mi_error_desc_destroy( )
	mi_error_desc_finish( )
	mi_error_desc_is_copy( )
		mi_error_desc_next( )
		mi_error_level( )
		mi_error_sqlcode( )
		mi_error_sql_state( )

Using callback functions	**mi_default_callback( )**
	mi_disable_callback( )
	mi_enable_callback( )
		mi_register_callback( )
		mi_retrieve_callback( )
		mi_unregister_callback( )

Smart-Large-Object Interface

The DataBlade API provides the following routines for handling smart large objects.

Function Category	DataBlade API Function	

Creating a smart large object	**mi_lo_copy( )**
	mi_lo_create( )
		mi_lo_from_file( )

	Deprecated function:	
	mi_lo_expand( )	

Performing I/O on a smart large object	**mi_lo_close( )**
	mi_lo_from_buffer( )
	mi_lo_lock( )
	mi_lo_open( )
	mi_lo_read( )
	mi_lo_readwithseek( )
	mi_lo_seek( )
	mi_lo_stat( )
		mi_lo_tell( )
		mi_lo_to_buffer( )
		mi_lo_truncate( )
		mi_lo_unlock( )
		mi_lo_utimes( )
		mi_lo_write( )
		mi_lo_writewithseek( )

Moving smart large objects to and from operating-system files	**mi_lo_filename( )**
	mi_lo_from_file( )
		mi_lo_from_file_by_lofd( )
		mi_lo_to_file( )

Manipulating LO handles	**mi_get_lo_handle( )**
	mi_lo_alter( )
	mi_lo_decrefcount( )
	mi_lo_delete_immediate( )
	mi_lo_filename( )
	mi_lo_from_string( )
	mi_lo_increfcount( )
	mi_lo_invalidate( )
		mi_lo_lolist_create( )
		mi_lo_ptr_cmp( )
		mi_lo_release( )
		mi_lo_to_string( )
		mi_lo_truncate( )
		mi_lo_validate( )
		mi_put_lo_handle( )

Handling LO-specification structures	**mi_lo_colinfo_by_ids( )**
	mi_lo_colinfo_by_name( )
	mi_lo_spec_free( )
	mi_lo_spec_init( )
	mi_lo_specget_def_open_flags( )
	mi_lo_specget_estbytes( )
	mi_lo_specget_extsz( )
	mi_lo_specget_flags( )
		mi_lo_specget_maxbytes( )
		mi_lo_specget_sbspace( )
		mi_lo_specset_def_open_flags( )
		mi_lo_specset_estbytes( )
		mi_lo_specset_extsz( )
		mi_lo_specset_flags( )
		mi_lo_specset_maxbytes( )
		mi_lo_specset_sbspace( )

Handling smart-large-object status	**mi_lo_stat( )**
	mi_lo_stat_atime( )
	mi_lo_stat_cspec( )
	mi_lo_stat_ctime( )
	mi_lo_stat_free( )
		mi_lo_stat_mtime_sec( )
		mi_lo_stat_mtime_usec( )
		mi_lo_stat_refcnt( )
		mi_lo_stat_size( )
		mi_lo_stat_uid( )

Important: In these reference pages the term "smart large object" refers to either a BLOB or CLOB object. References to a single type of smart large object use the data type specification.

Input and Output Operations

The DataBlade API provides the following functions to perform input and output (I/O) operations.

Function Category	DataBlade API Function	
Using the operating-system file interface	**mi_file_close( )**	**mi_file_sync( )**
	mi_file_errno( )	**mi_file_tell( )**
	mi_file_open( )	**mi_file_to_file( )**
	mi_file_read( )	**mi_file_unlink( )**
	mi_file_seek( )	**mi_file_write( )**
	Deprecated function:	
	mi_file_allocate( )	
Using the stream I/O interface (*Server side only*)	**mi_stream_close( )**	**mi_streamread_lo_by_lofd( )**
	mi_stream_eof( )	**mi_streamread_lvarchar( )**
	mi_stream_get_error( )	**mi_streamread_money( )**
	mi_stream_getpos( )	**mi_streamread_real( )**
	mi_stream_init( )	**mi_streamread_row( )**
	mi_stream_length( )	**mi_streamread_smallint( )**
	mi_stream_open_fio( )	**mi_streamread_string( )**
	mi_stream_open_mi_lvarchar( )	**mi_streamwrite_boolean( )**
	mi_stream_open_str( )	**mi_streamwrite_collection( )**
	mi_stream_read( )	**mi_streamwrite_date( )**
	mi_stream_seek( )	**mi_streamwrite_datetime( )**
	mi_stream_set_error( )	**mi_streamwrite_decimal( )**
	mi_stream_setpos( )	**mi_streamwrite_double( )**
	mi_stream_tell( )	**mi_streamwrite_int8( )**
	mi_stream_write( )	**mi_streamwrite_integer( )**
	mi_streamread_boolean( )	**mi_streamwrite_interval( )**
	mi_streamread_collection( )	**mi_streamwrite_lo( )**
	mi_streamread_date( )	**mi_streamwrite_lvarchar( )**
	mi_streamread_datetime( )	**mi_streamwrite_money( )**
	mi_streamread_decimal( )	**mi_streamwrite_real( )**
	mi_streamread_double( )	**mi_streamwrite_row( )**
	mi_streamread_int8( )	**mi_streamwrite_smallint( )**
	mi_streamread_integer( )	**mi_streamwrite_string( )**
	mi_streamread_interval( )	

Tracing (Server)

The DataBlade API provides the following functions for tracing within a user-defined routine.

Function Category	DataBlade API Function	
Tracing	**mi_tracefile_set( )**	**mi_tracelevel_set( )**

Control of the Virtual-Processor Environment (Server)

The DataBlade API provides the following functions for controlling the Virtual-Processor (VP) environment within a user-defined routine (UDR).

Function Category	DataBlade API Function	
Controlling the VP environment	*VP information*	
	mi_vpinfo_classid( )	**mi_vpinfo_vpid( )**
	mi_vpinfo_isnoyield( )	
	VP-class information	
	mi_class_id( )	**mi_class_name( )**
	mi_class_maxvps( )	**mi_class_numvp( )**
	Locking a UDR to a VP	
	mi_module_lock( )	**mi_udr_lock( )**
	Changing the VP environment	
	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.

Database Management (Client)

The DataBlade API provides the following functions for managing databases in a client LIBMI application.

Function Category	DataBlade API Function	
Managing databases	**mi_dbcreate( )**	**mi_get_dbnames( )**
	mi_dbdrop( )	

Miscellaneous Information

The DataBlade API provides the following functions to return miscellaneous information.

Function Category	DataBlade API Function	
Miscellaneous	**mi_client( )**	**mi_get_db_locale( )**
	mi_client_locale( )	

Database Server Version Information

The DataBlade API provides the following functions to return information about the database server.

Function Category	DataBlade API Function	
Version	**mi_server_library_version( )**	**mi_library_version( )**
	mi_version_comparison( )	

The ESQL/C Function Library

The IBM Informix ESQL/C function library provides functions for data conversion of the following data types.

Function Category	IBM Informix ESQL/C Library Function	
Byte	**bycmpr( )**	**byfill( )**
	bycopy( )	**byleng( )**

Function Category	IBM Informix ESQL/C Library Function	

Character processing	**ldchar( )**
	rdownshift( )
	rstod( )
	rstoi( )
	rstol( )
	rupshift( )
		stcat( )
		stchar( )
		stcmpr( )
		stcopy( )
		stleng( )

DECIMAL type
MONEY type
	decadd( )
	deccmp( )
	deccopy( )
	deccvasc( )
	deccvdbl( )
	deccvint( )
	deccvlong( )
	decdiv( )
	dececvt( ) and decfcvt( )
		decmul( )
		decround( )
		decsub( )
		dectoasc( )
		dectodbl( )
		dectoint( )
		dectolong( )
		dectrunc( )
		rfmtdec( )

DATE type	**rdatestr( )**
	rdayofweek( )
	rdefmtdate( )
	rfmtdate( )
	rjulmdy( )
		rleapyear( )
		rmdyjul( )
		rstrdate( )
		rtoday( )

DATETIME type	**dtaddinv( )**
	dtcurrent( )
	dtcvasc( )
	dtcvfmtasc( )
	dtextend( )
		dtsub( )
		dtsubinv( )
		dttoasc( )
		dttofmtasc( )

INTERVAL type	**incvasc( )**
	incvfmtasc( )
	intoasc( )
	intofmtasc( )
		invdivdbl( )
		invdivinv( )
		invextend( )
		invmuldbl( )

INT8 type	**ifx_int8add( )**
	ifx_int8cmp( )
	ifx_int8copy( )
	ifx_int8cvasc( )
	ifx_int8cvdbl( )
	ifx_int8cvdec( )
	ifx_int8cvflt( )
	ifx_int8cvint( )
	ifx_int8cvlong( )
		ifx_int8div( )
		ifx_int8mul( )
		ifx_int8sub( )
		ifx_int8toasc( )
		ifx_int8todbl( )
		ifx_int8todec( )
		ifx_int8toflt( )
		ifx_int8toint( )
		ifx_int8tolong( )

Other C-language data types	**rfmtdouble( )**	**rfmtlong( )**

The IBM Informix ESQL/C function library does provide additional functions. However, any function not listed in the preceding table is *not* valid within a DataBlade API module.

Chapter 2. Function Descriptions

© Copyright IBM Corp. 1996, 2008

In This Chapter

This chapter gives comprehensive reference information about the functions that are valid within DataBlade API modules. It lists the function descriptions in alphabetical order, with syntax, usage information, and return values for the functions.

Important: These function descriptions contain a section titled "Return Values." This section lists the possible return values for the associated DataBlade API function. Whether the calling code actually receives a return value, however, depends on whether the DataBlade API function throws an MI_Exception event when it encounters a runtime error. For more information, see the *IBM Informix DataBlade API*

Programmer's Guide.

ax_reg( )

The **ax_reg( )** function allows DataBlade modules or applications using user-defined routines (UDRs) to register XA-compliant, external data sources (also called *resource managers*) with the Dynamic Server transaction manager. The registration is dynamic and is applicable for the current transaction only. The DataBlade should register participating data sources into each transaction.

Syntax

```
int ax_reg(int rmid,
          XID *xid,
          int4 flags)
```

rmid	is the resource manager ID.
xid	is a valid pointer to the XID data structure, which is defined in the **$INFORMIXDIR/incl/public/xa.h** file. Valid XID information is returned if the **ax_reg( )** function returns `TM_OK`.
flags	must be set to `TMNOFLAGS`. The value for `TMNOFLAGS` is defined in the **$INFORMIXDIR/incl/public/xa.h** file.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **ax_reg( )** function registers the resource manager ID of the XA data source into the current transaction. When the **ax_reg( )** function is called, you should set *flags* to `TMNOFLAGS`.

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( )** unregisters the data source from the transaction.

The caller is responsible for allocating the space to which *xid* points.

The resource manager ID must be present in a row in the **'informix'.sysxadatasources** system catalog table created with this SQL statement:

```
CREATE XADATASOURCE datasourcename USING xadstype
```

An application can use the **mi_xa_get_xadatasource_rmid( )** function to get the resource manager ID.

For more information on this statement, see the *IBM Informix Guide to SQL: Syntax*.

The **ax_reg( )** function must be repeated for each transaction.

Warning: If *xid* does not point to a buffer that is at least as large as the size of an XID, the **ax_reg( )** function can overwrite the caller's data space. In addition, the buffer must be properly aligned on a long word boundary in case structure assignments are performed.

If the function call is successful, the function returns `TM_OK` and the `xid` value. If the function call is not successful, an error appears.

If you receive an error, check for any of the following problems:

1. Make sure the *rmid* value is correct.
2. Make sure memory for *xid* is allocated.
3. Make sure *flags* is set to `TMNOFLAGS`.
4. Make sure that the **ax_reg( )** function is called from within the transaction.
5. Make sure that the **ax_reg( )** function is not called:
   - From the sub-ordinator of a distributed transaction.
   - From within the 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, which creates a type of XA-compliant external data source.

The **mi_xa_register_xadatasource( )** function also allows DataBlade modules to register XA-compliant, external data sources. However, the **ax_reg( )** function and the **mi_xa_register_xadatasource( )** function use different parameters and have different return values.

Return Values

TM_OK	indicates that the data source is registered.
TMER_TMERR	indicates that an error occurred and the data source is not registered.
TMER_INVAL	indicates that invalid arguments were specified.
TMER_PROTO	indicates that the routine was invoked in an improper context.

Related Topics

See the descriptions of **mi_xa_get_xadatasource_rmid( )**, **ax_unreg( )**, **mi_xa_register_xadatasource( )**, and **mi_xa_unregister_xadatasource( )**. Also refer to the ″Distributed Transaction Processing: The XA Specification.″ This is the X/OPEN standard specification that is available on the Internet.

For more information on working with XA data sources, see the *IBM Informix DataBlade API Programmer's Guide*.

ax_unreg( )

The **ax_unreg( )** function allows DataBlade modules or applications using user-defined routines (UDRs) to unregister previously registered XA-compliant, external data sources (also called *resource managers*) from transactions.

Syntax

```
int ax_unreg(int rmid,
                                int4 flags)
```

rmid	is the resource manager ID.
flags	must be set to `TMNOFLAGS`. The value for `TMNOFLAGS` is defined in the **$INFORMIXDIR/incl/public/xa.h** file.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **ax_unreg( )** function unregisters an XA data source from the transaction in which it was previously registered. By default, all XA-compliant external data sources are unregistered at the end of a transaction. Use the **ax_unreg( )** function to unregister the XA data source before the end of the transaction so the data source does not participate in the transaction.

Multiple registrations of the same XA data source in a transaction have the same effect as a single registration. Since Dynamic Server does not maintain a count of the number of times an application has registered, a single call to **ax_unreg( )** unregisters the data source from the transaction.

The resource manager ID must be present in a row in the **'informix'.sysxadatasources** system catalog table that was created with this SQL statement:

```
CREATE XADATASOURCE datasourcename USING xadstype
```

An application can use the **mi_xa_get_xadatasource_rmid( )** function to get the resource manager ID.

When the **ax_unreg()** function is called, you should set *flags* to `TMNOFLAGS`.

If the function call is successful, the function returns `TM_OK`. If the function call is not successful, an error appears.

If you receive an error, check for any of the following problems:

1. Make sure the *rmid* value is correct.
2. Make sure the flags passed as `TMNOFLAGS`.
3. Make sure that the **ax_unreg( )** function is called from within the transaction.
4. Make sure that the **ax_unreg( )** function is not called:
   - From the sub-ordinator of a distributed transaction.
   - From within the 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, which creates a type of XA-compliant external data source.

5. Make sure you are not unregistering an XA data source that is not registered.

The **mi_xa_unregister_xadatasource( )** function also allows DataBlade modules to unregister previously registered XA-compliant, external data sources from transactions. However, the **ax_unreg( )** function and the **mi_xa_unregister_xadatasource( )** function use different parameters and have different return values.

Return Values

TM_OK	indicates that the data sources are unregistered.
TMER_TMERR	indicates that an error occurred and the function was not successful.
TMER_INVAL	indicates that invalid arguments were specified.
TMER_PROTO	indicates that the routine was invoked in an improper context.

Related Topics

See the descriptions of **ax_reg( )**, **mi_xa_get_xadatasource_rmid( )**, **mi_xa_register_xadatasource( )**, and **mi_xa_unregister_xadatasource( )**. Also refer to the ″Distributed Transaction Processing: The XA Specification.″ This is the X/OPEN standard specification that is available on the Internet.

For more information on working with XA data sources, see the *IBM Informix DataBlade API Programmer's Guide*.

biginttoint2( )

The **biginttoint2( )** function converts a BIGINT type number to an int2 type number.

Syntax

```
mint biginttoint2(bigintv, int2p)
 const bigint bigintv
 int2 *int2p
```

bigintv	is a bigint value to convert to an int2 integer value.
int2p	is a pointer to an int variable to contain the result of the conversion.

Return Values

0	The conversion was successful.
<0	The conversion failed.

bigintcvint2( )

The **bigintcvint2( )** function converts an int2 type number to a BIGINT type number.

Syntax

```
mint bigintcvint2(int2v, bigintp)
 const int2 int2v
 bigint *bigintp
```

int2v	is the int2 value to convert to a bigint value.
bigintp	is a pointer to a bigint variable to contain the result of the conversion.

Return Values

0	The conversion was successful.
<0	The conversion failed.

biginttoint4( )

The **biginttoint4( )** function converts a BIGINT type number to an int4 type number.

Syntax

```
mint biginttoint4(bigintv, int4p)
 const bigint bigintv
 int4 *int4p
```

bigintv	is a bigint value to convert to an int4 integer value.
int4p	is a pointer to an int4 variable to contain the result of the conversion.

Return Values

0	The conversion was successful.
<0	The conversion failed.

bigintcvint4( )

The **bigintcvint4( )** function converts an int4 type number to a BIGINT type number.

Syntax

```
mint bigintcvint4(int4v, bigintp)
 const int4 int4v
 bigint *bigintp
```

int4v	is the int4 value to convert to a bigint value.
bigintp	is a pointer to a bigint variable to contain the result of the conversion.

Return Values

0	The conversion was successful.
<0	The conversion failed.

biginttoasc( )

The **biginttoasc( )** function converts a BIGINT type value to a C char type value.

Syntax

```
mint biginttoasc(bigintv, strng_val, len)
 const bigint bigintv
 char *strng_val
 mint len
```

bigintv	is a bigint value to convert to a text string.
strng_val	is a pointer to the first byte of the character buffer to contain the text string.
len	is the size of *strng_val*, in bytes, minus 1 for the null terminator.

Return Values

0	The conversion was successful.
<0	The conversion failed.

bigintcvasc( )

The **bigintcvasc( )** function converts a C char type value to a BIGINT type number.

Syntax

```
mint bigintcvasc(strng_val, len, bigintp)
 const char *strng_val
 mint len
 bigint *bigintp
```

strng_val	is a pointer to a string.
len	is the length of the *strng_val* string.
bigintp	is a pointer to a bigint variable to contain the result of the conversion.

Return Values

0	The conversion was successful.
<0	The conversion failed.

bigintcvdec( )

The **bigintcvdec( )** function converts a decimal type number to a BIGINT type number.

Syntax

```
mint bigintcvdec(decp, bigintp)
 const dec_t *decp
 bigint *bigintp
```

decp	is a pointer to the decimal structure that contains the value to convert to a bigint value.
bigintp	is a pointer to a bigint variable to contain the result of the conversion.

Return Values

0	The conversion was successful.
<0	The conversion failed.

biginttodec( )

The **biginttodec( )** function converts a BIGINT type number to a decimal type number.

Syntax

```
mint biginttodec(bigintv, decp)
 const bigint bigintv
 dec_t *decp
```

bigintv	is a bigint value to convert to decimal.
decp	is a pointer to a decimal variable to contain the result of the conversion.

Return Values

0	The conversion was successful.
<0	The conversion failed.

biginttodbl( )

The **biginttodbl( )** function converts a BIGINT type number to a double type number.

Syntax

```
mint biginttodbl(bigintv, dbl)
 const bigint bigintv
 double *dbl
```

bigintv	is a bigint value to convert to double.
dbl	is a pointer to a double variable to contain the result of the conversion.

Return Values

0	The conversion was successful.
<0	The conversion failed.

bigintcvdbl( )

The **bigintcvdbl( )** function converts a double type number to a BIGINT type number.

Syntax

```
mint bigintcvdbl(dbl, bigintp)
 const double dbl
 bigint *bigintp
```

dbl	is the double value to convert to bigint.
bigintp	is a pointer to a bigint variable to contain the result of the conversion.

Return Values

0	The conversion was successful.
<0	The conversion failed.

biginttoflt( )

The **biginttoflt( )** function converts a BIGINT type number to a float type number.

Syntax

```
mint biginttoflt(bigintv, fltp)
 const bigint bigintv
 float *fltp
```

bigintv	is a bigint value to convert to float.
fltp	is a pointer to a float variable to contain the result of the conversion.

Return Values

0	The conversion was successful.
<0	The conversion failed.

bigintcvflt( )

The **bigintcvflt( )** function converts a float type number to a BIGINT type number.

Syntax

```
mint bigintcvflt(dbl, bigintp)
 const double dbl
 bigint *bigintp
```

dbl	is the float value to convert to bigint.
bigintp	is a pointer to a bigint value to contain the result of the conversion.

Return Values

0	The conversion was successful.
<0	The conversion failed.

bigintcvifx_int8( )

The **bigintcvifx_int8( )** function converts and int8 type number to a BIGINT type number.

Syntax

```
mint bigintcvifx_int8(int8p, bigintp)
 const ifx_int8_t *int8p
 bigint *bigintp
```

int8p	is the int8 value to convert to a bigint value.
bigintp	is a pointer to a bigint variable to contain the result of the conversion.

Return Values

0	The conversion was successful.
<0	The conversion failed.

biginttoifx_int8( )

The **biginttoifx_int8( )** function converts a BIGINT type number to an int8 type number.

Syntax

```
void biginttoifx_int8(bigintv, int8p)
 const bigint bigintv
 ifx_int8_t *int8p
```

bigintv	is a bigint value to convert to int8.
int8p	is a pointer to an int8 structure to contain the result of the conversion.

bycmpr( )

The **bycmpr( )** function compares two groups of contiguous bytes for a given length. This function returns the result of the comparison.

Syntax

```
mint bycmpr(byte1, byte2, length)
   char *byte1;
   char *byte2;
   mint length;
```

byte1	is a pointer to the location at which the first group of contiguous bytes starts.
byte2	is a pointer to the location at which the second group of contiguous bytes starts.
length	is the number of bytes to compare.

Usage

The **bycmpr( )** function performs a byte-by-byte comparison of the two groups of contiguous bytes until it finds a difference or until it compares *length* number of bytes. The **bycmpr( )** function returns an integer whose value (`0`, `-1`, or `+1`) indicates the result of the comparison between the two groups of bytes.

The **bycmpr( )** function subtracts the bytes of the *byte2* group from those of the *byte1* group to accomplish the comparison.

Return Values

0	The two groups are identical.
-1	The *byte1* group is less than the *byte2* group.
+1	The *byte1* group is greater than the *byte2* group.

bycopy( )

The **bycopy( )** function copies a given number of bytes from one location to another.

Syntax

```
void bycopy(from, to, length)
   char *from;
   char *to;
   mint length;
```

from	is a pointer to the first byte of the group of bytes to copy.
to	is a pointer to the first byte of the destination group of bytes. The memory area to which *to* points can overlap the area to which the *from* argument points. In this case, the function does not preserve the value to which *from* points.
length	is the number of bytes to copy.

Important: Take care not to overwrite areas of memory adjacent to the destination area.

byfill( )

The **byfill( )** function fills a specified area with one character.

Syntax

```
void byfill(to, length, ch)
   char *to;
   mint length;
   char ch;
```

to	is a pointer to the first byte of the memory area to fill.
length	is the number of times to repeat the character within the area.
ch	is the character to use to fill the area.

Important: Take care not to overwrite areas of memory adjacent to the area that you want byfill( ) to fill.

byleng( )

The **byleng( )** function returns the number of significant characters in a string, not counting trailing blanks.

Syntax

```
mint byleng(from, count)
   char *from;
   mint count;
```

from	is a pointer to a fixed-length string (not null-terminated).
count	is the number of bytes in the fixed-length string. This does not include trailing blanks.

decadd( )

The **decadd( )** function adds two **decimal** type values.

Syntax

```
mint decadd(n1, n2, sum)
   dec_t *n1;
   dec_t *n2;
   dec_t *sum;
```

n1	is a pointer to the **decimal** structure of the first operand.
n2	is a pointer to the **decimal** structure of the second operand.
sum	is a pointer to a **decimal** structure to contain the sum (*n1* + *n2*).

Usage

The *sum* value can be the same as the value of either *n1* or *n2*.

Return Values

0	The operation was successful.
-1200	The operation resulted in overflow.
-1201	The operation resulted in underflow.

deccmp( )

The **deccmp( )** function compares two **decimal** type numbers.

Syntax

```
mint deccmp(n1, n2)
   dec_t *n1;
   dec_t *n2;
```

n1	is a pointer to the **decimal** structure of the first number to compare.
n2	is a pointer to the **decimal** structure of the second number to compare.

Return Values

-1	The first value is less than the second value.
0	The two values are identical.
1	The first value is greater than the second value.
DECUNKNOWN	Either value is null.

deccopy( )

The **deccopy( )** function copies a value from one **decimal** structure to another.

Syntax

```
void deccopy(source, target)
   dec_t *source;
   dec_t *target;
```

source	is a pointer to the **decimal** structure that contains the value to copy.
target	is a pointer to a **decimal** structure to which to copy the value.

The **deccopy( )** function does not return a status value. To determine the success of the copy operation, look at the contents of the **decimal** structure to which the *target* argument points.

deccvasc( )

The **deccvasc( )** function converts a value held as printable characters in a C **char** type into a **decimal** type number.

Syntax

```
mint deccvasc(strng_val, len, dec_val)
   char *strng_val;
   mint len;
   dec_t *dec_val;
```

strng_val	is a pointer to the string value to convert to a **decimal** value.
len	is the length of the *strng_val* string.
dec_val	is a pointer to a **decimal** structure to contain the result of the conversion.

Usage

The character string, *strng_val*, can contain the following symbols:

- A leading sign, either a plus (+) or minus (-)
- A decimal point, and digits to the right of the decimal point
- An exponent that is preceded by either `e` or `E`. You can precede the exponent by a sign, either a plus (+) or minus (-).

The deccvasc**( )** function ignores leading spaces in the character string.

Return Values

0	The conversion was successful.
-1200	The number is too large to fit into a **decimal** type structure (overflow).
-1201	The number is too small to fit into a **decimal** type structure (underflow).
-1213	The string has non-numeric characters.
-1216	The string has a bad exponent.

deccvdbl( )

The **deccvdbl( )** function converts a C **double** type number into a **decimal** type number.

Syntax

```
mint deccvdbl(dbl_val, dec_val)
   double dbl_val;
   dec_t *dec_val;
```

dbl_val	is the **double** value to convert to a **decimal** value.
dec_val	is a pointer to a **decimal** structure to contain the result of the conversion.

Return Values

0	The conversion was successful.
<0	The conversion failed.

deccvint( )

The **deccvint( )** function converts a C **int** type number into a **decimal** type number.

Syntax

```
mint deccvint(int_val, dec_val)
   mint int_val;
   dec_t *dec_val;
```

int_val	is the **mint** value to convert to a **decimal** value.
dec_val	is a pointer to a **decimal** structure to contain the result of the conversion.

Return Values

0	The conversion was successful.
<0	The conversion failed.

deccvlong( )

The **deccvlong( )** function converts an **int4** type value into a **decimal** type value.

Syntax

```
mint deccvlong(lng_val, dec_val)
   int4 lng_val;
   dec_t *dec_val;
```

lng_val	is the **int4** value to convert to a **decimal** value.
dec_val	is a pointer to a **decimal** structure to contain the result of the conversion.

Return Values

0	The conversion was successful.
<0	The conversion failed.

decdiv( )

The **decdiv( )** function divides two **decimal** type values.

Syntax

```
mint decdiv(n1, n2, quotient)   /* quotient = n1 / n2 */
   dec_t *n1;
   dec_t *n2;
   dec_t *quotient;
```

n1	is a pointer to the **decimal** structure of the first operand.
n2	is a pointer to the **decimal** structure of the second operand.
quotient	is a pointer to a **decimal** structure to contain the quotient of *n1* divided by *n2*.

Usage

The *quotient* value can be the same as the value of either *n1* or *n2*.

Return Values

0	The operation was successful.
-1200	The operation resulted in overflow.
-1201	The operation resulted in underflow.
-1202	The operation attempted to divide by zero.

dececvt( ) and decfcvt( )

The **dececvt( )** and **decfcvt( )** functions are analogous to the subroutines under ECVT(3) in the *UNIX Programmer's Manual*. The **dececvt( )** function works in the same fashion as the **ecvt(3) function**, and the **decfcvt( )** function works in the same fashion as the **fcvt(3) function**. They both convert a **decimal** type number to a C **char** type value.

Syntax

```
char *dececvt(dec_val, ndigit, decpt, sign)
   dec_t *dec_val;
   mint ndigit;
   mint *decpt;
   mint *sign;

char *decfcvt(dec_val, ndigit, decpt, sign)
   dec_t *dec_val;
   mint ndigit;
   mint *decpt;
   mint *sign;
```

dec_val	is a pointer to the **decimal** structure that contains the value to convert.
ndigit	is the length of the ASCII string for **dececvt( )**. It is the number of digits to the right of the decimal point for **decfcvt( )**.
decpt	is a pointer to an integer that is the position of the decimal point relative to the start of the string. A negative or zero value for **decpt* means to the left of the returned digits.
sign	is a pointer to the sign of the result. If the sign of the result is negative, **sign* is nonzero; otherwise, **sign* is zero.

Usage

The **dececvt( )** function converts the **decimal** value to which *np* points into a null-terminated string of *ndigit* ASCII digits and returns a pointer to the string. A subsequent call to this function overwrites the string.

The **dececvt( )** function rounds low-order digits.

The **decfcvt( )** function is identical to **dececvt( )**, except that *ndigit* specifies the number of digits to the right of the decimal point instead of the total number of digits.

Let *dec_val* point to a **decimal** value of 12345.67 and suppress all arguments except *ndigit*. The following table shows the values that the **dececvt( )** function returns for four different *ndigit* values.

ndigit Value	Return String	*decpt Value	*sign
4	″1235″	5	0
10	″1234567000″	5	0
1	″1″	5	0
3	″123″	5	0

decmul( )

The **decmul( )** function multiplies two **decimal** type values.

Syntax

```
mint decmul(n1, n2, product)
   dec_t *n1;
   dec_t *n2;
   dec_t *product;
```

n1	is a pointer to the **decimal** structure of the first operand.
n2	is a pointer to the **decimal** structure of the second operand.
product	is a pointer to a **decimal** structure to contain the product of *n1* times *n2*.

Usage

The *product* value can be the same as the value of either *n1* or *n2*.

Return Values

0	The operation was successful.
-1200	The operation resulted in overflow.
-1201	The operation resulted in underflow.

decround( )

The **decround( )** function rounds a **decimal** type number to fractional digits.

Syntax

```
void decround(d, s)
   dec_t *d;
   mint s;
```

d	is a pointer to the **decimal** structure that contains the value to round.
s	is the number of fractional digits to round *d*. Use a positive integer or zero (0) for this argument.

Usage

The rounding factor is $5x10^{-s-1}$. To round a value, the **decround( )** function adds the rounding factor to a positive number or subtracts this factor from a negative number. It then truncates to *s* digits, as the following table shows.

Value Before Round	Value of *s*	Rounded Value
1.4	0	1.0
1.5	0	2.0
1.684	2	1.68
1.685	2	1.69
1.685	1	1.7
1.685	0	2.0

decsub( )

The **decsub( )** function subtracts two **decimal** type values.

Syntax

```
mint decsub(n1, n2, difference)
   dec_t *n1;
   dec_t *n2;
   dec_t *difference;
```

n1	is a pointer to the **decimal** structure of the first operand.
n2	is a pointer to the **decimal** structure of the second operand.
difference	is a pointer to a **decimal** structure to contain the difference of *n1* minus *n2*.

Usage

The *difference* value can be the same as the value of either *n1* or *n2*.

Return Values

0	The operation was successful.
-1200	The operation resulted in overflow.
-1201	The operation resulted in underflow.

dectoasc( )

The **dectoasc( )** function converts a **decimal** type number to a C **char** type value.

Syntax

```
mint dectoasc(dec_val, strng_val, len, right)
   dec_t *dec_val;
   char *strng_val;
   mint len;
   mint right;
```

dec_val	is a pointer to the **decimal** structure that contains the value to convert to a text string.
strng_val	is a pointer to the first byte of the character buffer where the dectoasc**( )** function is to place the text string.
len	is the size of *strng_val*, in bytes, minus 1 for the null terminator.
right	is an integer that indicates the number of decimal places to the right of the decimal point.

Usage

If *right* = -1, the decimal value of *dec_val* determines the number of decimal places.

If the **decimal** number does not fit into a character string of length *len*, **dectoasc( )** converts the number to an exponential notation. If the number still does not fit, dectoasc**( )** fills the string with asterisks. If the number is shorter than the string, dectoasc**( )** left-justifies the number and pads it on the right with blanks.

Because the character string that **dectoasc( )** returns is not null terminated, your program must add a null character to the string before you print it.

Return Values

0	The conversion was successful.
-1	The conversion failed.

dectodbl( )

The **dectodbl( )** function converts a **decimal** type number into a C **double** type number.

Syntax

```
mint dectodbl(dec_val, dbl_val)
   dec_t *dec_val;
   double *dbl_val;
```

dec_val	is a pointer to the **decimal** structure that contains the value to convert to a **double** value.
dbl_val	is a pointer to a **double** variable to contain the result of the conversion.

Usage

The floating-point format of the host computer can result in loss of precision in the conversion of a **decimal** type number to a **double** type number.

Return Values

0	The conversion was successful.
<0	The conversion failed.

dectoint( )

The **dectoint( )** function converts a **decimal** type number into a C **int** type number.

Syntax

```
mint dectoint(dec_val, int_val)
   dec_t *dec_val;
   mint *int_val;
```

dec_val	is a pointer to the **decimal** structure that contains the value to convert to an **mint** type value.
int_val	is a pointer to an **mint** variable to contain the result of the conversion.

Usage

The **dectoint( )** library function converts a **decimal** value to a C integer. The size of a C integer depends on the hardware and operating system of the computer you are using; therefore, the **dectoint( )** function equates an integer value with the SQL SMALLINT data type. The valid range of a SMALLINT is between `32767` and `-32767`. To convert larger **decimal** values to larger integers, use the **dectolong( )** library function.

Return Values

0	The conversion was successful.
<0	The conversion failed.
-1200	The magnitude of the **decimal** type number is greater than 32767.

dectolong( )

The **dectolong( )** function converts a **decimal** type number into an **int4** type number.

Syntax

```
mint dectolong(dec_val, lng_val)
   dec_t *dec_val;
   int4  *lng_val;
```

dec_val	is a pointer to the **decimal** structure that contains the value to convert to an **itn4** integer.
lng_val	is a pointer to an **int4** variable to contain the result of the conversion.

Return Values

0	The conversion was successful.
-1200	The magnitude of the **decimal** type number is greater than 2,147,483,647.

dectrunc( )

The **dectrunc( )** function truncates a rounded decimal type number to fractional digits**.**

Syntax

```
void dectrunc(d, s)
   dec_t *d;
   mint s;
```

d	is a pointer to the **decimal** structure that contains the rounded number to truncate.
s	is the number of fractional digits to which to truncate the rounded number. Use a positive integer or zero (0) for this argument.

Usage

The following table shows the sample output from **dectrunc( )** with various inputs.

Value Before Truncation	Value of *s*	Truncated Value
1.4	0	1.0
1.5	0	1.0
1.684	2	1.68
1.685	2	1.68
1.685	1	1.6
1.685	0	1.0

dtaddinv( )

The **dtaddinv( )** function adds an **interval** value to a **datetime** value. The result is a **datetime** value.

Syntax

```
mint dtaddinv(dt, inv, res)
   dtime_t *dt;
   intrvl_t *inv;
   dtime_t *res;
```

dt	is a pointer to an initialized **datetime** variable.
inv	is a pointer to an initialized **interval** variable.
res	is a pointer to a **datetime** variable to contain the result.

Usage

The **dtaddinv( )** function adds the **interval** value in *inv* to the **datetime** value in *dt* and stores the **datetime** value in *res*. This result inherits the qualifier of *dt*.

The **interval** value must be in either the **year to month** or **day to fraction(5)** ranges.

The **datetime** value must include all the fields present in the **interval** value.

If you do not initialize the variables *dt* and *inv*, the function might return an unpredictable result.

Return Values

0	The addition was successful.
<0	Error in addition.

dtcurrent( )

The **dtcurrent( )** function assigns the current date and time to a **datetime** variable.

Syntax

```
void dtcurrent(d)
   dtime_t *d;
```

d	is a pointer to a **datetime** variable to initialize.

Usage

When the variable qualifier is set to zero (`0`) (or any invalid qualifier), the **dtcurrent( )** function initializes it with the `year to fraction(3)` qualifier.

When the variable contains a valid qualifier, the **dtcurrent( )** function extends the current date and time to agree with the qualifier.

Example Calls

The following statements set the variable **now** to the current time, to the nearest millisecond:

```
now.dt_qual = TU_DTENCODE(TU_HOUR,TU_F3);
   dtcurrent(&now);
```

dtcvasc( )

The **dtcvasc( )** function converts a string that conforms to ANSI SQL standard for a DATETIME value to a **datetime** value.

Syntax

```
mint dtcvasc(inbuf, dtvalue)
   char *inbuf;
   dtime_t *dtvalue;
```

inbuf	is a pointer to a buffer to contain an ANSI-standard DATETIME string.
dtvalue	is a pointer to an initialized **datetime** variable.

Usage

You must initialize the **datetime** variable in *dtvalue* with the qualifier that you want this variable to have.

The character string in *inbuf* must have values that conform to the **year to second** qualifier in the ANSI SQL format. The *inbuf* string can have leading and trailing spaces. However, from the first significant digit to the last, *inbuf* can only contain characters that are digits and delimiters that conform to the ANSI SQL standard for DATETIME values.

If you specify a year value as one or two digits, the **dtcvasc( )** function assumes that the year is in the present century. You can set the **DBCENTURY** environment variable to determine which century **dtcvasc( )** uses when you omit a century from the date.

If the character string is an empty string, the **dtcvasc( )** function sets to null the value to which *dtvalue* points. If the character string is acceptable, the function sets the value in the **datetime** variable and returns zero. Otherwise, the function leaves the variable unchanged and returns a negative error code.

Return Values

0	Conversion was successful.
-1260	It is not possible to convert between the specified types.
-1261	Too many digits in the first field of **datetime** or **interval**.
-1262	Non-numeric character in **datetime** or **interval**.
-1263	A field in a **datetime** or **interval** value is out of range or incorrect.
-1264	Extra characters exist at the end of a **datetime** or **interval**.
-1265	Overflow occurred on a **datetime** or **interval** operation.
-1266	A **datetime** or **interval** value is incompatible with the operation.
-1267	The result of a **datetime** computation is out of range.
-1268	A parameter contains an invalid **datetime** qualifier.

dtcvfmtasc( )

The **dtcvfmtasc( )** function uses a formatting mask to convert a character string to a **datetime** value.

Syntax

```
mint dtcvfmtasc(inbuf, fmtstring, dtvalue)
   char *inbuf;
   char *fmtstring;
   dtime_t *dtvalue;
```

inbuf	is a pointer to the buffer that contains the string to convert.
fmtstring	is a pointer to a buffer that contains the formatting mask to use for the *inbuf* string. 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*).
dtvalue	is a pointer to an initialized **datetime** variable.

Usage

You must initialize the **datetime** variable in *dtvalue* with the qualifier that you want this variable to have. The **datetime** variable does not need to specify the same qualifier that the formatting mask implies. When the **datetime** qualifier is different from the implied formatting-mask qualifier, **dtcvfmtasc( )** extends the **datetime** value (as if it had called the **dtextend( )** function).

All qualifier fields in the character string in *inbuf* must be contiguous. In other words, if the qualifier is **hour to second**, you must specify all values for **hour**, **minute**, and **second** somewhere in the string, or the **dtcvfmtasc( )** function returns an error.

The *inbuf* character string can have leading and trailing spaces. However, from the first significant digit to the last, *inbuf* can contain only digits and delimiters that are appropriate for the qualifier fields that the formatting mask implies.

The **dtcvfmtasc( )** function returns an error if the formatting mask, *fmtstring*, is an empty string. If *fmtstring* is a null pointer, the **dtcvfmtasc( )** function must determine the format to use when it reads the character string in *inbuf*. When you use the default locale, the function uses the following precedence:

1. The format that the **DBTIME** environment variable specifies (if **DBTIME** is set)

   For more information, see the *IBM Informix Guide to SQL: Reference*.
2. The format that the **GL_DATETIME** environment variable specifies (if **GL_DATETIME** is set)

   For more information, see the *IBM Informix GLS User's Guide*.
3. The default date format that conforms to the standard ANSI SQL format:

   ```
   %iY-%m-%d %H:%M:%S
   ```

The ANSI SQL format specifies a qualifier of **year to second** for the output. You can express the year as four digits (2007) or as two digits (07). When you use a two-digit year (**%y**) in a formatting mask, the **dtcvfmtasc( )** function uses the value of the **DBCENTURY** environment variable to determine which century to use. If you do not set **DBCENTURY**, **dtcvfmtasc( )** assumes the present century for two-digit years. For information on how to set **DBCENTURY**, see the *IBM Informix*

Guide to SQL: Reference.

Global Language Support

When you use a nondefault locale (one other than U.S. English) and do not set the **DBTIME** or **GL_DATETIME** environment variables, **dtcvfmtasc( )** uses the default date and time format that the locale defines. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

When the character string and the formatting mask are acceptable, the **dtcvfmtasc( )** function sets the **datetime** variable in *dtvalue* and returns zero. Otherwise, it returns an error code and the **datetime** variable contains an unpredictable value.

Return Values

0	The conversion was successful.
<0	The conversion failed.

dtextend( )

The **dtextend( )** function extends a **datetime** value to a different qualifier. Extending is the operation of adding or dropping fields of a DATETIME value to make it match a given qualifier.

Syntax

```
mint dtextend(in_dt, out_dt)
   dtime_t *in_dt, *out_dt;
```

in_dt	is a pointer to the **datetime** variable to extend.
out_dt	is a pointer to a **datetime** variable with a valid qualifier to use for the extension.

Usage

The **dtextend( )** function copies the qualifier-field digits of the *in_dt* **datetime** variable to the *out_dt* **datetime** variable. The qualifier of the *out_dt* variable controls the copy.

The function discards any fields in *in_dt* that the *out_dt* variable does not include. The function fills in any fields in *out_dt* that are not present in *in_dt*, as follows:

- It fills in fields to the left of the most-significant field in *in_dt* from the current time and date.
- It fills in fields to the right of the least-significant field in *in_dt* with zeros.

In the following example, a variable **fiscal_start** is set up with the first day of a fiscal year that begins on June 1. The **dtextend( )** function generates the current year.

```
datetime work, fiscal_start;

work.dt_qual = TU_DTENCODE(TU_MONTH,TU_DAY);
dtcvasc("06-01",&work);
fiscal_start.dt_qual = TU_DTENCODE(TU_YEAR,TU_DAY);
dtextend(&work,&fiscal_start);
```

Return Values

0	The operation was successful.
-1268	A parameter contains an invalid **datetime** qualifier.

dtsub( )

The **dtsub( )** function subtracts one **datetime** value from another. The result is an **interval** value.

Syntax

```
mint dtsub(d1, d2, inv)
   dtime_t *d1, *d2;
   intrvl_t *inv;
```

d1	is a pointer to an initialized **datetime** variable.
d2	is a pointer to an initialized **datetime** variable.
inv	is a pointer to an **interval** variable to contain the result.

Usage

The **dtsub( )** function subtracts the **datetime** value *d2* from *d1* and stores the **interval** result in *inv*. The result can be either a positive or a negative value. If necessary, the function extends *d2* to match the qualifier for *d1*, before the subtraction.

Initialize the qualifier for *inv* with a value in either the **year to month** or **day to fraction(5)** classes. When *d1* contains fields in the **day to fraction** class, the **interval** qualifier must also be in the **day to fraction** class.

Return Values

0	The subtraction was successful.
<0	An error occurred while performing the subtraction.

dtsubinv( )

The **dtsubinv( )** function subtracts an **interval** value from a **datetime** value. The result is a **datetime** value.

Syntax

```
mint dtsubinv(dt, inv, res)
   dtime_t *dt;
   intrvl_t *inv;
   dtime_t *res;
```

dt	is a pointer to an initialized **datetime** variable.
inv	is a pointer to an initialized **interval** variable.
res	is a pointer to a **datetime** variable to contain the result.

Usage

The **dtsubinv( )** function subtracts the **interval** value in *inv* from the **datetime** value in *dt* and stores the **datetime** value in *res*. This result inherits the qualifier of *dt*.

The **datetime** value must include all the fields present in the **interval** value. When you do not initialize the variables *dt* and *inv,* the function might return an unpredictable result.

Return Values

0	The subtraction was successful.
<0	An error occurred while performing the subtraction.

dttoasc( )

The **dttoasc( )** function converts the field values of a **datetime** variable to an ASCII string that conforms to ANSI SQL standards.

Syntax

```
mint dttoasc(dtvalue, outbuf)
   dtime_t *dtvalue;
   char *outbuf;
```

dtvalue	is a pointer to an initialized **datetime** variable.
outbuf	is a pointer to a buffer to receive the ANSI-standard DATETIME string for the value in *dtvalue*.

Usage

The **dttoasc( )** function converts the digits of the fields in the **datetime** variable to their character equivalents and copies them to the *outbuf* character string with delimiters (hyphen, space, colon, or period) between them. You must initialize the **datetime** variable in *dtvalue* with the qualifier that you want the character string to have.

The character string does *not* include the qualifier or the parentheses that SQL statements use to delimit a DATETIME literal. The *outbuf* string conforms to ANSI SQL standards. It includes one character for each delimiter, plus the fields, which are of the following sizes.

Field	**Field Size**
Year	Four digits
Fraction of DATETIME	As specified by precision
All other fields	Two digits

A **datetime** value with the **year to fraction(5)** qualifier produces the maximum length of output. The string equivalent contains 19 digits, 6 delimiters, and the null terminator, for a total of 26 bytes:

```
YYYY-MM-DD HH:MM:SS.FFFFF
```

If you do not initialize the qualifier of the **datetime** variable, the **dttoasc( )** function returns an unpredictable value, but this value does not exceed 26 bytes.

Return Values

0	The conversion was successful.
<0	The conversion failed.

dttofmtasc( )

The **dttofmtasc( )** function uses a formatting mask to convert a **datetime** variable to a character string.

Syntax

```
mint dttofmtasc(dtvalue, outbuf, buflen, fmtstring)
   dtime_t *dtvalue;
   char *outbuf;
   mint buflen;
   char *fmtstring;
```

dtvalue	is a pointer to an initialized **datetime** variable.
outbuf	is a pointer to a buffer to contain the string for the value in *dtvalue*.
buflen	is the length of the *outbuf* buffer.
fmtstring	is a pointer to a buffer that contains the formatting mask to use for the *outbuf* string. 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*.)

Usage

You must initialize the **datetime** variable in *dtvalue* with the qualifier that you want the character string to have. If you do not initialize the **datetime** variable, the function returns an unpredictable value. The character string in *outbuf* does *not* include the qualifier or the parentheses that SQL statements use to delimit a DATETIME literal.

The formatting mask, *fmtstring*, does not need to imply the same qualifiers as the **datetime** variable. When the implied formatting-mask qualifier is different from the **datetime** qualifier, **dttofmtasc( )** extends the **datetime** value (as if it called the **dtextend( )** function).

If the formatting mask is an empty string, the function sets character string in *outbuf* to an empty string. If *fmtstring* is a null pointer, the **dttofmtasc( )** function must determine the format to use for the character string in *outbuf.* When you use the default locale, the function uses the following precedence:

1. The format that the **DBTIME** environment variable specifies (if **DBTIME** is set)

   For more information, see the *IBM Informix Guide to SQL: Reference*.
2. The format that the **GL_DATETIME** environment variable specifies (if **GL_DATETIME** is set)

   For more information, see the *IBM Informix GLS User's Guide*.
3. The default date format that conforms to the standard ANSI SQL format:

   ```
   %iY-%m-%d %H:%M:%S
   ```

When you use a two-digit year (**%y**) in a formatting mask, the **dttofmtasc( )** function uses the value of the **DBCENTURY** environment variable to determine which century to use. If you do not set **DBCENTURY**, **dttofmtasc( )** uses the present century for two-digit years. For information on how to set **DBCENTURY**, see the *IBM Informix Guide to SQL: Reference*.

Global Language Support

When you use a nondefault locale (one other than U.S. English) and do not set the **DBTIME** or **GL_DATETIME** environment variables, **dttofmtasc( )** uses the default DATETIME format that the client locale defines. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Return Values

0	The conversion was successful.
<0	The conversion failed. Check the text of the error message.

ifx_int8add( )

The **ifx_int8add( )** function adds two **int8** type values.

Syntax

```
mint ifx_int8add(n1, n2, sum)
   ifx_int8_t *n1;
   ifx_int8_t *n2;
   ifx_int8_t *sum;
```

n1	is a pointer to the **int8** structure that contains the first operand.
n2	is a pointer to the **int8** structure that contains the second operand.
sum	is a pointer to an **int8** structure to contain the sum of *n1* + *n2*.

Usage

The *sum* value can be the same as the value of either *n1* or *n2*.

Return Values

0	The operation was successful.
-1284	The operation resulted in overflow or underflow.

ifx_int8cmp( )

The **ifx_int8cmp( )** function compares two **int8** type numbers.

Syntax

```
mint ifx_int8cmp(n1, n2)
   ifx_int8_t *n1;
   ifx_int8_t *n2;
```

n1	is a pointer to the **int8** structure that contains the first number to compare.
n2	is a pointer to the **int8** structure that contains the second number to compare.

Return Values

-1	The first value is less than the second value.
0	The two values are identical.
1	The first value is greater than the second value.
INT8UNKNOWN	Either value is null.

ifx_int8copy( )

The **ifx_int8copy( )** function copies one **int8** structure to another.

Syntax

```
void ifx_int8copy(source, target)
   ifx_int8_t *source;
   ifx_int8_t *target;
```

source	is a pointer to the **int8** structure that contains the source **int8** value to copy.
target	is a pointer to the target **int8** structure.

The **ifx_int8copy( )** function does not return a status value. To determine the success of the copy operation, look at the contents of the **int8** structure to which the *target* argument points.

ifx_int8cvasc( )

The **ifx_int8cvasc( )** function converts a value held as printable characters in a C **char** type into an **int8** type number.

Syntax

```
mint ifx_int8cvasc(strng_val, len, int8_val)
   char *strng_val
   mint len;
   ifx_int8_t *int8_val;
```

strng_val	is a pointer to a string.
len	is the length of the *strng_val* string.
int8_val	is a pointer to an **int8** structure to contain the result of the conversion.

Usage

The character string, *strng_val*, can contain the following symbols:

- A leading sign, either a plus (+) or minus (-)
- An exponent that is preceded by either `e` or `E`

  You can precede the exponent by a sign, either plus (+) or minus (-).

The *strng_val* character string should *not* contain a decimal separator or digits to the right of the decimal separator. The **ifx_int8svasc( )** function truncates the decimal separator and any digits to the right of the decimal separator. The **ifx_int8cvasc( )** function ignores leading spaces in the character string.

Global Language Support

When you use a nondefault locale (one other than U.S. English), **ifx_int8cvasc( )** supports non-ASCII characters in the *strng_val* character string. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Return Values

0	The conversion was successful.
-1213	The string has non-numeric characters.
-1284	The operation resulted in overflow or underflow.

ifx_int8cvdbl( )

The **ifx_int8cvdbl( )** function converts a C **double** type number into an **int8** type number.

Syntax

```
mint ifx_int8cvdbl(dbl_val, int8_val)
   double dbl_val;
   ifx_int8_t *int8_val;
```

dbl_val	is the **double** value to convert to an **int8** value.
int8_val	is a pointer to an **int8** structure to contain the result of the conversion.

Return Values

0	The conversion was successful.
<0	The conversion failed.

ifx_int8cvdec( )

The **ifx_int8cvdec( )** function converts a **decimal** type value into an **int8** type value.

Syntax

```
mint ifx_int8cvdec(dec_val, int8_val)
   dec_t *dec_val;
   ifx_int8_t *int8_val;
```

dec_val	is a pointer to the **decimal** structure that contains the value to convert to an **int8** value.
int8_val	is a pointer to an **int8** structure to contain the result of the conversion.

Return Values

0	The conversion was successful.
<0	The conversion failed.

ifx_int8cvflt( )

The **ifx_int8cvflt( )** function converts a C **float** type number into an **int8** type number.

Syntax

```
mint ifx_int8cvflt(flt_val, int8_val)
   double flt_val;
   ifx_int8_t *int8_val;
```

flt_val	is the **float** value to convert to an **int8** value.
int8_val	is a pointer to an **int8** structure to contain the result of the conversion.

Return Values

0	The conversion was successful.
<0	The conversion failed.

ifx_int8cvint( )

The **ifx_int8cvint( )** function converts a C **int** type number into an **int8** type number.

Syntax

```
mint ifx_int8cvint(int_val, int8_val)
   mint int_val;
   ifx_int8_t *int8_val;
```

int_val	is the **mint** value to convert to an **int8** value.
int8_val	is a pointer to an **int8** structure to contain the result of the conversion.

Return Values

0	The conversion was successful.
<0	The conversion failed.

ifx_int8cvlong( )

The **ifx_int8cvlong( )** function converts a C **long** type value into an **int8** type value.

Syntax

```
mint ifx_int8cvlong(lng_val, int8_val)
   int4 lng_val;
   ifx_int8_t *int8_val;
```

lng_val	is the **int4** integer to convert to an **int8** value.
int8_val	is a pointer to an **int8** structure to contain the result of the conversion.

Return Values

0	The conversion was successful.
<0	The conversion failed.

ifx_int8div( )

The **ifx_int8div( )** function divides two **int8** type values.

Syntax

```
mint ifx_int8div(n1, n2, quotient)
   ifx_int8_t *n1;
   ifx_int8_t *n2;
   ifx_int8_t *quotient;
```

n1	is a pointer to the **int8** structure that contains the dividend.
n2	is a pointer to the **int8** structure that contains the divisor.
quotient	is a pointer to an **int8** structure to contain the quotient of *n1*/*n2*.

Usage

The *quotient* value can be the same as the value of either *n1* or *n2*.

Return Values

0	The operation was successful.
-1202	The operation attempted to divide by zero (0).

ifx_int8mul( )

The **ifx_int8mul( )** function multiplies two **int8** type values.

Syntax

```
mint ifx_int8mul(n1, n2, product)
   ifx_int8_t *n1;
   ifx_int8_t *n2;
   ifx_int8_t *product;
```

n1	is a pointer to the **int8** structure that contains the first operand.
n2	is a pointer to the **int8** structure that contains the second operand.
product	is a pointer to an **int8** structure to contain the product of *n1* * *n2*.

Usage

The *product* value can be the same as the value of either *n1* or *n2*.

Return Values

0	The operation was successful.
-1284	The operation resulted in overflow or underflow.

ifx_int8sub( )

The **ifx_int8sub( )** function subtracts two **int8** type values.

Syntax

```
mint ifx_int8sub(n1, n2, difference)
   ifx_int8_t *n1;
   ifx_int8_t *n2;
   ifx_int8_t *difference;
```

n1	is a pointer to the **int8** structure that contains the first operand.
n2	is a pointer to the **int8** structure that contains the second operand.
difference	is a pointer to an **int8** structure to contain the difference of *n1* and *n2* (*n1* - *n2*).

Usage

The *difference* value can be the same as the value of either *n1* or *n2*.

Return Values

0	The subtraction was successful.
-1284	The subtraction resulted in overflow or underflow.

ifx_int8toasc( )

The **ifx_int8toasc( )** function converts an **int8** type number to a C **char** type value.

Syntax

```
mint inf_int8toasc(int8_val, strng_val, len)
   ifx_int8_t *int8_val;
   char *strng_val;
   int len;
```

int8_val	is a pointer to the **int8** structure that contains the value to convert to a text string.
strng_val	is a pointer to the first byte of the character buffer to contain the text string.
len	is the size of *strng_val*, in bytes, minus 1 for the null terminator.

Usage

If the **int8** number does not fit into a character string of length *len*, **ifx_int8toasc( )** converts the number to an exponential notation. If the number still does not fit, **ifx_int8toasc( )** fills the string with asterisks. If the number is shorter than the string, **ifx_int8toasc( )** left justifies the number and pads it on the right with blanks.

Because the character string that **ifx_int8toasc( )** returns is not null terminated, your program must add a null character to the string before you print it.

Global Language Support

When you use a nondefault locale (one other than U.S. English), **ifx_int8toasc( )** supports non-ASCII characters in the *strng_val* character string. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Return Values

0	The conversion was successful.
-1207	The converted value does not fit into the allocated space.

ifx_int8todbl( )

The **ifx_int8todbl( )** function converts an **int8** type number into a C **double** type number.

Syntax

```
mint ifx_int8todbl(int8_val, dbl_val)
   ifx_int8_t *int8_val;
   double *dbl_val;
```

int8_val	is a pointer to the **int8** structure that contains the value to convert to a **double** value.
dbl_val	is a pointer to a **double** variable to contain the result of the conversion.

Usage

The floating-point format of the host computer can result in loss of precision in the conversion of an **int8** type number to a **double type number**.

Return Values

0	The conversion was successful.
<0	The conversion failed.

ifx_int8todec( )

The **ifx_int8todec( )** function converts an **int8** type number into a **decimal** type number.

Syntax

```
mint ifx_int8todec(int8_val, dec_val)
   ifx_int8_t *int8_val;
   dec_t *dec_val;
```

int8_val	is a pointer to an **int8** structure that contains the value to convert to a **decimal** value.
dec_val	is a pointer to a **decimal** structure to contain the result of the conversion.

Return Values

0	The conversion was successful.
<0	The conversion was not successful.

ifx_int8toflt( )

The **ifx_int8toflt( )** function converts an **int8** type number into a C **float** type number.

Syntax

```
mint ifx_int8toflt(int8_val, flt_val)
   ifx_int8_t *int8_val;
   mint *flt_val;
```

int8_val	is a pointer to an **int8** structure that contains the value to convert to a **float** value.
flt_val	is a pointer to a **float** variable to contain the result of the conversion.

Usage

The **ifx_int8toflt( )** library function converts an **int8** value to a C float. The size of a C float depends upon the hardware and operating system of the computer you are using.

Return Values

0	The conversion was successful.
<0	The conversion failed.

ifx_int8toint( )

The **ifx_int8toint( )** function converts an **int8** type number into a C **int** type number.

Syntax

```
mint ifx_int8toint(int8_val, int_val)
   ifx_int8_t *int8_val;
   mint *int_val;
```

int8_val	is a pointer to an **int8** structure that contains the value to convert to an **mint** value.
int_val	is a pointer to an **mint** variable to contain the result of the conversion.

Usage

The **ifx_int8toint( )** library function converts an **int8** value to a C integer. The size of a C integer depends upon the hardware and operating system of the computer you are using. Therefore, the **ifx_int8toint( )** function equates an integer value with the SQL SMALLINT data type. The valid range of a SMALLINT is between 32767 and -32767. To convert larger **int8** values to larger integers, use the **ifx_int8tolong( )** library function.

Return Values

0	The conversion was successful.
<0	The conversion failed.

ifx_int8tolong( )

The **ifx_int8tolong( )** function converts an **int8** type number into a C **long** type number.

Syntax

```
mint ifx_int8tolong(int8_val, lng_val)
   ifx_int8_t *int8_val;
   int4 *lng_val;
```

int8_val	is a pointer to the **int8** structure that contains the value to convert to an **int4** integer value.
lng_val	is a pointer to an **int4** structure to contain the result of the conversion.

Return Values

0	The conversion was successful.
-1200	The magnitude of the **int8** type number is greater than 2,147,483,647.

incvasc( )

The **incvasc( )** function converts a string that conforms to the ANSI SQL standard for an INTERVAL value to an **interval** value.

Syntax

```
mint incvasc(inbuf, invvalue)
   char *inbuf;
   intrvl_t *invvalue;
```

inbuf	is a pointer to the buffer that contains the ANSI-standard INTERVAL string to convert.
invvalue	is a pointer to an initialized **interval** variable.

Usage

You must initialize the **interval** variable in *invvalue* with the qualifier that you want this variable to have.

The character string in *inbuf* can have leading and trailing spaces. However, from the first significant digit to the last, *inbuf* can only contain characters that are digits and delimiters that are appropriate to the qualifier fields of the **interval** variable.

If the character string is an empty string, the **incvasc( )** function sets the value in *invvalue* to null. If the character string is acceptable, the function sets the value in the **interval** variable and returns zero (0). Otherwise, the function sets the value in the **interval** value to null.

Return Values

0	The conversion was successful.
-1260	It is not possible to convert between the specified types.
-1261	Too many digits in the first field of **datetime** or **interval**.
-1262	Non-numeric character in **datetime** or **interval**.
-1263	A field in a **datetime** or **interval** value is out of range or incorrect.
-1264	Extra characters at the end of a **datetime** or **interval** value.
-1265	Overflow occurred on a **datetime** or **interval** operation.
-1266	A **datetime** or **interval** value is incompatible with the operation.
-1267	The result of a **datetime** computation is out of range.
-1268	A parameter contains an invalid **datetime** or **interval** qualifier.

incvfmtasc( )

The **incvfmtasc( )** function uses a formatting mask to convert a character string to an **interval** value.

Syntax

```
mint incvfmtasc(inbuf, fmtstring, invvalue)
   char *inbuf;
   char *fmtstring;
   intrvl_t *invvalue;
```

inbuf	is a pointer to the buffer that contains the string to convert.
fmtstring	is a pointer to a buffer that contains the formatting mask to use for the *inbuf* string. 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*).
invvalue	is a pointer to an initialized **interval** variable.

Usage

You must initialize the **interval** variable in *invvalue* with the qualifier you want this variable to have. The **interval** variable does not need to specify the same qualifier as the formatting mask. When the **interval** qualifier is different from the implied formatting-mask qualifier, **incvfmtasc( )** converts the result to appropriate units as necessary. However, both qualifiers must belong to the same interval class: either the **year to month** class or the **day to fraction** class.

All fields in the character string in *inbuf* must be contiguous. In other words, if the qualifier is **hour to second**, you must specify all values for **hour**, **minute**, and **second** somewhere in the string, or **incvfmtasc( )** returns an error.

The *inbuf* character string can have leading and trailing spaces. However, from the first significant digit to the last, *inbuf* can contain only digits and delimiters that are appropriate for the qualifier fields that the formatting mask implies.

If the character string is acceptable, the **incvfmtasc( )** function sets the **interval** value in *invvalue* and returns zero. Otherwise, the function returns an error code and the **interval** variable contains an unpredictable value.

The formatting directives **%B**, **%b**, and **%p**, which the **DBTIME** environment variable accepts, are not applicable in *fmtstring* because *month name* and *A.M.* and *P.M.* information is not relevant for intervals of time. Use the **%Y** directive if the **interval** is more than 99 years (**%y** can handle only two digits). For hours, use **%H** (not **%I**, because **%I** can represent only 12 hours). If *fmtstring* is an empty string, the function returns an error.

Return Values

0	The conversion was successful.
<0	The conversion failed.

intoasc( )

The **intoasc( )** function converts the field values of an **interval** variable to an ASCII string that conforms to the ANSI SQL standard.

Syntax

```
mint intoasc(invvalue, outbuf)
   intrvl_t *invvalue;
   char *outbuf;
```

invvalue	is a pointer to the initialized **interval** variable to convert.
outbuf	is a pointer to a buffer to contain the ANSI-standard INTERVAL string for the value in *invvalue*.

Usage

The **intoasc( )** function converts the digits of the fields in the **interval** variable to their character equivalents and copies them to the *outbuf* character string with delimiters (hyphen, space, colon, or period) between them. You must initialize the **interval** variable in *invvalue* with the qualifier that you want the character string to have.

The character string does *not* include the qualifier or the parentheses that SQL statements use to delimit an INTERVAL literal. The *outbuf* string conforms to ANSI SQL standards. It includes one character for each delimiter (hyphen, space, colon, or period) plus fields with the following sizes.

Field	**Field Size**
Leading field	As specified by precision
Fraction	As specified by precision
All other fields	Two digits

An **interval** value with the **day(5) to fraction(5)** qualifier produces the maximum length of output. The string equivalent contains 16 digits, 4 delimiters, and the null terminator, for a total of 21 bytes:

```
DDDDD HH:MM:SS.FFFFF
```

If you do not initialize the qualifier of the **interval** variable, the **intoasc( )** function returns an unpredictable value, but this value does not exceed 21 bytes.

Return Values

0	The conversion was successful.
<0	The conversion failed.

intofmtasc( )

The **intofmtasc( )** function uses a formatting mask to convert an **interval** variable to a character string.

Syntax

```
mint intofmtasc(invvalue, outbuf, buflen, fmtstring)
   intrvl_t *invvalue;
   char *outbuf;
   mint buflen;
   char *fmtstring;
```

invvalue	is a pointer to the initialized **interval** variable to convert.
outbuf	is a pointer to a buffer to contain the string for the value in *invvalue*.
buflen	is the length of the *outbuf* buffer.
fmtstring	is a pointer to a buffer that contains the formatting mask to use for the *outbuf* string. 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*).

Usage

You must initialize the **interval** variable in *invvalue* with the qualifier that you want the character string to have. If you do not initialize the **interval** variable, the function returns an unpredictable value. The character string in *outbuf* does *not* include the qualifier or the parentheses that SQL statements use to delimit an INTERVAL literal.

The formatting mask, *fmtstring*, does not need to imply the same qualifiers as the **interval** variable. When the implied formatting-mask qualifier is different from the **interval** qualifier, **intofmtasc( )** converts the result to appropriate units, as necessary (as if it called the **invextend( )** function). However, both qualifiers must belong to the same class: either the **year to month** class or the **day to fraction** class.

If *fmtstring* is an empty string, the **intofmtasc( )** function sets *outbuf* to an empty string.

The formatting directives **%B**, **%b**, and **%p**, which the **DBTIME** environment variable accepts, are not applicable in *fmtstring* because *month name* and *A.M.* and *P.M.* information is not relevant for intervals of time. Use the **%Y** directive if the **interval** is more than 99 years (**%y** can handle only two digits). For hours, use **%H** (not **%I**, because**%I** can represent only 12 hours). If *fmtstring* is an empty string, the function returns an error.

If the character string and the formatting mask are acceptable, the **incvfmtasc( )** function sets the **interval** value in *invvalue* and returns zero. Otherwise, the function returns an error code and the **interval** variable contains an unpredictable value.

Return Values

0	The conversion was successful.

<0	The conversion failed.

invdivdbl( )

The **invdivdbl( )** function divides an **interval** value by a numeric value.

Syntax

```
mint invdivdbl(iv, num, ov)
   intrvl_t *iv;
   double num;
   intrvl_t *ov;
```

iv	is a pointer to the **interval** value to be divided.
num	is a numeric divisor value.
ov	is a pointer to an **interval** variable with a valid qualifier, to contain the result of the division.

Usage

The input and output qualifiers must both belong to the same **interval** class: either the **year to month** class or the **day to fraction(5)** class. If the qualifier for *ov* is different from the qualifier for *iv* (within the same class), the **invdivdbl( )** function extends the result (as the **invextend( )** function defines).

The **invdivdbl( )** function divides the **interval** value in *iv* by *num* and stores the result in *ov*.

The value in *num* can be either a positive or a negative value.

Return Values

0	The division was successful.
<0	The division failed.
-1200	A numeric value is too large (in magnitude).
-1201	A numeric value is too small (in magnitude).
-1202	The *num* parameter is zero (0).
-1265	Overflow occurred on an **interval** operation.
-1266	An **interval** value is incompatible with the operation.
-1268	A parameter contains an invalid **interval** qualifier.

invdivinv( )

The **invdivinv( )** function divides an **interval** value by another **interval** value.

Syntax

```
mint invdivinv(i1, i2, num)
   intrvl_t *i1, *i2;
   double *num;
```

i1	is a pointer to the **interval** value that is the dividend.
i2	is a pointer to the **interval** value that is the divisor.
num	is a pointer to a **double** variable to contain the quotient.

Usage

The **invdivinv( )** function divides the **interval** value in *i1* by *i2* and stores the result in *num*. The result can be either positive or negative.

Both the input and output qualifiers must belong to the same **interval** class: either the **year to month** class or the **day to fraction(5)** class. If necessary, the **invdivinv( )** function extends the **interval** value in *i2* to match the qualifier for *i1* before the division.

Return Values

0	The division was successful.
<0	The division failed.
-1200	A numeric value is too large (in magnitude).
-1201	A numeric value is too small (in magnitude).
-1266	An **interval** value is incompatible with the operation.
-1268	A parameter contains an invalid **interval** qualifier.

invextend( )

The **invextend( )** function copies an **interval** value under a different qualifier. Extending is the operation of adding or dropping fields of an INTERVAL value to make it match a given qualifier. For INTERVAL values, both qualifiers must belong to the same **interval** class: either the **year to month** class or the **day to fraction(5)** class.

Syntax

```
mint invextend(in_inv, out_inv)
   intrvl_t *in_inv, *out_inv;
```

in_inv	is a pointer to the **interval** value to extend.
out_inv	is a pointer to an **interval** variable with a valid qualifier to use for the extension.

Usage

The **invextend( )** function copies the qualifier-field digits of *in_inv* **interval** variable to the *out_inv* **interval** variable. The qualifier of the *out_inv* variable controls the copy.

The function discards any fields in *in_inv* that are to the right of the least-significant field in *out_inv*. The function fills in any fields in *out_inv* that are not present in *in_inv* as follows:

- It fills the fields to the right of the least-significant field in *in_inv* with zeros.
- It sets the fields to the left of the most-significant field in *in_inv* to valid **interval** values.

Return Values

0	The conversion was successful.
<0	The conversion failed.
-1266	An **interval** value is incompatible with the operation.
-1268	A parameter contains an invalid **interval** qualifier.

invmuldbl( )

The **invmuldbl( )** function multiplies an **interval** value by a numeric value.

Syntax

```
mint invmuldbl(iv, num, ov)
   intrvl_t *iv;
   double num;
   intrvl_t *ov;
```

iv	is a pointer to the **interval** value to multiply.
num	is the numeric **double** value.
ov	is a pointer to an **interval** variable with a valid qualifier, to contain the result of the multiplication.

Usage

The **invmuldbl( )** function multiplies the **interval** value in *iv* by *num* and stores the result in *ov*. The value in *num* can be either positive or negative.

Both the input and output qualifiers must belong to the same **interval** class: either the **year to month** class or the **day to fraction(5)** class. If the qualifier for *ov* is different from the qualifier for *iv* (but of the same class), the **invmuldbl( )** function extends the result (as the **invextend( )** function defines).

Return Values

0	The multiplication was successful.
<0	The multiplication failed.
-1200	A numeric value is too large (in magnitude).
-1201	A numeric value is too small (in magnitude).
-1266	An **interval** value is incompatible with the operation.
-1268	A parameter contains an invalid **interval** qualifier.

ldchar( )

The **ldchar( )** function copies a fixed-length string into a null-terminated string and removes any trailing blanks.

Syntax

```
void ldchar(from, count, to)
   char *from;
   mint count;
   char *to;
```

from	is a pointer to the fixed-length source string.
count	is the number of bytes in the fixed-length source string.
to	is a pointer to the first byte of a null-terminated destination string. The *to* argument can point to the same location as the *from* argument or to a location that overlaps the *from* argument. If this is the case, ldchar**( )** does not preserve the value to which *from points.*

mi_alloc( )

The **mi_alloc( )** function allocates a block of user memory of a specified size and returns a pointer to that block.

Syntax

```
void *mi_alloc(size)
   mi_integer size;
```

size	is the number of bytes to allocate.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_alloc( )** function allocates *size* number of bytes of user memory for a DataBlade API module. The **mi_alloc( )** function is a constructor function for user memory.

Server Only

From the point of view of a C user-defined routine, this function behaves like the **malloc( )** system call, except that the database server can reclaim memory that **mi_alloc( )** allocates. The **mi_alloc( )** function allocates the memory in the current memory duration. By default, the current memory duration is PER_ROUTINE. With the PER_ROUTINE duration, the database server frees the allocated memory after the C UDR returns.

Except in callback routines, you can change the memory 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( )** and provides the option of specifying the duration.
- Call **mi_switch_mem_duration( )** before you call **mi_alloc( )**.

  The **mi_switch_mem_duration( )** function changes the current memory duration for all subsequent memory allocations.

In UDR routines, the database server automatically frees memory allocated with **mi_alloc( )** when an exception is raised.

Important: In C UDRs, use DataBlade API memory-management functions to allocate memory. Use of a DataBlade API memory-management function guarantees that the database server can automatically free the memory, especially in cases of return values or exceptions, where the routine would not otherwise be able to free the memory.

End of Server Only

Client Only

In client LIBMI applications, **mi_alloc( )** works exactly as **malloc( )** does: it allocates storage on the heap of the client process. The database server does not perform any automatic storage retrieval. The client LIBMI application must use **mi_free( )** to free explicitly all allocations that **mi_alloc( )** makes.

Important: Client LIBMI applications ignore memory duration.

Client LIBMI applications can use either DataBlade API memory-management functions or system memory-management functions (such as **malloc( )**).

End of Client Only

The **mi_alloc( )** function returns a pointer to the newly allocated user memory. Cast this pointer to match the structure of the user-defined buffer or structure that you allocate. A DataBlade API module can use **mi_free( )** to free memory allocated by **mi_alloc( )** when that memory is no longer needed.

Return Values

A **void** pointer	is a pointer to the newly allocated memory. Cast this pointer to match the user-defined buffer or structure for which the memory was allocated.
NULL	indicates that the function was unable to allocate the memory.

The **mi_alloc( )** function does *not* throw an MI_Exception event when it encounters a runtime error. Therefore, it does not cause callbacks to be invoked.

Related Topics

See also the descriptions of **mi_dalloc( ), mi_free( ), mi_realloc( ) , mi_switch_mem_duration( ),** and **mi_zalloc( ).**

For more information, see the discussion on how to allocate user memory in the *IBM Informix DataBlade API Programmer's Guide*.

mi_binary_to_date( )

The **mi_binary_to_date( )** function creates a text (string) representation of a date from the internal (binary) DATE representation.

Syntax

```
mi_lvarchar *mi_binary_to_date(date_data)
   mi_date date_data;
```

date_data	is the internal DATE representation of the date.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_binary_to_date( )** function converts the internal DATE value that *date_data* contains into a date string. It returns a pointer to the buffer that contains the resulting date string.

Global Language Support

The **mi_binary_to_date( )** function formats the date string in the date format of the current processing locale. It also performs any code-set conversion necessary between the current processing locale and the target locale.

End of Global Language Support

Important: The **mi_binary_to_date( )** function is supported only for compatibility with earlier versions of existing DataBlade API modules. The DataBlade API will eventually discontinue support for this function. Use the **mi_date_to_string( )** function in any new DataBlade API modules.

Return Values

An **mi_lvarchar** pointer
is a pointer to the date string that **mi_binary_to_date( )** has created.

NULL indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_date_to_binary( )** and **mi_date_to_string( ).**

mi_binary_to_datetime( )

The **mi_binary_to_datetime( )** function creates a text (string) representation of a date, time, or date and time value from the binary DATETIME representation.

Syntax

```
mi_lvarchar *mi_binary_to_datetime(dt_data)
   mi_datetime *dt_data;
```

dt_data	is a pointer to the internal DATETIME representation of the date, time, or date and time value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_binary_to_datetime( )** function converts the internal DATETIME value that *dt_data* references into a date, time, or date and time string. This function returns a pointer to the buffer that contains the resulting date, time, or date and time string.

Global Language Support

The **mi_binary_to_datetime( )** function formats the date, time, or date and time string in the date, time, or date and time format of the current processing locale. It also performs any code-set conversion necessary between the current processing locale and the target locale.

End of Global Language Support

Important: The **mi_binary_to_datetime( )** function is supported only for compatibility with earlier versions of existing DataBlade API modules. The DataBlade API will eventually discontinue support for this function. Use the **mi_datetime_to_string( )** function in any new DataBlade API modules.

Return Values

An **mi_lvarchar** pointer	is a pointer to the date and time, date, or time string that **mi_binary_to_datetime( )** creates.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_datetime_to_binary( )** and **mi_datetime_to_string( )**.

mi_binary_to_decimal( )

The **mi_binary_to_decimal( )** function creates a text (string) representation of a decimal value from the internal (binary) DECIMAL representation.

Syntax

```
mi_lvarchar *mi_binary_to_decimal(decimal_data)
   mi_decimal *decimal_data;
```

decimal_data is a pointer to the internal DECIMAL representation of the decimal value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_binary_to_decimal( )** function converts the internal DECIMAL value that *decimal_data* contains into a decimal string. It returns a pointer to the buffer that contains the resulting decimal string.

Global Language Support

The **mi_binary_to_decimal( )** function formats the decimal string in the numeric format of the current processing locale. It also performs any code-set conversion necessary between the current processing locale and the target locale.

End of Global Language Support

Important: The **mi_binary_to_decimal( )** function is supported only for compatibility with earlier versions of existing DataBlade API modules. The DataBlade API will eventually discontinue support for this function. Use the **mi_decimal_to_string( )** function in any new DataBlade API modules.

Return Values

An **mi_lvarchar** pointer	is a pointer to the decimal string that **mi_binary_to_decimal( )** creates.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_decimal_to_binary( )** and **mi_decimal_to_string( ).**

mi_binary_to_money( )

The **mi_binary_to_money( )** function creates a text (string) representation of a monetary value from the internal (binary) MONEY representation.

Syntax

```
mi_lvarchar *mi_binary_to_money(money_data)
   mi_money *money_data;
```

money_data	is a pointer to the internal MONEY representation of the monetary value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_binary_to_money( )** function converts the internal MONEY value that *money_data* contains into a monetary string. It returns a pointer to the buffer that contains the resulting monetary string.

Global Language Support

The **mi_binary_to_money( )** function formats the monetary string in the monetary format of the current processing locale. It also performs any code-set conversion necessary between the current processing locale and the target locale.

End of Global Language Support

Important: The **mi_binary_to_money( )** function is supported only for compatibility with earlier versions of existing DataBlade API modules. The DataBlade API will eventually discontinue support for this function. Use the **mi_money_to_string( )** function in any new DataBlade API modules.

Return Values

An **mi_lvarchar** pointer	is a pointer to the monetary string that **mi_binary_to_money( )** creates.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_money_to_binary( )** and **mi_money_to_string( ).**

mi_binary_query( )

The **mi_binary_query( )** function reports whether the last SQL statement sent on *a particular connection* returns results in a binary representation.

Syntax

```
mi_integer mi_binary_query(conn)
   MI_CONNECTION *conn;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Return Values

0	indicates that the query is returning values in text (external) representation.
1	indicates that the query is returning values in binary (internal) representation.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_exec( ), mi_get_result( ), mi_open( ), mi_server_connect( ),** and **mi_server_reconnect( ).**

mi_call( )

The **mi_call( )** function checks if there is sufficient stack space for the specified user-defined routine and extends the stack size if necessary.

Syntax

```
mi_integer mi_call(retval, routine, nargs, argument_list)
   mi_integer *retval;
   mi_integer (*routine)( );
   mi_integer nargs;
   argument_list
```

retval	is a pointer to the location of the user-defined-function return value.
routine	is a pointer to the user-defined routine.
nargs	is the number of the arguments (up to a maximum of 10) in the argument list.
argument_list	is a comma-separated list of from 0 to 10 arguments to pass to the user-defined routine.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_call( )** function allocates stack space for the user-defined routine that *routine* references. The function determines the amount of memory to allocate based on the following information:

- The return value, which *retval* references (for user-defined functions only)
- The number of arguments, *nargs* (for all user-defined routines)

The **mi_call( )** function creates a new stack *only* if the current stack is not large enough to hold the arguments and possible return value of the UDR. If a new stack is required, **mi_call( )** creates it and copies onto it the number of arguments that *nargs* specifies. Pass the actual argument values as a comma-separated list following the *nargs* value.

When **mi_call( )** copies the arguments onto this new stack, the function uses the **MI_DATUM** size as the size of each argument. If a routine argument is larger than an **MI_DATUM** structure, **mi_call( )** does not copy all the argument bytes. You must use the correct passing mechanism for routine arguments to ensure that they fit into the size of an **MI_DATUM** structure.

After arguments are copied to the stack, the **mi_call( )** function calls the specified UDR and executes it. If the UDR is a user-defined function, **mi_call( )** puts its return value at the address that *retval* references.

Return Values

MI_NOMEM	indicates that the **mi_call( )** function was unable to allocate virtual memory.
MI_TOOMANY	indicates that the *nargs* argument is greater than `10`.

MI_CONTINUE
: indicates that the current stack is large enough to hold the invocation of *routine*: no reallocation is required.

MI_DONE
: indicates that the current stack was not large enough to hold the invocation of *routine*: **mi_call( )** allocated a new thread stack, then called *routine* and put any return value in *retval*.

Related Topics

See also the descriptions of **mi_call_on_vp( )** and **mi_process_exec( ).**

For more information, see the discussion of control modes for query data in the *IBM Informix DataBlade API Programmer's Guide*.

mi_call_on_vp( )

The **mi_call_on_vp( )** function enables you to switch execution to a specified virtual processor (VP) to execute a specified C function.

Syntax

```
mi_integer mi_call_on_vp(VP_id, retval, C_func, nargs, argument_list)
   mi_integer VP_id;
   mi_integer *retval;
   mi_integer (*C_func)( );
   mi_integer nargs;
   argument_list;
```

VP_id	is the integer VP-class identifier of the VP on which to execute the specified C function. The VP that executes this C function must be a CPU VP or in a user-defined VP class. You *cannot* execute the C function in some other system VP class.
retval	is a pointer to the integer return value of the specified C function.
C_func	is a pointer to the executable C function to execute on the specified VP.
nargs	is the number of arguments (up to a maximum of 10) to pass to the specified C function.
argument_list	is a comma-separated list of from 0 to 10 arguments to pass to the specified C function.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Warning: This advanced function can adversely affect your UDR if you use the function incorrectly. Use it only when no regular DataBlade API function can perform the task you need done.

Usage

The **mi_call_on_vp( )** function calls the C function that *C_func* references on the VP that VP_*id* identifies. To the specified C function, **mi_call_on_vp( )** passes the number of arguments that *nargs* specifies. In the **mi_call_on_vp( )** function call, pass the actual argument values as a comma-separated list following the *nargs* value. When the C function completes execution, **mi_call_on_vp( )** puts its return value in the address that *retval* references.

The **mi_call_on_vp( )** function takes the following steps:

- Switches the current thread to the VP of the specified VP identifier
- Executes the C function that the C_*func* pointer indicates
- Returns control to the originating VP

After a successful return, the *C_func* function has been executed on the specified VP.

The **mi_call_on_vp( )** function does not switch VPs in the following cases:

- If the originating VP and specified VP (VP_*id*) are identical

  The **mi_call_on_vp( )** function performs no switching but does *not* return an error.

- If the specified VP (VP_*id*) is not a CPU VP or user-defined VP

  The **mi_call_on_vp( )** function performs no switching and returns an error (MI_ERROR). The message log provides more information about why the switch failed.

Return Values

MI_DONE	indicates that the *C_func* function was successfully called and its return value is referenced by *retval*.
MI_TOOMANY	indicates that the *nargs* argument is greater than `10`.
MI_ERROR	indicates that the function was unable to switch VPs. The online log provides more information on why the switch failed.

Related Topics

See also the descriptions of **mi_call( )** and **mi_process_exec( ).**

For more information on how to switch VPs, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_cast_get( )

The **mi_cast_get( )** function looks up a registered cast function by the type identifiers of its source and target data types and creates its function descriptor.

Syntax

```
MI_FUNC_DESC *mi_cast_get(conn, source_type, target_type, cast_status)
   MI_CONNECTION *conn;
   MI_TYPEID *source_type;
   MI_TYPEID *target_type;
   mi_char *cast_status;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).** can be a pointer to a session-duration connection descriptor established by a previous call to **mi_get_session_connection( ).** Use of a session-duration connection descriptor is an *advanced* feature of the DataBlade API.
source_type	is a pointer to the type identifier of the source data type for the cast.
target_type	is a pointer to the type identifier of the target data type for the cast.
cast_status	is a pointer to the status flag that **mi_cast_get( )** sets to indicate the kind of cast function that it locates or the cause of an error.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

The **mi_cast_get( )** function obtains a function descriptor for a cast function whose source and target data types the *source_type* and *target_type* arguments reference. The **mi_cast_get( )** function accepts source and target data types as pointers to type identifiers. This function is one of the functions of the Fastpath interface. It is a constructor function for the function descriptor.

To obtain a function descriptor for a cast function, the **mi_cast_get( )** function performs the following tasks:

1. Looks in the **syscasts** system catalog table for the cast function that casts from the *source_type* data type to the *target_type* data type
2. Allocates a function descriptor for the cast function and saves the routine sequence in this descriptor
3. Allocates an **MI_FPARAM** structure for the cast function and saves the argument and return-value information in this structure
4. Sets the *cast_status* output parameter to provide status information about the cast function
5. Returns a pointer to the function descriptor that it allocates for the cast function

Server Only

When you pass a public connection descriptor (from **mi_open( )**), the **mi_cast_get( )** function allocates the new function descriptor in the PER_COMMAND memory duration. If you pass a session-duration connection descriptor (from **mi_get_session_connection( )**), **mi_cast_get( )** allocates the new function descriptor in the PER_SESSION memory duration. This function descriptor is called a session-duration function descriptor. For more information, see the *IBM Informix DataBlade API Programmer's Guide*.

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. Use them only when a regular connection descriptor or function descriptor cannot perform the task you need done.

End of Server Only

The *cast_status* flag can have one of the following constant values.

Cast-Type Constant	Purpose
MI_ERROR_CAST	The mi_cast_get**( )** function failed.
MI_NO_CAST	A cast does not exist between the two specified types. The user must write a function to perform the cast.
MI_NOP_CAST	A cast is not needed between the two types. The types are equivalent and therefore no cast is required.
MI_SYSTEM_CAST	A built-in cast exists to cast between two data types, usually built-in data types.
MI_EXPLICIT_CAST	A user-defined cast function exists to cast between the two types, and the cast is explicit.
MI_IMPLICIT_CAST	A user-defined cast function exists to cast between the two types, and the cast is implicit.

The following call to **mi_cast_get( )** looks for a cast function that converts from the INTEGER data type to an opaque data type named **percent**:

```
MI_TYPEID *src_type, *trgt_type;
mi_char cast_stat;
MI_FUNC_DESC *func_desc;
MI_CONNECTION *conn;
...
src_type = mi_typestring_to_id(conn, "INTEGER");
trgt_type = mi_typestring_to_id(conn, "percent");

func_desc = mi_cast_get(conn, &src_type, &trgt_type,
   &cast_stat);
if ( func_desc == NULL )
   {
   switch ( cast_stat )
      {
      case MI_NO_CAST:
         mi_db_error_raise(NULL, MI_EXCEPTION,
            "No cast function found.");
         break;
      case MI_ERROR_CAST:
         mi_db_error_raise(NULL, MI_EXCEPTION,
            "Error in mi_cast_get( ) function.");
         break;
      case MI_NOP_CAST:
         mi_db_error_raise(NULL, MI_EXCEPTION,
            "No cast function needed.");
         break;
      ...
```

An **MI_FUNC_DESC** pointer	is a pointer to the function descriptor of the cast function that casts from *source_type* to *target_type*.
NULL	indicates that the *cast_status* value is one of the following constants:

MI_ERROR_CAST	The function was not successful.
MI_NO_CAST	A cast does not exist between the two specified types. The user must write a function to perform the cast.
MI_NOP_CAST	A cast is not needed between the two types. The types are equivalent and therefore no cast is required.

Related Topics

See also the descriptions of **mi_fparam_get( ), mi_get_session_connection( ), mi_open( ), mi_routine_end( ), mi_routine_exec( ), mi_routine_get( ), mi_server_connect( ), mi_server_reconnect( ),** and **mi_td_cast_get( ).**

mi_class_id( )

The **mi_class_id( )** function obtains the VP-class identifier for a specified virtual-processor (VP) class.

Syntax

```
mi_integer mi_class_id(VPclass_name)
   const char *VP_classname;
```

VPclass_name	is a pointer to the name of the VP class whose VP-class identifier the function is to return.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Warning: This advanced function can adversely affect your UDR if you use the function incorrectly. Use it only when no regular DataBlade API function can perform the task you need done.

Usage

The **mi_class_id( )** function returns the VP-class identifier for the VP-class name that VP*class_name* references. The VP-class name is *not* case specific; that is, either uppercase or lowercase letters are valid.

Tip: You can obtain the VP-class identifier for the active VP with the **mi_vpinfo_classid( )** function.

After you have a VP-class identifier, you can use the following DataBlade API functions to obtain additional information about the VP class.

DataBlade API Function	VP-Class Information
mi_class_name( )	VP-class name
mi_class_maxvps( )	Maximum number of VPs in the VP class
mi_class_numvp( )	Number of active VPs in the VP class

Return Values

>=0	is the integer VP-class identifier for the VP class associated with the specified VP-class name.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_class_maxvps( ), mi_class_name( ), mi_class_numvp( ),** and **mi_vpinfo_classid( ).**

For information about how to obtain information on VPs and VP classes, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_class_maxvps( )

The **mi_class_maxvps( )** function obtains the maximum number of virtual processors (VPs) in a VP class.

Syntax

```
mi_integer mi_class_maxvps(VPclass_id, error)
   mi_integer VPclass_id;
   mi_integer *error;
```

VPclass_id is a VP-class identifier for the VP class whose maximum number of VPs the function is to return. This argument can be either of the following values:

A valid VP-class identifier	Obtains the maximum number of VPs in the specified VP class.
MI_CURRENT_CLASS VP-class constant	Obtains the maximum number of VPs in the VP class of the current VP.

error is a pointer to an error value to indicate whether the specified VP-class identifier is valid.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Warning: This advanced function can adversely affect your UDR if you use the function incorrectly. Use it only when no regular DataBlade API function can perform the task you need done.

Usage

The **mi_class_maxvps( )** function returns the maximum number of VPs allowed for the VP class that the VP*class_id* argument specifies. The maximum number of VPs is the value assigned with the **max** option in the VPCLASS configuration parameter. For example, suppose you have the following VPCLASS configuration parameter:

```
VPCLASS=newvp, num=3, max=6
```

The **mi_class_maxvps( )** function returns the value of 6 when it receives the VP-class identifier for the **newvp** VP class. You can obtain a VP-class identifier with the **mi_vpinfo_classid( )** or **mi_class_id( )** function. The following VPCLASS definition does *not* include the **max** option. Therefore, the number of VPs to start is unlimited.

```
VPCLASS=newvp2, num=3
```

To indicate the unlimited maximum number of VPs for the **newvp2** class, **mi_class_maxvps( )** returns MI_ERROR.

Tip: To obtain the number of active VPs in the VP class (initialized by the **num** option of the VPCLASS configuration parameter), use the **mi_class_numvp( )** function.

Return Values

>=0	indicates the maximum number of VPs in the specified VP class. If the specified VP-class identifier is valid, **mi_class_maxvps( )** sets the *error* argument to MI_OK. If this VP-class identifier is not valid, the function sets *error* to MI_ERROR.
-1	indicates that the specified VP class does not have a maximum number of VPs included in its definition. The number of VPs to start is unlimited.

Related Topics

See also the descriptions of **mi_class_id( ), mi_class_name( ), mi_class_numvp( ),** and **mi_vpinfo_classid( ).**

For information about how to obtain information on VPs and VP classes, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_class_name( )

The **mi_class_name( )** function obtains the name of a virtual-processor (VP) class.

Syntax

```
char *mi_class_name(VPclass_id)
   mi_integer VPclass_id;
```

*VP*class_id	is a VP-class identifier for the VP class whose name the function is to return. This argument can be either of the following values: **A valid VP-class identifier** Obtains the name of the specified VP class. **MI_CURRENT_CLASS VP-class constant** Obtains the name of the VP class of the current VP.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Warning: This advanced function can adversely affect your UDR if you use the function incorrectly. Use it only when no regular DataBlade API function can perform the task you need done.

Usage

The **mi_class_name( )** function obtains the name of the VP class that VP*class_id* specifies. The VP-class name is assigned to the VP class in the VPCLASS configuration parameter. For example, suppose you have the following VPCLASS configuration parameter:

```
VPCLASS=newvp, num=3, max=6
```

The **mi_class_name( )** function returns the string 'newvp' when it receives the VP-class identifier for the **newvp** VP class. You can obtain a VP-class identifier with the **mi_vpinfo_classid( )** or **mi_class_id( )** function.

Return Values

A **char** pointer	is a pointer to the name of the VP class associated with the specified VP-class identifier.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_class_id( ), mi_class_maxvps( ), mi_class_numvp( ),** and **mi_vpinfo_classid( ))**.

For information about how to obtain information on VPs and VP classes, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_class_numvp( )

The **mi_class_numvp( )** function obtains the number of active virtual processors (VPs) in a VP class.

Syntax

```
mi_integer mi_class_numvp(VPclass_id)
   mi_integer VPclass_id;
```

VPclass_id	is a VP-class identifier for the VP class whose number of active VPs the function is to return. This argument can be either of the following values:

A valid VP-class identifier	Obtains the number of active VPs in the specified VP class.
MI_CURRENT_CLASS VP-class constant	Obtains the number of active VPs in the VP class of the current VP.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Warning: This advanced function can adversely affect your UDR if you use the function incorrectly. Use it only when no regular DataBlade API function can perform the task you need done.

Usage

The **mi_class_numvp( )** function returns the number of active VPs in the VP class that the VP*class_id* argument specifies. An active VP is a VP that is currently executing a task. The number of active VPs is initially the value assigned with the **num** option in the VPCLASS configuration parameter. For example, suppose you have the following VPCLASS configuration parameter:

```
VPCLASS=newvp, num=3, max=6
```

The **mi_class_numvp( )** function returns the value of 3 when it receives the VP-class identifier for the **newvp** VP class. However, if the DBA dynamically adds or removes VPs, the value that **mi_class_numvp( )** returns might not coincide with the **num** option. You can obtain a VP-class identifier with the **mi_vpinfo_classid( )** or **mi_class_id( )** function.

This function is useful to determine if the current VP is part of a single-instance VP class. For such a VP class, the **mi_class_numvp( )** function returns a value of 1.

Tip: To obtain the maximum number of VPs defined for the VP class (by the **max** option of the VPCLASS configuration parameter), use the **mi_class_maxvps( )** function.

Return Values

>=0	is the number of active VPs in the VP class associated with the specified VP-class identifier.
MI_ERROR	indicates that the function was not successful or that the specified VP class does not exist.

Related Topics

See also the descriptions of **mi_class_id( ), mi_class_maxvps( ), mi_class_name( ),** and **mi_vpinfo_classid( ))**.

For information about how to obtain information on VPs and VP classes, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_client( )

The **mi_client( )** function dynamically determines whether a DataBlade API module is running in the database server or as a client application.

Syntax

```
mi_integer mi_client( )
```

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_client( )** function tells a DataBlade API module which implementation of the DataBlade API it is using, as follows:

- C UDRs use the server-side implementation of the DataBlade API.
- Client LIBMI applications use the client-side implementation of the DataBlade API.

This function is useful in DataBlade API code that runs in both the server-side and client-side implementations of the DataBlade API.

Return Values

0	indicates that the DataBlade API module is running as a C UDR.
1	indicates that the DataBlade API module is running as a client LIBMI application.
MI_ERROR	indicates that the function was not successful.

mi_client_locale( )

The **mi_client_locale( )** function returns the name of the client locale.

Syntax

```
char *mi_client_locale( )
```

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_client_locale( )** function returns the client locale, which indicates the IBM Informix GLS locale that the client application uses. The client-locale name gives information about language, territory, code set, and optionally other information about the client computer. For more information about the client locale, see the *IBM Informix GLS User's Guide*.

The **mi_client_locale( )** function initializes the DataBlade API when it is the first DataBlade API function in a client LIBMI application or user-defined routine.

Return Values

An **mi_char** pointer	is a pointer to the name of the client locale.
NULL	indicates that the function was not successful.

mi_close( )

The **mi_close( )** function closes a connection.

Syntax

```
mi_integer mi_close(conn)
   MI_CONNECTION *conn;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_close( )** function frees the connection descriptor that *conn* references. This function is the destructor function for a connection descriptor. After **mi_close( )** is called, *conn* is no longer a valid open connection and all information in the associated session context becomes invalid. This information includes:

- Save sets created on the connection
- Callbacks registered for the connection
- Statement descriptors for the connection (both implicit and explicit)
- Cursors opened on the connection (with their associated rows)
- Function descriptors

The **mi_close( )** function also deallocates the connection descriptor itself.

Server Only

In a C UDR, the **mi_close( )** function closes a UDR connection, freeing the associated connection descriptor. After a UDR connection closes, any open smart large objects and file descriptors remain valid for the duration of the session. Use the **mi_lo_close( )** function to explicitly close a smart large object. Use the **mi_file_close( )** function to explicitly close an operating-system file.

Important: Do not use the **mi_close( )** function to free a session-duration connection descriptor. A session-duration connection descriptor is valid until the end of the session.

End of Server Only

Client Only

In a client LIBMI application, the **mi_close( )** function closes a client connection, freeing the associated connection descriptor. Closing a client connection ends the session.

End of Client Only

Return Values

MI_OK	indicates that the function was successful.

MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_file_close( ), mi_lo_close( ), mi_open( ), mi_server_connect( ),** and **mi_server_reconnect( ).**

mi_close_statement( )

The **mi_close_statement( )** function closes an open cursor.

Syntax

```
mi_integer mi_close_statement(stmt_desc)
   MI_STATEMENT *stmt_desc;
```

stmt_desc	is a pointer to a statement descriptor that identifies a prepared statement whose cursor has been opened with **mi_open_prepared_statement( ).**

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_close_statement( )** function closes the cursor associated with the prepared statement that *stmt_desc* references. This cursor is an explicit cursor, which the **mi_open_prepared_statement( )** function has opened. To close a cursor, **mi_close_statement( )** clears out all rows from the cursor. A cursor must be closed before it can be opened again. After you close a cursor, you can reopen it with another call to **mi_open_prepared_statement( ).** After the cursor is reopened, it is empty.

Important: When you close a cursor, the cursor remains allocated. When you free a prepared statement with **mi_drop_prepared_statement( ),** the prepared statement *and* the associated cursor are deallocated. You must reprepare the statement with **mi_prepare( )** to reopen the cursor. It is recommended that you explicitly free cursors with **mi_drop_prepared_statement( )** once you no longer need them; otherwise, prepared statements and their cursors remain until the associated session ends.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_drop_prepared_statement( ), mi_open_prepared_statement( ),** and **mi_prepare( ).**

mi_collection_card( )

The **mi_collection_card( )** function returns a count of the elements in a collection (that is, its cardinality).

Syntax

```
mi_integer mi_collection_card(coll_ptr, isnull)
   MI_COLLECTION *coll_ptr;
   mi_boolean *isnull;
```

coll_ptr	is a valid pointer to a collection structure.
isnull	is a valid pointer to an SQL NULL indicator flag, which can have either of the following values:

MI_TRUE
The collection value is SQL NULL.

MI_FALSE
The collection value is not SQL NULL. The collection contains zero or more elements.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_collection_card( )** function returns the cardinality of the collection that *coll_ptr* references. The collection value can be SQL NULL, in which case the *isnull* flag will be set to MI_TRUE and the return value will be `0`. A collection does not have to be open for **mi_collection_card( )** to obtain its cardinality.

Return Values

0	(*isnull* = MI_TRUE) indicates the collection value is SQL NULL.
>=0	(*isnull* = MI_FALSE) provides the cardinality of the collection.
MI_ERROR	indicates an invalid parameter.

mi_collection_close( )

The **mi_collection_close( )** function closes a collection and frees the collection descriptor.

Syntax

```
mi_integer mi_collection_close(conn, coll_desc)
   MI_CONNECTION *conn;
   MI_COLL_DESC *coll_desc;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
coll_desc	is a pointer to the collection descriptor.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

The **mi_collection_close( )** function performs the following tasks:

- It closes and frees the collection cursor that is associated with the collection descriptor that *coll_desc* references.
- It frees the collection descriptor that *coll_desc* references.

This collection was opened by a previous call to the **mi_collection_open( )** or **mi_collection_open_with_options( )** function. This function is the destructor function for the collection descriptor and the associated collection cursor.

Important: When you close a collection cursor with **mi_collection_close( )**, the cursor does not remain allocated. To reuse the cursor, you must re-create it with the **mi_collection_open( )** function. When you free a collection descriptor with **mi_collection_close( )**, the collection structure *remains* allocated.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_collection_copy( ), mi_collection_create( ), mi_collection_delete( ), mi_collection_fetch( ), mi_collection_free( ), mi_collection_insert( ), mi_collection_open( ), mi_collection_open_with_options( ), mi_collection_update( ), mi_open( ), mi_server_connect( ),** and **mi_server_reconnect( ).**

For a description of collections, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_collection_copy( )

The **mi_collection_copy( )** function copies a collection to a new collection variable.

Syntax

```
MI_COLLECTION *mi_collection_copy(conn, coll_ptr)
   MI_CONNECTION *conn;
   MI_COLLECTION *src_coll;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
coll_ptr	is a pointer to the collection structure for the source collection. This argument is the collection to copy.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_collection_copy( )** function copies the collection that *coll_ptr* references to a newly allocated collection structure. This function is a constructor function for the collection structure. It allocates a new collection structure in the current memory duration.

For example, the **mi_collection_copy( )** function creates a copy of the **colval** collection:

```
MI_COLLECTION  *ret_val, *new_retval;
MI_DATUM        colval;

for (i = 0; i < numcols; i++)
   switch( mi_value(row, i, &colval, &collen) )
      {
      case MI_COLLECTION_VALUE:
         ret_val = ( MI_COLLECTION * ) colval;

         if ( (new_retval =
               mi_collection_copy(conn, ret_val))
               == NULL )
            mi_db_error_raise( NULL, MI_FATAL,
               "Fatal Error: Cannot copy collection");
         break;
      } /* end switch */
```

In this example, the **mi_value( )** function returns a pointer to **colval**, which is a collection column. However, the memory that is allocated to **colval** is freed when you close the connection with **mi_close( ).** The copy of **colval** that the call to the **mi_collection_copy( )** function creates is in the current memory duration. This new collection, which **new_retval** references, can now be returned from this function.

Return Values

An **MI_COLLECTION** pointer	is the address of the newly allocated collection.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_close( ), mi_collection_copy( ), mi_collection_create( ), mi_collection_delete( ), mi_collection_fetch( ), mi_collection_free( ), mi_collection_insert( ), mi_collection_open( ), mi_collection_update( ), mi_open( ), mi_server_connect( ), mi_server_reconnect( ),** and **mi_value( ).**

For a description of collections, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_collection_create( )

The **mi_collection_create( )** function creates a collection.

Syntax

```
MI_COLLECTION *mi_collection_create(conn, typeid_ptr)
   MI_CONNECTION *conn;
   MI_TYPEID *typeid_ptr;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
typeid_ptr	is a pointer to the type identifier for the new collection.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_collection_create( )** function creates a new collection structure of the collection type that *typeid_ptr* references. This function is a constructor function for the collection structure. To access items of the collection, you must first open the collection with the **mi_collection_open( )** function.

Server Only

The **mi_collection_create( )** function allocates a new collection structure in the current memory duration.

End of Server Only

Return Values

An **MI_COLLECTION** pointer	is a pointer to the newly created collection.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_collection_card( ), mi_collection_close( ), mi_collection_copy( ), mi_collection_delete( ), mi_collection_fetch( ), mi_collection_free( ), mi_collection_insert( ), mi_collection_open( ), mi_collection_update( ), mi_open( ), mi_server_connect( ), mi_server_reconnect( ), mi_typestring_to_id( ),** and **mi_typename_to_id( ).**

For a description of collections, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_collection_delete( )

The **mi_collection_delete( )** function deletes a single element of a collection.

Syntax

```
mi_integer mi_collection_delete(conn, coll_desc, action, jump)
   MI_CONNECTION *conn;
   MI_COLL_DESC *coll_desc;
   MI_CURSOR_ACTION action;
   mi_integer jump;
```

conn is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**

coll_desc is a pointer to the collection descriptor.

action Determines the orientation of the deletion. When a collection opens, elements are available in a cursor. The current cursor position is before the first element. Possible *action* values follow:

MI_CURSOR_NEXT	Deletes the *next* element after the current cursor position.
MI_CURSOR_PRIOR	Deletes the element *before* the current cursor position.
MI_CURSOR_FIRST	Deletes the *first* element.
MI_CURSOR_LAST	Deletes the *last* element.
MI_CURSOR_ABSOLUTE	Moves *jump* elements from the beginning of the cursor and deletes the element at the new cursor position.
MI_CURSOR_RELATIVE	Moves *jump* elements from the current position and deletes the element at this new cursor position.
MI_CURSOR_CURRENT	Deletes the element at the current cursor position.

jump is the absolute or relative offset of the deletion for list collections only. If *action* is not MI_CURSOR_ABSOLUTE or MI_CURSOR_RELATIVE when *jump* is specified, the function raises an exception and returns MI_NULL_VALUE.

For absolute positioning, a *jump* value of 1 is the first element.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_collection_delete( )** function deletes the element that the *action* argument indicates from the open collection that *coll_desc* references. The function deletes the specified element from the collection cursor that is associated with the *coll_desc* collection descriptor. After the deletion of an element, the cursor position remains on the deleted element.

Return Values

MI_OK	indicates that the function was successful.
MI_NULL_VALUE	indicates that the function call specifies *jump* and *action* is not MI_CURSOR_ABSOLUTE or MI_CURSOR_RELATIVE.
MI_ERROR	indicates that the function was not successful, including an attempt to delete a nonexistent element.

Related Topics

See also the descriptions of **mi_collection_close( ), mi_collection_copy( ), mi_collection_create( ), mi_collection_fetch( ), mi_collection_free( ), mi_collection_insert( ), mi_collection_open( ), mi_collection_update( ), mi_open( ), mi_server_connect( ),** and **mi_server_reconnect( ).**

For a description of collections, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_collection_fetch( )

The **mi_collection_fetch( )** function fetches a single element from a collection.

Syntax

```
mi_integer mi_collection_fetch(conn, coll_desc, action, jump, value_buf,
   value_len)
   MI_CONNECTION *conn;
   MI_COLL_DESC *coll_desc;
   MI_CURSOR_ACTION action;
   mi_integer jump;
   MI_DATUM *value_buf;
   mi_integer *value_len;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
coll_desc	is a pointer to the collection descriptor for the collection from which to fetch.
action	determines the orientation of the fetch. When a collection opens, elements are available in a cursor. The current cursor position is before the first element. Possible *action* values follow:

MI_CURSOR_NEXT	Fetches the next element *after* the current cursor position.
MI_CURSOR_PRIOR	Fetches the element *before* the current cursor position.
MI_CURSOR_FIRST	Fetches the *first* element.
MI_CURSOR_LAST	Fetches the *last* element.
MI_CURSOR_ABSOLUTE	Moves *jump* elements from the beginning of the cursor and fetches the element at the new cursor position.
MI_CURSOR_RELATIVE	Moves *jump* elements from the current position and fetches the element at this new cursor position.
MI_CURSOR_CURRENT	Fetches the element at the current cursor position.

jump	is the absolute or relative offset of the fetch for list collections only. If *action* is not MI_CURSOR_ABSOLUTE or MI_CURSOR_RELATIVE when *jump* is specified, the function raises an exception and returns MI_NULL_VALUE. For absolute positioning, a *jump* value of 1 is the first element.
value_buf	is a pointer to the **MI_DATUM** structure that contains the collection element. The **mi_collection_fetch( )** function allocates the **MI_DATUM** structure. You do *not* need to supply the buffer to hold the returned value.
value_len	is a pointer to the length of the fetched collection element in the *value_buf* argument.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_collection_fetch( )** function fetches the element that *action* specifies from the open collection that *coll_desc* references. This function retrieves the specified element from the collection cursor associated with *coll_desc* and puts this element into the **MI_DATUM** structure that *value_buf* references.

If you need to save the data for later use, you must create your own copy of the data that *value_buf* references. For example, if the collection element is a character string, the data in *value_buf* is a pointer to an **mi_lvarchar** structure. To save the data in the **mi_lvarchar** structure, you can copy it with the **mi_var_copy( )** function.

Return Values

MI_NULL_VALUE	indicates that the returned data is NULL.
MI_END_OF_DATA	indicates that no more data remains to be fetched or the *jump* position is past the last element.
MI_ERROR	indicates that the connection handle is invalid.
MI_ROW_VALUE	indicates that the fetched element is a row.
MI_COLLECTION_VALUE	indicates that the fetched element is a collection.
MI_NORMAL_VALUE	indicates that execution of the function was successful.

The **mi_collection_fetch( )** function raises an exception for a bad argument, such as a null connection.

Related Topics

See also the descriptions of **mi_collection_close( ), mi_collection_copy( ), mi_collection_create( ), mi_collection_delete( ), mi_collection_free( ), mi_collection_insert( ), mi_collection_open( ), mi_collection_update( ), mi_open( ), mi_server_connect( ), mi_server_reconnect( ),** and **mi_var_copy( ).**

For descriptions of collections and of MI_DATUM values, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_collection_free( )

The **mi_collection_free( )** function frees a collection.

Syntax

```
mi_integer mi_collection_free(conn, coll_ptr)
   MI_CONNECTION *conn;
   MI_COLLECTION *coll_ptr;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
coll_ptr	is a pointer to the collection structure.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_collection_free( )** function frees the collection that *coll_ptr* references. This function is the destructor function for the collection structure.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_collection_close( ), mi_collection_copy( ), mi_collection_create( ), mi_collection_delete( ), mi_collection_fetch( ), mi_collection_insert( ), mi_collection_open( ), mi_collection_update( ), mi_open( ), mi_server_connect( ),** and **mi_server_reconnect( ).**

For a description of collections, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_collection_insert( )

The **mi_collection_insert( )** function inserts a single element into a collection.

Syntax

```
mi_integer mi_collection_insert(conn, coll_desc, insrt_datum, action, jump)
   MI_CONNECTION *conn;
   MI_COLL_DESC *coll_desc;
   MI_DATUM insrt_datum;
   MI_CURSOR_ACTION action;
   mi_integer jump;
```

conn is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**

coll_desc is a pointer to the collection descriptor.

insrt_datum is the MI_DATUM value to contain the collection element that **mi_collection_insert( )** inserts.

action determines the orientation of the insertion. When a collection opens, elements are available in a cursor. The current cursor position is before the first element. Possible *action* values follow:

MI_CURSOR_NEXT	Inserts an element *after* the current cursor position.
MI_CURSOR_PRIOR	Moves back one element and inserts an element *before* this new cursor location.
MI_CURSOR_FIRST	Inserts an element *before the first* element.
MI_CURSOR_LAST	Inserts an element *after the last* element.
MI_CURSOR_ABSOLUTE	Moves *jump* elements from the beginning of the cursor and inserts an element *before* the new cursor position.
MI_CURSOR_RELATIVE	Moves *jump* elements from the current position and inserts an element *before* this new cursor position.
MI_CURSOR_CURRENT	Inserts an element *before* the current cursor position.

jump is the absolute or relative offset of the insertion for LIST collections only. If *action* is not MI_CURSOR_ABSOLUTE or MI_CURSOR_RELATIVE when *jump* is specified, the function raises an exception and returns MI_NULL_VALUE.

For absolute positioning, a *jump* value of 1 is the first element.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_collection_insert( )** function inserts the element that *insrt_datum* references into the open collection that *coll_desc* references. The *action* argument determines where the new element value is inserted. This function inserts the specified element into the collection cursor associated with *coll_desc* and obtains the element to insert from an MI_DATUM value that *insrt_datum* references.

Return Values

MI_OK	indicates that the function was successful.
MI_NULL_VALUE	is returned when *jump* is not zero and the collection is not a list.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_collection_close( ), mi_collection_copy( ), mi_collection_create( ), mi_collection_delete( ), mi_collection_fetch( ), mi_collection_free( ), mi_collection_open( ), mi_collection_update( ), mi_open( ), mi_server_connect( ),** and **mi_server_reconnect( ).**

For descriptions of collections and of MI_DATUM values, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_collection_open( )

The **mi_collection_open( )** function opens a collection.

Syntax

```
MI_COLL_DESC *mi_collection_open(conn, coll_ptr)
   MI_CONNECTION *conn;
   MI_COLLECTION *coll_ptr;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
coll_ptr	is a pointer to the collection structure to open.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_collection_open( )** function opens the collection that the *coll_ptr* argument references. This function, a constructor function for the collection, creates a collection descriptor for the open collection. Other DataBlade API collection functions use this collection descriptor to access the elements of a collection.

Server Only

The **mi_collection_open( )** function allocates a new collection descriptor in the current memory duration.

End of Server Only

The collection descriptor contains the collection cursor, which provides access to the collection element by element. This cursor is an update scroll cursor. To use some other type of collection cursor for the collection, use the **mi_collection_open_with_options( )** function.

Return Values

An **MI_COLL_DESC** pointer	is the address of the collection descriptor for the opened collection.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_collection_close( ), mi_collection_copy( ), mi_collection_create( ), mi_collection_delete( ), mi_collection_fetch( ), mi_collection_free( ), mi_collection_insert( ), mi_collection_open_with_options( ), mi_collection_update( ), mi_open( ), mi_server_connect( ),** and **mi_server_reconnect( ).**

For a description of collections, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_collection_open_with_options( )

The **mi_collection_open_with_options( )** function opens a collection in a specified open mode.

Syntax

```
MI_COLL_DESC *mi_collection_open(conn, coll_ptr, control)
   MI_CONNECTION *conn;
   MI_COLLECTION *coll_ptr;
   mi_integer control;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
coll_ptr	is a pointer to the collection structure to open.
control	is a bit-mask integer value that specifies the type of collection cursor to create and open. The following bit-mask flags are valid:

MI_COLL_READONLY	Cursor is read-only.
MI_COLL_NOSCROLL	Cursor is a sequential cursor (*not* a scroll cursor).

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_collection_open_with_options( )** function is a constructor function for the collection descriptor. The function opens the collection that *coll_ptr* references and returns a collection descriptor for the open collection. Other DataBlade API collection functions use this collection descriptor to access the elements of a collection.

Server Only

The **mi_collection_open_with_options( )** function allocates a new collection descriptor in the current memory duration.

End of Server Only

This function creates a collection cursor to hold the collection elements. By default, this cursor is an update scroll cursor. If this type of cursor is not adequate for your DataBlade API module, you can create a collection cursor with characteristics specified by a bit mask in the *control* argument.

Collection Cursor Type	Control-Flag Value
Update scroll cursor	*None (default)*
Read-only scroll cursor	MI_COLL_READONLY
Update sequential cursor	MI_COLL_NOSCROLL
Read-only sequential cursor	MI_COLL_READONLY + MI_COLL_NOSCROLL

This function is useful for accessing collection subqueries.

Return Values

An **MI_COLL_DESC** pointer
: is the address of the collection descriptor for the opened collection.

NULL
: indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_collection_close( ), mi_collection_copy( ), mi_collection_create( ), mi_collection_delete( ), mi_collection_fetch( ), mi_collection_free( ), mi_collection_insert( ), mi_collection_open( ), mi_collection_update( ), mi_open( ), mi_server_connect( ),** and **mi_server_reconnect( ).**

For a description of collections, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_collection_update( )

The **mi_collection_update( )** function updates a collection.

Syntax

```
mi_integer mi_collection_update(conn, coll_desc, upd_datum, action, jump)
   MI_CONNECTION *conn;
   MI_COLL_DESC *coll_desc;
   MI_DATUM upd_datum;
   MI_CURSOR_ACTION action;
   mi_integer jump;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
coll_desc	is a pointer to the collection descriptor.
upd_datum	is a value or the address of data to insert into the collection. You can pass **int2**, **int4**, float for SQLSMFLOAT, and fixed-length user-defined types by value and all other types by reference.
action	determines the orientation of the update. When a collection opens, elements are available in a cursor. The current cursor position is before the first element. Possible *action* values follow:

MI_CURSOR_NEXT	Updates the next element *after* the current cursor position.
MI_CURSOR_PRIOR	Updates the element *before* the current cursor position.
MI_CURSOR_FIRST	Updates the *first* element.
MI_CURSOR_LAST	Updates the *last* element.
MI_CURSOR_ABSOLUTE	Moves *jump* elements from the beginning of the cursor and updates the element at the new cursor position.
MI_CURSOR_RELATIVE	Moves *jump* elements from the current position and updates the element at this new cursor position.
MI_CURSOR_CURRENT	Updates the element at the current cursor position.

jump	is the absolute or relative offset in the collection for LIST collections only. If *action* is not MI_CURSOR_ABSOLUTE or MI_CURSOR_RELATIVE when *jump* is specified, the function raises an exception and returns MI_NULL_VALUE. For absolute positioning, a *jump* value of 1 is the first element.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_collection_update( )** function updates the element that the *action* argument indicates in the open collection that *coll_desc* references. The function updates the element with the MI_DATUM value that *upd_datum* references. The function updates the specified element from the collection cursor associated with *coll_desc*.

Return Values

MI_OK	indicates that the function was successful.
MI_NULL_VALUE	is returned when *jump* is not zero and the collection is not a list.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_collection_close( ), mi_collection_copy( ), mi_collection_create( ), mi_collection_delete( ), mi_collection_fetch( ), mi_collection_free( ), mi_collection_insert( ), mi_collection_open( ), mi_open( ), mi_server_connect( ),** and **mi_server_reconnect( ).**

For descriptions of collections and of MI_DATUM values, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_column_count( )

The **mi_column_count( )** function obtains the number of columns in a row descriptor.

Syntax

```
mi_integer mi_column_count(row_desc)
   MI_ROW_DESC *row_desc;
```

row_desc	is a pointer to the row descriptor for which to count columns.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

A row descriptor can describe the structure of a row in a table or the structure of a row type. From the row descriptor that *row_desc* references, the **mi_column_count( )** function obtains the column count for either structure:

- The number of columns in the row
- The number of fields for the row type

Use the **mi_column_count( )** function to process returned row data on a column-by-column (or field-by-field) basis.

Return Values

>=0	is the number of columns or fields in the specified row descriptor.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_column_id( ), mi_column_name( ), mi_column_nullable( ), mi_column_precision( ), mi_column_scale( ), mi_column_type_id( ), mi_column_typedesc( ), mi_get_row_desc( ),** and **mi_get_row_desc_without_row( ).**

For more information about row descriptors, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_column_id( )

The **mi_column_id( )** function obtains the column identifier of a specified column from a row descriptor.

Syntax

```
mi_integer mi_column_id(row_desc, colname)
   MI_ROW_DESC *row_desc;
   mi_string *colname;
```

row_desc	is a pointer to the row descriptor for the row that contains the specified column.
colname	is a pointer to a null-terminated string to contain the column name.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

A row descriptor can describe the structure of a row in a table or the structure of a row type. From the row descriptor that *row_desc* references, the **mi_column_id( )** function obtains a column identifier for either structure:

- The column identifier of the *colname* column in the row
- The column identifier of the *colname* field for the row type

A column identifier is the position of the column or field within the row descriptor. Column numbering follows C array-indexing conventions: the first column in the row is at position zero (`0`).

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.

Return Values

>=0	is the column position of the specified column or field in the specified row descriptor.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_column_count( ), mi_column_name( ), mi_column_nullable( ), mi_column_precision( ), mi_column_scale( ), mi_column_type_id( ), mi_column_typedesc( ), mi_get_row_desc( ),** and **mi_get_row_desc_without_row( ).**

For more information about row descriptors, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_column_name( )

The **mi_column_name( )** function obtains the name of a specified column from a row descriptor.

Syntax

```
mi_string *mi_column_name(row_desc, column_id)
   MI_ROW_DESC *row_desc;
   mi_integer column_id;
```

row_desc	is a pointer to the row descriptor for the row that contains the column.
column_id	indicates the column identifier, which specifies the position of the column in the specified row descriptor. Column numbering follows C array-indexing conventions: the first column in the row descriptor is at position zero (0).

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

A row descriptor can describe the structure of a row in a table or the structure of a row type. From the row descriptor that *row_desc* references, the **mi_column_name( )** function obtains the name of a column for either structure:

- The name of the column at position *column_id* in the row
- The name of the field at position *column_id* for the row type

The *name* that **mi_column_name( )** returns is a pointer to a null-terminated string.

Server Only

The **mi_column_name( )** function allocates memory in the current memory duration for the string that it returns.

End of Server Only

Return Values

An **mi_string** pointer	is a pointer to the null-terminated column or field name.
NULL	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_column_count( ), mi_column_id( ), mi_column_nullable( ), mi_column_precision( ), mi_column_scale( ), mi_column_type_id( ), mi_column_typedesc( ), mi_get_row_desc( ),** and **mi_get_row_desc_without_row( ).**

For more information about row descriptors, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_column_nullable( )

The **mi_column_nullable( )** function indicates whether a specified column in a row descriptor can contain SQL NULL values.

Syntax

```
mi_integer mi_column_nullable(row_desc, column_id)
   MI_ROW_DESC *row_desc;
   mi_integer column_id;
```

row_desc	is a pointer to the row descriptor of the row that contains the column.
column_id	indicates the column identifier of the column, which specifies the position of the column in the specified row descriptor. Column numbering follows C array-indexing conventions: the first column in the row is at position zero (0).

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

A row descriptor can describe the structure of a row in a table or the structure of a row type. From the row descriptor that *row_desc* references, the **mi_column_nullable( )** function checks for a NOT NULL column-level constraint for either structure:

- Whether the column at position *column_id* in the row can contain SQL NULL values
- Whether the field at position *column_id* for the row type can contain SQL NULL values

When you declare a table, you can specify that a particular column is not able to hold NULL values with the NOT NULL column-level constraint. For more information on column-level constraints, see the description of CREATE TABLE in the *IBM Informix Guide to SQL: Syntax*.

Return Values

0	indicates that the specified column or field is defined to reject SQL NULL values; that is, it was defined with the NOT NULL constraint.
1	indicates that the specified column or field is defined to accept SQL NULL values; that is, it has *not* been defined with the NOT NULL constraint.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_column_count( ), mi_column_id( ), mi_column_name( ), mi_column_precision( ), mi_column_scale( ), mi_column_type_id( ), mi_column_typedesc( ), mi_get_row_desc( ),** and **mi_get_row_desc_without_row( ).**

For more information about row descriptors, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_column_precision( )

The **mi_column_precision( )** function obtains the precision of the specified column from a row descriptor.

Syntax

```
mi_integer mi_column_precision(row_desc, column_id)
   MI_ROW_DESC *row_desc;
   mi_integer column_id;
```

row_desc	is a pointer to the row descriptor of the row that contains the column.
column_id	indicates the column identifier, which specifies the position of the column in the specified row descriptor. Column numbering follows C array-indexing conventions: the first column in the row is at position zero (0).

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

A row descriptor can describe the structure of a row in a table or the structure of a row type. From the row descriptor that *row_desc* references, the **mi_column_precision( )** function obtains the precision of a column for either structure:

- The precision of the column at position *column_id* in the row
- The precision of the field at position *column_id* for the row type

The precision is an attribute of the column data type and represents the total number of digits the column can hold, as follows.

Data Type	Meaning
DECIMAL, MONEY	Number of significant digits in the fixed-point or floating-point (DECIMAL) column
DATETIME, INTERVAL	Number of digits that are stored in the date and/or time column with the specified qualifier
Character, Varying-character	Maximum number of characters in the column

If you call **mi_column_precision( )** on a column or row-type field of some other data type, the function returns zero (0).

Return Values

0	indicates that no precision was set for the specified column or field.
>0	is the precision of the specified column or field.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_column_count( ), mi_column_id( ), mi_column_name( ), mi_column_nullable( ), mi_column_scale( ), mi_column_type_id( ), mi_column_typedesc( ), mi_get_row_desc( ),** and **mi_get_row_desc_without_row( ).**

For more information about row descriptors or about the precision of a fixed-point data type, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_column_scale( )

The **mi_column_scale( )** function obtains the scale of the specified column from a row descriptor.

Syntax

```
mi_integer mi_column_scale(row_desc, column_id)
   MI_ROW_DESC *row_desc;
   mi_integer column_id;
```

row_desc	is a pointer to the row descriptor of the row that contains the column.
column_id	indicates the column identifier, which specifies the position of the column in the specified row descriptor. Column numbering follows C array-indexing conventions: the first column in the row is at position zero (0).

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

A row descriptor can describe the structure of a row in a table or the structure of a row type. From the row descriptor that *row_desc* references, the **mi_column_scale( )** function obtains the scale of a column for either structure:

- The scale of the column at position *column_id* in the row
- The scale of the field at position *column_id* for the row type

The scale is an attribute of the data type. The meaning of the scale depends on the associated data type, as the following table shows.

Data Type	Meaning of Scale
DECIMAL (fixed-point), MONEY	The number of digits to the right of the decimal point
DECIMAL (floating-point)	The value 255
DATETIME, INTERVAL	The encoded integer value for the end qualifier of the data type, which *end_qual* represents in the following qualifier: *start_qual* TO *end_qual*

If you call **mi_column_scale( )** on a column or row-type field of some other data type, the function returns zero (0).

Return Values

0	indicates that the data type of the specified column or field is something other than DECIMAL, MONEY, FLOAT, or SMALLFLOAT.
>0	is the scale of the specified column or field.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_column_count( ), mi_column_id( ), mi_column_name( ), mi_column_nullable( ), mi_column_precision( ), mi_column_type_id( ), mi_column_typedesc( ), mi_get_row_desc( ),** and **mi_get_row_desc_without_row( ).**

For more information about row descriptors or about the scale of a fixed-point data type, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_column_type_id( )

The **mi_column_type_id( )** function obtains the type identifier of the specified column from a row descriptor.

Syntax

```
MI_TYPEID *mi_column_type_id(row_desc, column_id)
   MI_ROW_DESC *row_desc;
   mi_integer column_id;
```

row_desc	is a pointer to the row descriptor of the row that contains the column.
column_id	indicates the column identifier, which specifies the position of the column in the specified row descriptor. Column numbering follows C array-indexing conventions: the first column in the row is at position zero (0).

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

A row descriptor can describe the structure of a row in a table or the structure of a row type. From the row descriptor that *row_desc* references, the **mi_column_type_id( )** function obtains the type identifier of a column for either structure:

- The type identifier of the column at position *column_id* in the row
- The type identifier of the field at position *column_id* for the row type

The type identifier is an integer that uniquely identifies a data type.

Return Values

An **MI_TYPEID** pointer	is a pointer to the type identifier of the data type for the specified column or field.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_column_count( ), mi_column_id( ), mi_column_name( ), mi_column_nullable( ), mi_column_precision( ), mi_column_scale( ), mi_column_typedesc( ), mi_get_row_desc( ),** and **mi_get_row_desc_without_row( ).**

For more information about row descriptors or about type identifiers, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_column_typedesc( )

The **mi_column_typedesc( )** function obtains the type descriptor of the specified column from a row descriptor.

Syntax

```
MI_TYPE_DESC *mi_column_typedesc(row_desc, column_id)
   MI_ROW_DESC *row_desc;
   mi_integer column_id;
```

row_desc	is a pointer to the row descriptor of the row that contains the column.
column_id	indicates the column identifier, which specifies the position of the column in the specified row descriptor. Column numbering follows C array-indexing conventions: the first column in the row is at position zero (0).

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

A row descriptor can describe the structure of a row in a table or the structure of a row type. From the row descriptor that *row_desc* references, the **mi_column_typedesc( )** function obtains the type descriptor of a column for either structure:

- The type descriptor for the column at position *column_id* in the row
- The type descriptor for the field at position *column_id* for the row type

The type descriptor is a DataBlade API structure that provides information about a data type.

Return Values

An **MI_TYPE_DESC** pointer	is a pointer to the type descriptor for the specified column or field.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_column_count( ), mi_column_id( ), mi_column_name( ), mi_column_nullable( ), mi_column_precision( ), mi_column_scale( ), mi_column_type_id( ), mi_get_row_desc( ),** and **mi_get_row_desc_without_row( ).**

For more information about row descriptors or about type descriptors, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_command_is_finished( )

The **mi_command_is_finished( )** function reports whether the current statement has finished executing.

Syntax

```
mi_integer mi_command_is_finished(conn)
   MI_CONNECTION *conn;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

The **mi_command_is_finished( )** function determines whether execution has completed for the current statement on the connection that *conn* references. The current statement is the most recently executed SQL statement sent to the database server on that connection. The current statement is finished when either of the following conditions occurs:

- Successful completion

  The database server has executed the statement and the DataBlade API module has retrieved the last row from any query.
- Unsuccessful completion

  A database server exception has terminated the statement abnormally.

The current statement must be finished before the database server can process the next statement. Use the **mi_query_finish( )** function to force a statement to finish processing.

Return Values

0	indicates that the current statement active on the current connection has *not* completed.
1	indicates that the current statement active on the current connection has completed.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_result( ), mi_open( ), mi_query_finish( ), mi_server_connect( ),** and **mi_server_reconnect( ).**

mi_convert_from_codeset( )

The **mi_convert_from_codeset( )** function converts an input string from a code set in a specified locale to that of the server-processing locale.

Syntax

```
mi_integer mi_convert_from_codeset(string, locale_name)
   char *string;
   char *locale_name;
```

string	is a pointer to the string data to convert.
locale_name	is the name of the locale whose code set to use to convert the data.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	No

Usage

Global Language Support

The database server automatically converts text columns (CHAR, NCHAR, NVARCHAR, TEXT, VARCHAR) between the server-processing and client locales. However, the **mi_convert_from_codeset( )** function is useful for a user-defined type that has an embedded string or additional information about local code sets. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Related Topics

See also the description of **mi_convert_to_codeset( ).**

mi_convert_to_codeset( )

The **mi_convert_to_codeset( )** function converts a string to a code set in a specified locale.

Syntax

```
mi_integer mi_convert_to_codeset(string, locale_name)
   char *string;
   char *locale_name;
```

string	is a pointer to the string data to convert.
locale_name	is the name of the locale whose code set to use to convert the data.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	No

Usage

Global Language Support

The database server automatically converts text columns (CHAR, NCHAR, NVARCHAR, TEXT, VARCHAR) between the server-processing and client locales. However, the **mi_convert_to_codeset( )** function is useful for a user-defined type that has an embedded string or other strings that require code-set conversion. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Related Topics

See also the description of **mi_convert_from_codeset( ).**

mi_current_command_name( )

The **mi_current_command_name( )** function returns the name of the SQL statement that invoked the C UDR.

Syntax

```
mi_string *mi_current_command_name(*conn)
   MI_CONNECTION *conn;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_current_command_name( )** function returns the SQL statement name as a null-terminated string in memory that the function allocates with the current memory duration. This statement name is only the verb of the statement, not the entire statement syntax.

For example, suppose the following statement executes:

```
DELETE FROM customer WHERE customer_id = 1998;
```

The **mi_current_command_name( )** function returns only the verb of this statement: `delete`.

The *conn* parameter is redundant when you call **mi_current_command_name( )** within the context of a user-defined routine. This function is useful in a user-defined routine that needs to determine what kind of SQL statement invoked it.

Important: The **mi_current_command_name( )** function does not return the name of the current SQL statement or of an SQL prepared statement. To obtain the name of the current SQL statement, use the **mi_result_command_name( )** function after the **mi_get_result( )** function returns *MI_DML* or *MI_DDL*. To return the name of an SQL prepared statement, you can use the **mi_statement_command_name( )** function.

Return Values

An **mi_string** pointer	is a pointer to the verb of the last statement or command.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_result( ), mi_result_command_name( ),** and **mi_statement_command_name( ).**

For more information on how to call a C UDR with an SQL statement, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_dalloc( )

The **mi_dalloc( )** function allocates the specified amount of memory for the specified memory duration and returns a pointer to the allocated block.

Syntax

```
void *mi_dalloc (size, duration)
   mi_integer size;
   MI_MEMORY_DURATION duration;
```

size	is the number of bytes to allocate.
duration	is a value that specifies the memory duration of the user memory to allocate. Valid values for *duration* follow:

PER_ROUTINE	For the duration of one iteration of the UDR
PER_COMMAND	For the duration of the execution of the current subquery
PER_STATEMENT *(Deprecated)*	For the duration of the current SQL statement
PER_STMT_EXEC	For the duration of the execution of the current SQL statement
PER_STMT_PREP	For the duration of the current prepared SQL statement
PER_TRANSACTION *(Advanced)*	For the duration of one transaction
PER_SESSION *(Advanced)*	For the duration of the current client session
PER_SYSTEM *(Advanced)*	For the duration of the database server execution

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes (however, the application ignores memory duration)	Yes

Usage

The **mi_dalloc( )** function allocates *size* number of bytes of shared memory with *duration* memory duration for a DataBlade API module. The **mi_dalloc( )** function is a constructor function for user memory. This function behaves exactly like **mi_alloc( )** except that **mi_dalloc( )** enables you to specify the memory duration explicitly at allocation time.

Server Only

For most memory allocations in a C UDR, the *duration* argument should be one of the following public memory-duration constants:

- PER_ROUTINE (or PER_FUNCTION)

If you specify PER_ROUTINE, the database server frees the allocated memory when the UDR returns.

- PER_COMMAND

  If you specify PER_COMMAND, the database server frees memory when the execution of a subquery is complete.
- PER_STMT_EXEC

  If you specify PER_STMT_EXEC, the database server frees memory when the execution of an SQL statement is complete.
- PER_STMT_PREP

  If you specify PER_STMT_PREP, the database server frees memory when a prepared SQL statement terminates.

Warning: Use an advanced memory duration in your C UDR 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.

End of Server Only

In C UDRs, the database server automatically frees memory allocated with **mi_dalloc( )** when an exception is raised.

Important: In C UDRs, use DataBlade API memory-management functions to allocate memory. Use of a DataBlade API memory-management function guarantees that the database server can automatically free the memory, especially in case of a return value or exception, where the routine would not be able to free the memory.

A UDR can use **mi_free( )** to free memory allocated by **mi_dalloc( )** explicitly when that memory is no longer needed.

Client Only

In client LIBMI applications, **mi_dalloc( )** works exactly like **malloc( )**: storage is allocated on the heap of the client process. The database server, however, does *not* automatically free this memory. Therefore, the client LIBMI application *must* use **mi_free( )** to free explicitly all allocations that **mi_dalloc( )** makes.

Important: Client LIBMI applications ignore memory duration.

End of Client Only

Within a callback function, a call to **mi_dalloc( )** that requests a duration of PER_COMMAND returns NULL. Therefore, a callback function must call **mi_dalloc( )** with a duration of PER_ROUTINE.

The **mi_dalloc( )** function returns a pointer to the newly allocated memory. Cast this pointer to match the structure of the user-defined buffer or structure that you allocate.

Return Values

A **void** pointer	is a pointer to the newly allocated memory. Cast this pointer to match the user-defined buffer or structure for which the memory was allocated.

NULL	indicates that the function was unable to allocate the memory.

The **mi_dalloc( )** function does *not* throw an MI_Exception event when it encounters a runtime error. Therefore, the function does not cause callbacks to be invoked.

Related Topics

See also the descriptions of **mi_alloc( ), mi_free( ), mi_switch_mem_duration( ),** and **mi_zalloc( ).**

For more information on memory allocation and memory duration, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_date_to_binary( )

The **mi_date_to_binary( )** function converts a text (string) representation of a date value to its binary (internal) DATE representation.

Syntax

```
mi_date mi_date_to_binary(date_string)
   mi_lvarchar *date_string;
```

date_string	is a pointer to the date string to convert to its internal DATE format.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_date_to_binary( )** function converts the date string that *date_string* references to the internal DATE value. An internal DATE value is the format that the database server uses to store a date in a column of the database.

Global Language Support

The **mi_date_to_binary( )** function accepts the date string in the date format of the current processing locale. The function also performs any code-set conversion necessary between the current processing locale and the target locale.

Important: The **mi_date_to_binary( )** function is supported only for compatibility with earlier versions of existing DataBlade API modules. The DataBlade API will eventually discontinue support for this function. Use the **mi_string_to_date( )** function in any new DataBlade API modules.

End of Global Language Support

Return Values

An **mi_date** value	is the internal (binary) DATE representation of *date_string*.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_binary_to_date( )** and **mi_string_to_date( ).**

mi_date_to_string( )

The **mi_date_to_string( )** function creates a text (string) representation of a date value from the binary (internal) DATE representation.

Syntax

```
mi_string *mi_date_to_string(date_data)
   mi_date date_data;
```

date_data	is the internal DATE format to convert to a date string.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_date_to_string( )** function converts the internal DATE value that *date_data* contains into a date string. An internal DATE value is the format that the database server uses to store a value in a DATE column of the database.

Important: The **mi_date_to_string( )** function replaces the **mi_binary_to_date( )** function for converting an internal DATE value to a date string in DataBlade API modules.

Global Language Support

The **mi_date_to_string( )** function formats the date string in the date format of the current processing locale. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Return Values

An **mi_string** pointer	is a pointer to the date string equivalent of *date_data*.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_datetime_to_string( ), mi_decimal_to_string( ), mi_interval_to_string( ), mi_money_to_string( ),** and **mi_string_to_date( ).**

For more information on how to convert internal DATE format to date strings, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_datetime_to_binary( )

The **mi_datetime_to_binary( )** function converts a text (string) representation of a date, time, or date and time value to its binary (internal) DATETIME representation.

Syntax

```
mi_datetime *mi_datetime_to_binary(dt_string)
   mi_lvarchar *dt_string;
```

dt_string	is a pointer to a date, time, or date and time string to convert to its internal DATETIME format.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_datetime_to_binary( )** function converts the date, time, or date and time string that *dt_string* references to its internal DATETIME value. An internal DATETIME value is the format that the database server uses to store a value in a DATETIME column of the database.

Global Language Support

The **mi_datetime_to_binary( )** function accepts the date, time, or date and time string in the date and time format of the current processing locale. The function also performs any code-set conversion necessary between the current processing locale and the target locale.

Important: The **mi_datetime_to_binary( )** function is supported only for compatibility with earlier versions of existing DataBlade API modules. The DataBlade API will eventually discontinue support for this function. Use the **mi_string_to_datetime( )** function in any new DataBlade API modules.

End of Global Language Support

Return Values

An **mi_datetime** pointer	is a pointer to the internal DATETIME representation of *dt_string*.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_binary_to_datetime( )** and **mi_string_to_datetime( ).**

mi_datetime_to_string( )

The **mi_datetime_to_string( )** function creates an ANSI SQL standard text (string) representation of a date, time, or date and time value from the binary (internal) DATETIME representation.

Syntax

```
mi_string *mi_datetime_to_string(dt_data)
   mi_datetime *dt_data;
```

dt_data	is a pointer to the internal DATETIME representation of the date, time, or date and time value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_datetime_to_string( )** function converts the internal DATETIME value that *dt_data* references into a date, time, or date and time string. This string has the following ANSI SQL standard format:

```
"YYYY-MM-DD HH:mm:SS.FFFFF"
```

YYYY	is the 4-digit year.
MM	is the 2-digit month.
DD	is the 2-digit day.
HH	is the 2-digit hour.
mm	is the 2-digit minute.
SS	is the 2-digit second.
FFFFF	is the fraction of a second, in which the date, time, or date and time qualifier specifies the number of digits, with a maximum precision of 5 digits.

If the internal DATETIME value contains only a subset of this range, **mi_datetime_to_string( )** creates a date, time, or date and time string with the appropriate portion of the preceding format. For example, suppose *dt_data* references the internal format of the date *01/31/07* and a time of 10:30 A.M. The **mi_datetime_to_string( )** function returns an **mi_string** value with the following date and time string:

```
"2007-01-31 10:30"
```

Important: The **mi_datetime_to_string( )** function replaces the **mi_binary_to_datetime( )** function for conversion of a DATETIME value to a date, time, or date and time string in DataBlade API modules.

Global Language Support

The **mi_datetime_to_string( )** function does *not* format the date, time, or date and time string in the date and time formats of the current processing locale.

End of Global Language Support

Return Values

An **mi_string** pointer	is a pointer to the date, time, or date and time string equivalent of *dt_data*.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_date_to_string( ), mi_decimal_to_string( ), mi_interval_to_string( ), mi_money_to_string( ),** and **mi_string_to_datetime( ).**

For more information on how to convert internal DATETIME values to date, time, or date and time strings, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_db_error_raise( )

The mi_db_error_raise( ) function raises an error or warning and sends the message to the calling program.

Syntax

```
mi_integer mi_db_error_raise (conn, msg_type, msg, optional_parameters)
   MI_CONNECTION *conn;
   mi_integer msg_type;
   char *msg;
   optional_parameters;
```

conn	is either a NULL-valued pointer or a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
msg_type	is a message-type constant: MI_MESSAGE, MI_EXCEPTION, or MI_SQL.
msg	If *msg_type* is MI_MESSAGE or MI_EXCEPTION, the *msg* value is the text of the message to pass. If *msg_type* is MI_SQL, the *msg* value is the five-character **SQLSTATE** value that represents the error or warning in the **syserrors** system catalog table, followed by a null terminator.
optional_parameters	If *msg_type* is MI_SQL and the text of the error or warning message requires parameters, **mi_db_error_raise( )** accepts a value for each parameter marker in the message.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_db_error_raise( ) function** raises an MI_Exception event to the database server. If the DataBlade API program registers a callback function to handle MI_Exception events, the database server invokes this callback. By default, the client application receives any messages that **mi_db_error_raise( )** raises, regardless of how the exception is raised on the database server.

Server Only

The *conn* parameter can be either of the following values:

- A NULL-valued pointer, which tells the database server to raise the exception against the parent connection
- A pointer to a valid connection descriptor for the connection against which to raise the exception

End of Server Only

Client Only

In a client LIBMI application, the *conn* parameter of **mi_db_error_raise( )** must point to a valid connection descriptor. When a client LIBMI application invokes the **mi_db_error_raise( )** function, the exception is passed to the database server for processing. If the client LIBMI application has not registered a callback for MI_Exception, the database server executes the system-default callback (which the **mi_default_callback( )** function implements).

End of Client Only

For more information, see the *IBM Informix DataBlade API Programmer's Guide*.

The **mi_db_error_raise( )** function can either pass a literal message or raise an SQL error or warning stored in the **syserrors** system catalog table. The *msg_type* parameter can contain one of the following values.

Message Type	**Description**
MI_MESSAGE	The **mi_db_error_raise( )** function raises a warning (an MI_Exception event with an exception level of MI_MESSAGE) with the *msg* warning message.
MI_EXCEPTION	The **mi_db_error_raise( )** function raises an error (an MI_Exception event with an exception level of MI_EXCEPTION) with the *msg* error message.
MI_SQL	The **mi_db_error_raise( )** function raises an exception with a custom message from the **syserrors** system catalog table whose **SQLSTATE** value is in *msg*. For more information on how to use custom messages, see the *IBM Informix DataBlade API Programmer's Guide*.

Global Language Support

The **locale** column of the **syserrors** system catalog table is used for the internationalization of error and warning messages. It is recommended that you use the **syserrors** catalog table to maintain language independence of messages. For more information on internationalized exception messages, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

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!!!");
```

The following call to **mi_db_error_raise( )** returns the predefined SQL message that is associated with an **SQLSTATE** value of ″`03I01`″ and has no markers:

```
mi_db_error_raise (conn, MI_SQL, "03I01", NULL);
```

Custom messages can contain parameter markers, whose values the *optional_parameters* parameter specifies. The *optional_parameters* parameter list specifies a pair of values for each parameter marker in the message and is terminated with a NULL-valued pointer.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful, with the following exception: when used in UDRs to raise fatal errors that are not handled, it does not return control to the calling function.

Related Topics

See also the descriptions of **mi_default_callback( ), mi_errmsg( ), mi_error_desc_copy( ), mi_error_desc_destroy( ), mi_error_desc_is_copy( ), mi_error_level( ), mi_error_sql_state( ),** and **mi_register_callback( ).**

For more information on how to raise exceptions, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_dbcreate( )

The **mi_dbcreate( )** function creates a new database on the given connection.

Syntax

```
mi_integer mi_dbcreate(conn, db_info, dbspace, create_flag)
   MI_CONNECTION *conn;
   const MI_DATABASE_INFO *db_info;
   const char *dbspace;
   MI_DBCREATE_FLAGS create_flag;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_server_connect( )** or **mi_server_reconnect( ).** The connection must not be an open (logged-in) connection. The *conn* value specifies the database server on which to create the database. A NULL-valued pointer specifies the default database server.
db_info	describes the new database.
dbspace	is the storage location for the new database. A NULL value specifies the root dbspace.
create_flag	is one of the following options:

MI_DBCREATE_DEFAULT	Creates the database without logging.
MI_DBCREATE_LOG	Creates the database with a log.
MI_DBCREATE_LOG_BUFFERED	Creates the database with a buffered log.
MI_DBCREATE_LOG_ANSI	Creates the database with ANSI log mode.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	No

Usage

This call is equivalent to the SQL CREATE DATABASE statement. Part of the argument *dbinfo* specifies the database name.

On entry, there must be no logged-in database active on the given connection.

For additional information about the **MI_DATABASE_INFO** structure, see the description of the **mi_get_database_info( )** function.

Return Values

MI_OK	indicates that the function was successful; the new database exists.
MI_ERROR	indicates that the function was not successful. An exception was raised, and the database was not created.

Related Topics

See also the descriptions of **mi_dbdrop( ), mi_get_database_info( ), mi_open( ), mi_server_connect( ),** and **mi_server_reconnect( ).**

mi_dbdrop( )

The **mi_dbdrop( )** function drops a database from the given connection.

Syntax

```
mi_integer mi_dbdrop(conn, db_info)
   MI_CONNECTION *conn;
   const MI_DATABASE_INFO *db_info;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
db_info	is a pointer to a database-information descriptor that identifies the database to drop.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	No

Usage

The **mi_dbdrop( )** function is equivalent to the SQL DROP DATABASE statement. A field in the database-information structure that *dbinfo* references contains the database name.

For additional information about the **MI_DATABASE_INFO** structure, see the description of the **mi_get_database_info( )** function.

Return Values

MI_OK	indicates that the function was successful; the database was dropped.
MI_ERROR	indicates that the function was not successful. An exception was raised, and the database was not created.

Related Topics

See also the descriptions of **mi_close( ), mi_dbcreate( ), mi_get_database_info( ), mi_dbdrop( ), mi_open( ), mi_server_connect( ),** and **mi_server_reconnect( ).**

mi_decimal_to_binary( )

The **mi_decimal_to_binary( )** function converts a text (string) representation of a decimal value to its binary (internal) DECIMAL representation.

Syntax

```
mi_decimal *mi_decimal_to_binary(decimal_string)
   mi_lvarchar *decimal_string;
```

decimal_string	is a pointer to the decimal string to convert to its internal DECIMAL format.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_decimal_to_binary( )** function converts the decimal string that *decimal_string* references to the internal DECIMAL value. An internal DECIMAL value is the format that the database server uses to store a value in a DECIMAL column of the database. This format can represent both fixed-point and floating-point decimal numbers.

Global Language Support

The **mi_decimal_to_binary( )** function accepts the decimal string in the numeric format of the current processing locale. The function also performs any code-set conversion necessary between the current processing locale and the target locale.

Important: The **mi_decimal_to_binary( )** function is supported only for compatibility with earlier versions of existing DataBlade API modules. The DataBlade API will eventually discontinue support for this function. Use the **mi_string_to_decimal( )** function in any new DataBlade API modules.

End of Global Language Support

Return Values

An **mi_decimal** pointer	is a pointer to the internal DECIMAL representation that **mi_decimal_to_binary( )** has created.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **"mi_binary_to_decimal( )" on page 2-93** and **mi_string_to_decimal( ).**

mi_decimal_to_string( )

The **mi_decimal_to_string( )** function creates a text (string) representation of a decimal value from the binary (internal) DECIMAL representation.

Syntax

```
mi_string *mi_decimal_to_string(decimal_data)
   mi_decimal *decimal_data;
```

decimal_data	is a pointer to the internal DECIMAL representation of the decimal value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_decimal_to_string( )** function converts the internal DECIMAL value that *decimal_data* contains into a decimal string. An internal DECIMAL value is the format that the database server uses to store a value in a DECIMAL column of the database.

Important: The **mi_decimal_to_string( )** function replaces the **mi_binary_to_decimal( )** function for internal DECIMAL-to-string conversion in DataBlade API modules.

Global Language Support

The **mi_decimal_to_string( )** function formats the decimal string in the numeric format of the current processing locale. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Return Values

An **mi_string** pointer	is a pointer to the decimal string that **mi_decimal_to_string( )** has created.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_date_to_string( ), mi_datetime_to_string( ), mi_interval_to_string( ), mi_money_to_string( ),** and **mi_string_to_decimal( ).**

For more information on how to convert internal DECIMAL values to decimal strings, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_default_callback( )

The **mi_default_callback( )** function is the system-default callback for all callbacks in a client LIBMI application.

Syntax

```
void mi_default_callback(event_type, conn, event_data, user_data)
   MI_EVENT_TYPE event_type;
   MI_CONNECTION *conn;
   void *event_data;
   void *user_data;
```

event_type	is the event type that the system-default callback is to handle.
conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
event_data	is a pointer to the event-specific structure for the callback.
user_data	is a pointer to the user-defined error structure. For the MI_LIB_DROPCONN error level of the MI_Client_Library_Error event, this argument is a flag to indicate whether to attempt a reconnection. If *user_data* is set to zero (`0`), the client LIBMI library attempts to reconnect to the database server; otherwise, no attempt is made to reconnect.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	No

Usage

The **mi_default_callback( )** function implements the system-default callback, which returns appropriate error and warning messages in response to client events. Client events include the MI_Client_Library_Error and MI_Exception events.

UNIX/Linux Only

On UNIX or Linux, the system-default callback sends the error or warning message to **stderr**.

End of UNIX/Linux Only

Windows Only

On Windows, the system-default callback displays the error or warning message in a Windows message box.

End of Windows Only

The client LIBMI library automatically calls the system-default callback when a client event occurs and the client LIBMI application has no callback registered for this event. A client LIBMI application can explicitly call **mi_default_callback( )**. For example, an application can register a special callback within a specific function and then re-register **mi_default_callback( )** after the function completes to return to default behavior.

Important: When a client LIBMI application connects to a database server, the **mi_default_callback( )** function does not report some warnings. The **SQLSTATE** values for these warnings begin with `01I`.

To override the default behavior of the system-default callback, the client application can register a callback that handles the client event with the **mi_register_callback( )** function.

Return Values

None.

Related Topics

See also the description of **mi_register_callback( ).**

mi_disable_callback( )

The **mi_disable_callback( )** function disables a callback for a single event or for all events.

Syntax

```
mi_integer mi_disable_callback(conn, event_type, cback_handle)
   MI_CONNECTION *conn;
   MI_EVENT_TYPE event_type;
   MI_CALLBACK_HANDLE *cback_handle;
```

conn	is either a NULL-valued pointer or a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
event_type	is the event type for the callback. For a list of valid event types, see the *IBM Informix DataBlade API Programmer's Guide*.
cback_handle	is a callback handle that a previous call to **mi_register_callback( )** has returned.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

The **mi_disable_callback( )** function disables the callback that *cback_handle* identifies for the event that *event_type* specifies. The *conn* value must be a pointer to the same connection descriptor as the connection on which the callback was registered with **mi_register_callback( ).**

Server Only

For a C UDR, *conn* must be NULL for the following event types:
- 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

Client Only

For a client LIBMI application, you must pass a valid connection descriptor to **mi_retrieve_callback( )** for callbacks that handle the following event types:
- MI_Exception
- MI_Xact_State_Change
- MI_Client_Library_Error

End of Client Only

The callback is automatically enabled when it is registered with **mi_register_callback( ).** You can explicitly disable the callback with the **mi_disable_callback( )** function.

Return Values

MI_OK	indicates that the function was successful; the callback was disabled.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_enable_callback( )** and **mi_register_callback( ).**

For a description of how to enable and disable callbacks or information on how to register a callback, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_drop_prepared_statement( )

The **mi_drop_prepared_statement( )** function drops a previously prepared statement.

Syntax

```
mi_integer mi_drop_prepared_statement(stmt_desc)
   MI_STATEMENT *stmt_desc;
```

stmt_desc	is a pointer to a statement descriptor that references a previously prepared statement. --

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_drop_prepared_statement( )** frees the statement descriptor that *stmt_desc* references. It is the destructor function for a statement descriptor. The **mi_prepare( )** function prepares a statement and returns a statement descriptor for the prepared statement. If a cursor (implicit or explicit) is associated with the prepared statement, **mi_drop_prepared_statement( )** also frees this cursor.

Important: It is recommended that you explicitly free prepared statements with **mi_drop_prepared_statement( )** once you no longer need them. Otherwise, these prepared statements remain until the associated session ends. To clear the cursor associated with a prepared statement instead of freeing the cursor, use the **mi_close_statement( )** function.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_close_statement( ), mi_exec_prepared_statement( ), mi_open_prepared_statement( ),** and **mi_prepare( ).**

mi_enable_callback( )

The **mi_enable_callback( )** function enables a callback for a specified event type.

Syntax

```
mi_integer mi_enable_callback(conn, event_type, cback_handle)
   MI_CONNECTION *conn;
   MI_EVENT_TYPE event_type;
   MI_CALLBACK_HANDLE *cback_handle;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
event_type	is the event type for the callback. For a list of valid event types, see the *IBM Informix DataBlade API Programmer's Guide*.
cback_handle	is a callback handle that a previous call to **mi_register_callback( )** has returned.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_enable_callback( )** function enables the callback that *cback_handle* identifies for the event that *event_type* specifies. The *conn* value must point to the same connection descriptor as the connection on which the callback was registered with **mi_register_callback( ).**

Server Only

For a C UDR, *conn* must be NULL for the following event types:
- 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

Client Only

For a client LIBMI application, you must pass a valid connection descriptor to **mi_retrieve_callback( )** for callbacks that handle the following event types:
- MI_Exception
- MI_Xact_State_Change
- MI_Client_Library_Error

End of Client Only

The callback is automatically enabled when it is registered with **mi_register_callback( ).** You can explicitly disable the callback with the **mi_disable_callback( )** function.

Return Values

MI_OK	indicates that the function was successful; the callback was enabled.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_disable_callback( )** and **mi_register_callback( ).**

For a description of how to enable and disable callbacks or information on how to register a callback, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_errmsg( )

The **mi_errmsg( )** function retrieves an error or warning message from an error descriptor into a user-allocated buffer.

Syntax

```
void mi_errmsg(err_desc, msgbuf, msgbuflen)
   MI_ERROR_DESC *err_desc;
   char *msgbuf;
   mi_integer msgbuflen;
```

err_desc	is a pointer to the error descriptor that describes the exception.
msgbuf	is a pointer to a user-allocated buffer to contain the message.
mgsbuflen	is the length of the *msgbuf* buffer.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_errmsg( )** function copies the text of an error or warning message from the error descriptor that *err_desc* references into the user-allocated buffer that *msgbuf* references. You must ensure that both *msgbuflen* and *msgbuf* are large enough to accommodate any error or warning text that the error descriptor might hold. If the buffer is too small, **mi_errmsg( )** truncates the message. System-generated messages, especially those that contain pathnames, can be quite long. A good starting value for *buflen* is usually `256`.

The **mi_errmsg( )** function always terminates the message with a null terminator, provided *msgbuflen* is greater than `0`. Message text that **mi_errmsg( )** retrieves for database server exceptions is the most specific text, that is, the text associated with the Informix **SQLCODE** value. The **mi_errmsg( )** function does *not* retrieve any ANSI or X/Open message text.

A separate, additional callback follows the callback for the **SQLCODE** value when an exception has an access-method error code (Informix RSAM status). This error code and message text are available through the **mi_errmsg( )** and **mi_error_sqlcode( )** functions.

Server Only

For a C UDR, **mi_errmsg( )** does *not* prefix the message text with the error number.

End of Server Only

Client Only

For a client LIBMI application, **mi_errmsg( )** adds error codes to the start of the message.

End of Client Only

The **mi_errmsg( )** function is useful in an exception callback for getting the error or warning message associated with an exception.

Return Values

None.

Related Topics

See also the descriptions of **mi_error_desc_copy( ), mi_error_desc_destroy( ), mi_error_desc_is_copy( ), mi_error_level( ), mi_error_sql_state( ),** and **mi_error_sqlcode( ).**

For general information about how to obtain information from an error descriptor, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_error_desc_copy( )

The **mi_error_desc_copy( )** function returns a copy of a specified error descriptor.

Syntax

```
MI_ERROR_DESC *mi_error_desc_copy(src_err_desc)
   MI_ERROR_DESC *src_err_desc;
```

src_err_desc is a pointer to the error descriptor to copy.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_error_desc_copy( )** function is the constructor function for the error descriptor. It creates a copy of the *src_err_desc* error descriptor.

Server Only

The **mi_error_desc_copy( )** function allocates a new error descriptor in the current memory duration.

End of Server Only

Important: Be sure to destroy the **MI_ERROR_DESC** structure with **mi_error_desc_destroy( )** when your DataBlade API code no longer needs it.

Return Values

An **MI_ERROR_DESC** pointer	is a pointer to the newly allocated copy of the error descriptor that *src_err_desc* references.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_errmsg( ), mi_error_desc_destroy( ), mi_error_desc_is_copy( ), mi_error_level( ), mi_error_sql_state( ),** and **mi_error_sqlcode( ).**

For a general discussion of how to copy error descriptors, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_error_desc_destroy( )

The **mi_error_desc_destroy( )** function frees an error descriptor that **mi_error_desc_copy( )** allocated.

Syntax

```
mi_integer mi_error_desc_destroy(err_desc)
   MI_ERROR_DESC *err_desc;
```

err_desc	is a pointer to the error descriptor to free.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_error_desc_destroy( )** function is a destructor function for the error descriptor. It frees the error descriptor that *err_desc* references. The **mi_error_desc_copy( )** function must previously have allocated the *err_desc* error descriptor. This function can fail if either of the following error conditions exists:

- The *err_desc* structure is not a valid **MI_ERROR_DESC** structure.
- The *err_desc* structure is internally managed (it was not created by **mi_error_desc_copy( )**) and therefore cannot be freed by the user.

Use the **mi_error_desc_is_copy( )** function to determine how the *err_desc* structure was allocated.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_errmsg( ), mi_error_desc_copy( ), mi_error_desc_is_copy( ), mi_error_level( ), mi_error_sql_state( ),** and **mi_error_sqlcode( ).**

For a general discussion of how to release a copy of an error descriptor, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_error_desc_finish( )

The **mi_error_desc_finish( )** function completes processing of the current exception list.

Syntax

```
mi_integer mi_error_desc_finish(err_desc)
   MI_ERROR_DESC *err_desc;
```

err_desc	is a pointer to the current error descriptor in the list of current exceptions.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_error_desc_finish( )** function discards any further messages in the exception list of which *err_desc* is part. The exception list contains the error descriptors for situations that generate multiple exceptions. Normally, an exception callback is called once for each exception. However, if a situation generates multiple exceptions, you probably do not want the callback to be invoked for each exception. This function prevents the current callback function from being invoked for any more exceptions in the current exception list.

Return Values

0	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_errmsg( ), mi_error_desc_copy( ), mi_error_desc_destroy( ), mi_error_desc_is_copy( ), mi_error_desc_next( ), mi_error_level( ), mi_error_sql_state( ),** and **mi_error_sqlcode( ).**

mi_error_desc_is_copy( )

The **mi_error_desc_is_copy( )** function determines whether the specified error descriptor is a user copy.

Syntax

```
mi_integer mi_error_desc_is_copy(err_desc)
   MI_ERROR_DESC *err_desc;
```

err_desc	is a pointer to the error descriptor to examine.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_error_desc_is_copy( )** function determines whether the *err_desc* error descriptor is a user copy. A user copy of an error descriptor is one that the **mi_error_desc_copy( )** function has allocated. The **mi_error_desc_destroy( )** function returns an error if you attempt to deallocate a system-allocated error descriptor. Use this function to determine when to perform deallocation of an error descriptor with **mi_error_desc_destroy( ).**

An invalid error descriptor can cause the **mi_error_desc_is_copy( )** function to fail.

Return Values

MI_TRUE	indicates that the error descriptor that *err_desc* references is a user copy.
MI_FALSE	indicates that the error descriptor that *err_desc* references is not a user copy.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_errmsg( ), mi_error_desc_copy( ), mi_error_desc_destroy( ), mi_error_desc_finish( ), mi_error_desc_next( ), mi_error_level( ), mi_error_sql_state( ),** and **mi_error_sqlcode( ).**

For a general discussion of how to determine if an error descriptor is a copy, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_error_desc_next( )

The **mi_error_desc_next( )** function gets the next error descriptor from the list of current exceptions that are associated with the current SQL statement.

Syntax

```
MI_ERROR_DESC *mi_error_desc_next(err_desc)
   MI_ERROR_DESC *err_desc;
```

err_desc	is a pointer to the current error descriptor in the list of current exceptions. The **mi_error_desc_next( )** function returns the error descriptor that follows this *err_desc* descriptor.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_error_desc_next( )** returns the next error descriptor from the exception list of which *err_desc* is part. The exception list contains the error descriptors for situations that generate multiple exceptions. Normally, an exception callback is called once for each exception. However, if a situation generates multiple exceptions, you probably do not want the callback to be invoked for each exception. The callback can use the **mi_error_desc_next( )** function to obtain the details of an exception and to prevent the callback from being called again for that exception.

Return Values

An **MI_ERROR_DESC** pointer	is a pointer to the error descriptor for the next exception in the exception list.
NULL	indicates that the function was not successful or no more error descriptors exist.

Related Topics

See also the descriptions of **mi_errmsg( ), mi_error_desc_copy( ), mi_error_desc_destroy( ), mi_error_desc_finish( ), mi_error_desc_is_copy( ), mi_error_level( ), mi_error_sql_state( ),** and **mi_error_sqlcode( ).**

mi_error_level( )

The **mi_error_level( )** function retrieves from an error descriptor the exception level associated with an exception or an error level associated with a client LIBMI error.

Syntax

```
mi_integer mi_error_level(err_desc)
   MI_ERROR_DESC *err_desc;
```

err_desc	is a pointer to an error descriptor that describes an MI_Exception or MI_Client_Library_Error event.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_error_level( )** function returns an error level from the error descriptor that *err_desc* references. This error level can be either an exception level for a database server exception or an error level for a client LIBMI error:

- If *err_desc* references an MI_Exception event, **mi_error_level( )** returns either of the following exception levels for the database server exception.

Exception Level	Exception-Level Constant
Warning	MI_MESSAGE
Runtime error	MI_EXCEPTION

  This function is useful in an exception callback to get the exception level associated with an MI_Exception event (an SQL error or warning).
- If *err_desc* references an MI_Client_Library_Error event, **mi_error_level( )** returns one of the following error levels for the client LIBMI error.

Exception Level	Exception-Level Constant
Bad argument to DataBlade API function	MI_LIB_BADARG
Unable to connect to database server	MI_LIB_BADSERV
Lost connection to database server	MI_LIB_DROPCONN
Internal DataBlade API error	MI_LIB_INTERR
Feature or function not currently implemented	MI_LIB_NOIMP
DataBlade API function called out of sequence	MI_LIB_USAGE

  This function is useful in a client LIBMI callback to get the precise type of error associated with an MI_Client_Library_Error event.

Return Values

MI_LIB_BADARG MI_LIB_USAGE MI_LIB_INTERR MI_LIB_NOIMP MI_LIB_DROPCONN MI_LIB_BADSERV	Any of these values indicates that the event that the error descriptor describes is MI_Client_Library_Error. The return value indicates the cause of the client LIBMI error.
MI_MESSAGE MI_EXCEPTION	Either of these values indicates that the event that the error descriptor describes is MI_Exception. The return value indicates the type of exception: warning (MI_MESSAGE) or error (MI_EXCEPTION).
MI_ERROR	This value indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_errmsg( ), mi_error_desc_copy( ), mi_error_desc_destroy( ), mi_error_desc_finish( ), mi_error_desc_is_copy( ), mi_error_desc_next( ), mi_error_sql_state( ),** and **mi_error_sqlcode( ).**

For a general discussion on how to obtain information from an error descriptor, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_error_sql_state( )

The **mi_error_sql_state( )** function retrieves the value of the **SQLSTATE** status variable from an error descriptor.

Syntax

```
mi_integer mi_error_sql_state(err_desc, sqlstate_buf, buflen)
   MI_ERROR_DESC *err_desc;
   char *sqlstate_buf;
   mi_integer buflen;
```

err_desc	is a pointer to the error descriptor that describes the SQL error or warning.
sqlstate_buf	is a pointer to a buffer to contain the **SQLSTATE** value from the *err_desc* error descriptor.
buflen	is the allocated size of the *sqlstate_buf* buffer. It must be at least six bytes long; otherwise, **mi_error_sql_state( )** raises an exception.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_error_sql_state( )** function copies the value of the **SQLSTATE** status variable from the error descriptor that *err_desc* references into the user-allocated buffer that *sqlstate_buf* references. The **SQLSTATE** status value is a sequence of five ASCII characters with a null byte at the end. To indicate success, **SQLSTATE** contains a value of ″`00000`″. Other values indicate types of warnings and runtime errors. In particular, an **SQLSTATE** value of ″`IX000`" indicates an Informix-specific error, which the value of **SQLCODE** identifies. You can use the **mi_error_sqlcode( )** function to obtain the **SQLCODE** value from an error descriptor.

This function is intended for use with the **MI_ERROR_DESC** structure passed to a callback routine when the callback exception is of event type MI_Exception.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_errmsg( ), mi_error_desc_copy( ), mi_error_desc_destroy( ), mi_error_desc_finish( ), mi_error_desc_is_copy( ), mi_error_desc_next( ), mi_error_level( ),** and **mi_error_sqlcode( ).**

For general information about how to obtain information from an error descriptor, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_error_sqlcode( )

The **mi_error_sqlcode( )** function retrieves the value of the Informix **SQLCODE** status variable from an error descriptor.

Syntax

```
mi_integer mi_error_sqlcode(err_desc, sqlcode_ptr)
   MI_ERROR_DESC *err_desc;
   mi_integer *sqlcode_ptr;
```

err_desc	is a pointer to the error descriptor that describes the SQL error or warning.
sqlcode_ptr	is a pointer to the **SQLCODE** value that **mi_error_sqlcode( )** is to obtain from the *err_desc* error descriptor.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_error_sqlcode( )** function copies the value of the Informix **SQLCODE** status variable from the error descriptor that *err_desc* references into the user-allocated variable that *sqlcode_ptr* references. The **SQLCODE** variable contains an Informix-specific integer value that is `0` to indicate success and negative to indicate an error.

This function is intended for use with the **MI_ERROR_DESC** structure passed to a callback routine when the callback exception is of event type MI_Exception. You can use the **mi_error_sqlcode( )** function to retrieve an **SQLCODE** value to provide more information about cases in which **SQLSTATE** indicates an Informix-specific error ("`IX000`" value).

A separate, additional callback follows the callback for the **SQLCODE** value when an exception has an access-method error code (Informix RSAM status). This error code and message text are available through the **mi_errmsg( )** and **mi_error_sqlcode( )** functions.

If no **SQLCODE** value is defined, as for **mi_db_error_raise( )** exceptions, **mi_error_sqlcode( )** sets *sqlcode_ptr* to `0`.

Tip: The **SQLSTATE** variable is an ANSI-complaint way to indicate errors. The **SQLCODE** variable is specific to Informix. If your application is to be ANSI compliant, use **SQLSTATE** rather than **SQLCODE**.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_errmsg( ), mi_error_desc_copy( ), mi_error_desc_destroy( ), mi_error_desc_finish( ), mi_error_desc_is_copy( ), mi_error_desc_next( ), mi_error_level( ),** and **mi_error_sql_state( ).**

For general information about how to obtain information from an error descriptor, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_exec( )

The **mi_exec( )** function sends an SQL statement to the database server for parsing, optimization, and execution.

Syntax

```
mi_integer mi_exec(conn, stmt_strng, control)
   MI_CONNECTION *conn;
   const mi_string *stmt_strng;
   mi_integer control;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
stmt_strng	is a pointer to the statement string that contains the text of the SQL statement to execute.
control	is a bit mask of flags that determines the control mode for any results that the executed SQL statement returns. The valid control flags follow:

MI_QUERY_BINARY	The query results are in binary representation rather than text strings.
MI_QUERY_NORMAL	The query results are in text representation (null-terminated strings).

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_exec( )** function performs the following tasks:

- Sends the statement string to the database server for parsing, optimization, and execution
- If any rows are to be returned, creates and opens an implicit cursor, which is read only and sequential

  If the SQL statement is *not* a SELECT **statement, the statement is executed. If it is** SELECT**, mi_exec( ) automatically creates and opens a cursor for the retrieved rows**.

Tip: The **mi_exec( )** function does not handle execution of prepared statements. To send prepared statements to the database server, use **mi_exec_prepared_statement( )** or **mi_open_prepared_statement( ).**

The mi_exec( ) function only sends the statement; it does not return results to the DataBlade API module. To get results after the execution of **mi_exec( ),** the DataBlade API module needs to execute the **mi_get_result( )** function in a loop.

The current statement must finish before the database server can process the next statement. The **mi_query_finish( )** function can be used to force a statement to finish processing. It is strongly advised that you use either **mi_query_finish( )** or an **mi_get_result( )** loop after each **mi_exec( )** function.

Important: Do not use a handle returned by the **mi_get_session_connection( )** function in a call to the **mi_exec( )** function. You need to use **mi_lo_open( )** to obtain a handle.

For general information about how to send statements with **mi_exec( )**, see the *IBM Informix DataBlade API Programmer's Guide*.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

A successful return indicates *only* that the connection is valid and the statement was successfully executed (for statements other than SELECT) or a cursor was successfully opened (for SELECT). It does *not* indicate the success of the SQL statement. Use **mi_get_result( )** to determine the success of the SQL statement.

Related Topics

See also the descriptions of **mi_command_is_finished( ), mi_exec_prepared_statement( ), mi_get_result( ), mi_open_prepared_statement( ),** and **mi_query_finish( ).**

mi_exec_prepared_statement( )

The **mi_exec_prepared_statement( )** function sends a prepared statement to the database server for execution.

Syntax

```
mi_integer mi_exec_prepared_statement(stmt_desc, control,
   params_are_binary, n_params, values, lengths, nulls, types, retlen,
   rettypes)
   MI_STATEMENT *stmt_desc;
   mi_integer control;
   mi_integer params_are_binary;
   mi_integer n_params;
   MI_DATUM values[];
   mi_integer lengths[];
   mi_integer nulls[];
   mi_string *types[];
   mi_integer retlen;
   mi_string *rettypes[];
```

stmt_desc is a pointer to the statement descriptor for the prepared statement to execute. The **mi_prepare( )** function generates this statement descriptor.

control is a bit mask of flags that controls the behavior of **mi_exec_prepared_statement( )**. The valid control flags follow:

MI_BINARY	Returns results in binary representation.
0	Returns results in text representation.

params_are_binary is set to one of the following values:

1 (MI_TRUE)	The input parameters are passed in their internal (binary) representation.
0 (MI_FALSE)	The input parameters are passed in their external (text) representation.

n_params is the number of input parameters in the prepared SQL statement. It is also the number of entries in the input-parameter-value arrays: *values*, *lengths*, *nulls*, and *types*.

values is an array of **MI_DATUM** structures that contain the values of the input parameters in the prepared statement.

lengths is an array of the lengths (in bytes) of the input-parameter values.

nulls is an array that indicates whether each input parameter contains an SQL NULL value. Each array element is set to one of the following values:

1 (MI_TRUE)	The value of the associated input parameter *is* an SQL NULL value.
0 (MI_FALSE)	The value of the associated input parameter is *not* an SQL NULL value.

types is either an array of pointers to the names of the data types for the input parameters or a NULL-valued pointer.

retlen	is the length of the *rettypes* array. Valid *retlen* values follow:	
	>0	Indicates the number of columns that the query returns.
	0	Indicates that no result values exist.
rettypes	is either an array of pointers to the names of the data types to which the return columns are cast or, if result values do not need to be cast, a NULL-valued pointer.	

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_exec_prepared_statement( )** function performs the following tasks:

- Binds any input-parameter values to the input-parameter placeholders in the prepared SQL statement that *stmt_desc* references

  For any input parameter specified in the statement text of the SQL statement, you must initialize the *values*, *lengths*, and *nulls* arrays. If the prepared statement has input parameters and is *not* an INSERT statement, you must supply the data types of the parameters in the *types* array. You can provide the input-parameter values in either of the representations that correspond with the *params_are_binary* flag.

  For information on how to bind input-parameter values, see the *IBM Informix DataBlade API Programmer's Guide*.
- Sends the prepared statement to the database server for execution
- If the prepared statement is a query (that is, it returns rows), opens an implicit cursor

  The cursor is stored as part of the statement descriptor. Only one cursor per statement descriptor is current. An implicit cursor is a read-only sequential cursor. If you need some other kind of cursor to access the returned rows, use the **mi_open_prepared_statement( )** function to define an explicit cursor.

The mi_exec_prepared_statement( ) function only sends the statement to the database server; it does *not* return results to the application. To get results after the execution **mi_exec_prepared_statement( )**, the DataBlade API module must examine the results through a loop with **mi_get_result( ).** However, the *control* argument of **mi_exec_prepared_statement( )** does determine the control mode for the statement results.

The **mi_exec_prepared_statement( )** function allocates a type descriptor for each of the data types of the input parameters in the *types* array. 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. 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.

If the prepared statement is a SELECT statement, you can set the data types of the selected columns by setting a pointer to a type name for each returned column in the *rettypes* array. If the pointer is NULL, the type is not modified. It will either be the return type of the column or the type set by a previous **mi_exec_prepared_statement( )** call. You cannot set the return types of subcolumns of fields of a row type.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

A successful return indicates only that the connection is valid and the statement was successfully executed (for statements other than SELECT) or a cursor was successfully opened (for SELECT). It does *not* indicate the success of the SQL statement. Use **mi_get_result( )** to determine the success of the SQL statement.

Related Topics

See also the descriptions of **mi_drop_prepared_statement( ), mi_exec( ), mi_get_result( ), mi_open_prepared_statement( ),** and **mi_prepare( ).**

mi_fetch_statement( )

The **mi_fetch_statement( )** function fetches specified rows from the database server into a cursor that is associated with an opened prepared statement.

Syntax

```
mi_integer mi_fetch_statement(stmt_desc, cursor_action, jump, num_rows)
   MI_STATEMENT *stmt_desc;
   MI_CURSOR_ACTION cursor_action;
   mi_integer jump;
   mi_integer num_rows;
```

stmt_desc	is a pointer to the statement descriptor for the prepared statement that the **mi_open_prepared_statement( )** function has opened.
cursor_action	determines the orientation of the fetch. When a cursor opens, the current cursor position is before the first element. Possible values for *cursor_action* follow:

MI_CURSOR_NEXT	Fetches the next *num_rows* rows.
MI_CURSOR_PRIOR	Fetches the previous *num_rows* rows.
MI_CURSOR_FIRST	Fetches the first *num_rows* rows.
MI_CURSOR_LAST	Fetches the last *num_rows* rows.
MI_CURSOR_ABSOLUTE	Moves *jump* rows into the retrieved rows and fetches *num_rows* rows.
MI_CURSOR_RELATIVE	Moves *jump* rows from the current position in the retrieved rows and fetches *num_rows* rows.

jump	is the relative or absolute offset of the fetch.
num_rows	is the number of rows to fetch. Use zero (0) to fetch all rows.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fetch_statement( )** function fetches *num_rows* rows from the *cursor_action* orientation into an explicit cursor, which **mi_open_prepared_statement( )** has opened. The **mi_fetch_statement( )** function does *not* return any rows to the DataBlade API module but copies retrieved rows from the database server into the row cursor that is associated with the *stmt_desc* statement descriptor. To access a row, use the **mi_next_row( )** function, which retrieves the row from the cursor. After you access all rows in the cursor, **mi_next_row( )** returns the NULL-valued pointer and sets its *error* argument to MI_NO_MORE_RESULTS.

To specify the number of rows to fetch, use the *num_rows* argument.

num_rows Value	Description
zero (0)	**mi_fetch_statement( )** fetches *all* resulting rows into the cursor.
>0	**mi_fetch_statement( )** fetches only *num_rows* rows into the cursor.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful; the cursor could not be fetched or the statement is invalid.

Related Topics

See also the descriptions of **mi_get_result( ), mi_next_row( ), mi_open_prepared_statement( ), mi_prepare( ),** and **mi_result_row_count( ).**

mi_file_allocate( )

The **mi_file_allocate( )** function ensures that a specified number of files are available to be opened.

Syntax

```
mi_integer mi_file_allocate(num_files)
   mi_integer num_files;
```

num_files	specifies how many file descriptors to allocate.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_file_allocate( )** function allocates *num_files* number of file descriptors for use with operating-system calls, such as UNIX **open( )** or **fopen( )**.

Tip: The **mi_file_allocate( )** function is provided for compatibility with earlier versions only. This function is not required for file access in DataBlade API modules. In new code, use DataBlade API file-access functions such as **mi_file_open( )** and **mi_file_close( ).**

Server Only

This function does not perform any tasks when called within a UDR.

End of Server Only

Return Values

>=0	is the number of file descriptors that **mi_file_allocate( )** has allocated.
MI_ERROR	indicates that the function was not successful.

The **mi_file_allocate( )** function does *not* throw an MI_Exception event when it encounters a runtime error. It does not cause callbacks to be invoked.

mi_file_close( )

The **mi_file_close( )** function closes an operating-system file.

Syntax

```
void mi_file_close(fd)
   mi_integer fd;
```

fd	is the file descriptor of the file to close.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_file_close( )** function closes the operating-system file that the *fd* file descriptor references. This file descriptor must have been obtained from a previous call to the **mi_file_open( )** function. The **mi_file_close( )** function is a destructor function for an operating-system file descriptor. It frees the *fd* file descriptor references. Unless you explicitly close a file with **mi_file_close( )**, files remain open for the duration of the client session.

Server Only

In a C UDR, this function can close files on either the server or client computer. You specify the location of the file when you open it with **mi_file_open( ).**

End of Server Only

Return Values

None.

Related Topics

See also the description of **mi_file_open( ).**

mi_file_errno( )

The **mi_file_errno( )** function returns the value of the system **errno** variable after a file input/output (I/O) operation. This value is the last **errno** value generated during an **mi_file*** function call and comes from the computer where the file is located. This function does not translate the value from one platform to another.

Syntax

```
mi_integer mi_file_errno( )
```

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

File I/O functions of the C library call underlying system or platform file I/O functions. These system or platform functions set a global variable called **errno** to indicate status. For a list of DataBlade API file-access functions, see the *IBM Informix DataBlade API Programmer's Guide*.

These file-access functions call the corresponding C library functions to perform their tasks. Therefore, the value of the **errno** is available for a DataBlade API module.

UNIX/Linux Only

On UNIX or Linux, you can check the **errno** variable directly, immediately after a DataBlade API file I/O function. Therefore, use of **mi_file_errno( )** is not required for UNIX or Linux. In fact, the **mi_file_errno( )** function on UNIX or Linux is the same as the **errno** variable.

End of UNIX/Linux Only

Windows Only

If you plan to port a user-defined routine (UDR) or DataBlade module to Windows, it is *strongly* recommended that you use the **mi_file_errno( )** to retrieve the **errno** value.

On Windows, global variables, such as **errno**, are not easily accessible. Therefore, to obtain the errno value after a DataBlade API file I/O operation, use the **mi_file_errno( )** function. DataBlade APIs and UDRs that execute on the Windows platform *must* use **mi_file_errno( )** to access the **errno** for file operations.

End of Windows Only

Server Only

This function can access the **errno** value only for a file that resides on a server computer.

End of Server Only

Return Values

None.

Related Topics

See also the descriptions of **mi_file_allocate( ), mi_file_close( ), mi_file_open( ), mi_file_read( ), mi_file_seek( ), mi_file_sync( ), mi_file_tell( ),** and **mi_file_to_file( ).**

mi_file_open( )

The **mi_file_open( )** function opens an operating-system file.

Syntax

```
mi_integer mi_file_open(filename, open_flags, open_mode)
   const char *filename;
   mi_integer open_flags;
   mi_integer open_mode;
```

filename	is the pathname of the file to open.
open_flags	is a value bit mask that can have any of the following values:

open flags that the operating-system open command supports: UNIX or Linux **open(2)** or Windows **_open**.

MI_O_SERVER_FILE (*default***)**	indicates that the file to open is on the server computer.
MI_O_CLIENT_FILE	indicates that the file to open is on the client computer. When you set this flag, you also need to include the appropriate file-mode flag values, as described later in this section.

open_mode	is the file-permission mode in a format that the operating-system open command supports: UNIX or Linux **open(2)** or Windows **_open**.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_file_open( )** function opens the *filename* file in the access mode that *open_flags* specifies and the open mode that *open_mode* specifies. The function returns an integer file descriptor to this open file, through which you can access an operating-system file. The **mi_file_open( )** function is the constructor function for a file descriptor. The **mi_file_open( )** function allocates a new file descriptor for the duration of the client session.

The file ownership (owner and permissions) that *open_mode* specifies must be compatible with what *open_flags* specifies and what the operating-system open command supports.

Server Only

From a C UDR, **mi_file_open( )** can access files on the server computer. The function uses the user and group identifier of the session user to open the file. The function assumes the file ownership of the server environment.

Client Only

In a client LIBMI application, **mi_file_open( )** assumes the file ownership of the application user.

End of Client Only

You can include environment variables in the *filename* path with the following syntax:

```
$ENV_VAR
```

This environment variable must be set in the server environment; that is, it must be set *before* the database server starts.

The *open_flags* argument contains two pieces of information:

- Whether the file to open resides on the server or client computer

  By default, **mi_file_open( )** assumes that the file to open resides on the server computer. If the file you need to open is on the client computer, include the MI_O_CLIENT_FILE flag in the *open_flags* bit mask. The file owner is the client user and file permissions will be consistent with the client user's umask setting.
- Which flags to send to the underlying operating-system call that opens a file

For opening client files, the **mi_file_open( )** function passes the *open_flags* argument to the underlying operating-system call that opens a file. These flags are translated to appropriate operating-system flags on the client side. (Therefore, the *open_flags* values must match those that your operating-system call supports.)

The file-mode flag values for the *open_flags* argument indicate the access modes of the file.

Server Only

Valid values for server-side processing using the MI_O_CLIENT_FILE flag include the following file-mode constants. When MI_O_CLIENT_FILE is specified, you must include an MI_* flag.

File-Mode Constant	Purpose
MI_O_EXCL	Open the file only if *fname_spec* does not exist. Raise an exception if *fname_spec* does exist.
MI_O_TRUNC	Truncate the file, if it already exists.
MI_O_APPEND	Append to the file, if it already exists.
MI_O_RDONLY	Open the file in read-only mode (*from_open_mode* only).
MI_O_RDWR	Open the file in read/write mode.
MI_O_WRONLY	Open the file in write-only mode (*to_open_mode* only).
MI_O_BINARY	Process the data as binary data (*to_open_mode* only).
MI_O_TEXT	Process the data as text data (not binary, which is used if you do not specify MI_O_TEXT).

End of Server Only

Client Only

The default for the **mi_file_open( )** function is to open the file on the server. The file mode is read/write for all users. The file owner is the client user ID. Valid values for the *open_flags* argument for opening server files include the following file-mode constants that can be used in client LIBMI applications.

File-Mode Constant	Purpose
O_EXCL	Open the file only if *fname_spec* does not exist. Raise an exception if *fname_spec* does exist.
O_TRUNC	Truncate the file, if it already exists.
O_APPEND	Append to the file, if it already exists.
O_RDONLY	Open the file in read-only mode (*from_open_mode* only).
O_RDWR	Open the file in read/write mode.
O_WRONLY	Open the file in write-only mode (*to_open_mode* only).
O_CREAT	Create the file, if the file does not exist.

For a complete list of **open( )** system calls, consult the `man` pages (UNIX) for your computer's operating system.

End of Client Only

Return Values

>=0	is the file descriptor for the file that **mi_file_open( )** has opened.
MI_ERROR	indicates that the function was not successful.

The **mi_file_open( )** function does *not* throw an MI_Exception event when it encounters a runtime error. Therefore, it does not cause callbacks to be invoked.

Related Topics

See also the descriptions of **mi_file_allocate( )** and **mi_file_close( )** and the *IBM Informix DataBlade API Programmer's Guide*.

mi_file_read( )

The **mi_file_read( )** function reads a specified number of bytes from an open operating-system file into a buffer.

Syntax

```
mi_integer mi_file_read(fd, buf, nbytes)
   mi_integer fd;
   char *buf;
   mi_integer nbytes;
```

fd	is the file descriptor of the file from which to read the data. The file descriptor is obtained by a previous call to **mi_file_open( ).**
buf	is a pointer to a user-allocated character buffer to contain the data read from the file.
nbytes	is the maximum number of bytes to read into the *buf* character buffer.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_file_read( )** function reads *nbytes* bytes of data from the file that the *fd* file descriptor identifies. The read begins at the current file seek position for *fd*. You can use the **mi_file_tell( )** function to obtain the current seek position. The function reads this data into the user-allocated buffer that *buf* references.

Server Only

In a C UDR, this function can read from files on either the server or client computer. You specify the location of the file when you open it with **mi_file_open( ).**

End of Server Only

Return Values

>=0	is the actual number of bytes that the function has read from the open file to the *buf* character buffer.
MI_ERROR	indicates that the function was not successful.

The **mi_file_read( )** function does *not* throw an MI_Exception event when it encounters a runtime error. Therefore, it does not cause callbacks to be invoked.

Related Topics

See also the descriptions of **mi_file_open( ), mi_file_seek( ), mi_file_tell( ),** and **mi_file_write( ).**

mi_file_seek( )

The **mi_file_seek( )** function sets the file seek position for the next read or write operation on the open file.

Syntax

```
mi_integer mi_file_seek(fd, offset, whence)
   mi_integer fd;
   mi_integer offset;
   mi_integer whence;
```

fd	is the file descriptor for the operating-system file on which to set the seek position. The file descriptor is obtained by a previous call to **mi_file_open( ).**
offset	is a pointer to the integer offset from the specified *whence* seek position.
whence	determines how to interpret the *offset* value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_file_seek( )** function uses the *whence* and *offset* arguments to determine the new seek position of the file that *fd* identifies, as follows:

- The *whence* argument identifies the position from which to start the seek operation.

  Valid values include the following *whence* constants for both client and server files.

Whence Constant	Starting Seek Position
SEEK_SET	Set position equal to offset bytes
SEEK_CUR	Set position to current location plus offset
SEEK_END	Set position to EOF plus offset

- The *offset* argument identifies the offset, in bytes, from the starting seek position (which the *whence* argument specifies) at which to begin the seek.

  This *offset* value can be negative for all values of *whence*. To obtain the current seek position for an open smart large object, use the **mi_file_tell( )** function.

Server Only

In a C UDR, this function can move to a new seek position in a file that resides on either the server or client computer. You specify the location of the file when you open it with **mi_file_open( ).**

End of Server Only

Return Values

>=0	is the new seek position, measured in number of bytes from the beginning of the file.
MI_ERROR	indicates that the function was not successful.

The **mi_file_seek( )** function does *not* throw an MI_Exception event when it encounters a runtime error. Therefore, it does not cause callbacks to be invoked.

Related Topics

See also the descriptions of **mi_file_open( ), mi_file_read( ), mi_file_tell( ),** and **mi_file_write( ).**

mi_file_sync( )

The **mi_file_sync( )** function forces a write to disk of all pages in an operating-system file.

Syntax

```
mi_integer mi_file_sync(fd)
   mi_integer fd;
```

fd	is the file descriptor of the file to be written to disk. The file descriptor is obtained by a previous call to **mi_file_open( ).**

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_file_sync( )** function writes to disk all pages of the operating-system file that the *fd* file descriptor identifies.

Server Only

This function can force a write only for a file that resides on the server computer.

End of Server Only

Windows Only

The **mi_file_sync( )** interface routine has no effect on Windows clients.

End of Windows Only

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was not successful.

The **mi_file_sync( )** function does *not* throw an MI_Exception event when it encounters a runtime error. Therefore, it does not cause callbacks to be invoked.

Related Topics

See also the description of **mi_file_open( ).**

mi_file_tell( )

The **mi_file_tell( )** function returns the current file seek position for an operating-system file, relative to the beginning of the file.

Syntax

```
mi_integer mi_file_tell(fd)
   mi_integer fd;
```

fd	is the file descriptor of the file whose seek position is requested. The file descriptor is obtained by a previous call to **mi_file_open( ).**

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_file_tell( )** function obtains the seek position for the operating-system file that *fd* identifies. The *file seek position* is the offset for the next read or write operation on the file that is associated with the file descriptor, *fd*.

Server Only

In a C UDR, this function can obtain the seek position of a file that resides on either the server or client computer. You specify the location of the file when you open it with **mi_file_open( ).**

End of Server Only

Return Values

>=0	is the current seek position, measured in number of bytes from the beginning of the file.
MI_ERROR	indicates that the function was not successful.

The **mi_file_tell( )** function does *not* throw an MI_Exception event when it encounters a runtime error. Therefore, it does not cause callbacks to be invoked.

Related Topics

See also the descriptions of **mi_file_read( ), mi_file_seek( ),** and **mi_file_write( ).**

mi_file_to_file( )

The **mi_file_to_file( )** function copies files between the database server and a client computer.

Syntax

```
char *mi_file_to_file(conn, fromfile, from_open_mode, tofile, to_open_mode)
   MI_CONNECTION *conn;
   const char *fromfile;
   mi_integer open_mode;
   const char *tofile;
   mi_integer toflags;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( )** or **mi_server_connect( ).**
fromfile	is the full pathname of the source file.
from_open_mode	is a bit-mask argument to indicate how to open the *fromfile* file and the location of this file. For a list of valid file-mode constants, see the table in the following ″Usage″ section.
tofile	is the full pathname of the destination file.
to_open_mode	is a bit-mask argument to indicate how to open the *tofile* file and the location of this file. For a list of valid file-mode constants, see the table in the following ″Usage″ section.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

You can use the **mi_file_to_file( )** function to copy a source file to a target file:
- From a server computer to a client computer
- From a client computer to a server computer
- From the server computer to the server computer

The **mi_file_to_file( )** function does not support copies from client computer to client computer.

You can include environment variables in the *fromfile* and *tofile* paths with the following syntax:

```
$ENV_VAR
```

These environment variables must be set in the server environment; that is, they must be set *before* the database server starts.

The **mi_file_to_file( )** function can create the target files on either the server computer or the client computer. The file-mode flag values for the *from_open_mode* and *to_open_mode* arguments indicate the access modes and locations of the source and target files. Valid values include the following file-mode constants.

File-Mode Constant	Purpose
MI_O_EXCL	Open the file only if *fname_spec* does not exist. Raise an exception if *fname_spec* does exist.

MI_O_TRUNC	Truncate the file, if it already exists.
MI_O_APPEND	Append to the file, if it already exists.
MI_O_RDONLY	Open the file in read-only mode (*from_open_mode* only).
MI_O_RDWR	Open the file in read/write mode.
MI_O_WRONLY	Open the file in write-only mode (*to_open_mode* only).
MI_O_BINARY	Process the data as binary data (*to_open_mode* only).
MI_O_TEXT	Process the data as text data (not binary, which is used if you do not specify MI_O_TEXT).
MI_O_SERVER_FILE	The *fname_spec* file is created on the server computer. The file mode is read/write for all users. The file owner is the client user ID.
MI_O_CLIENT_FILE	The *fname_spec* file is created on the client computer. The file owner is the client user and file permissions will be consistent with the client user's umask setting.

The default *to_open_mode* value follows:

```
MI_O_CLIENT_FILE | MI_O_WRONLY | MI_O_TRUNC
```

The *from_open_mode* and *to_open_mode* must include either MI_O_CLIENT_FILE or MI_O_SERVER_FILE, but not both. Because **mi_file_to_file( )** does not support copies from client computer to client computer, *from_open_mode* and *to_open_mode* cannot both be set to MI_O_CLIENT_FILE.

Return Values

A **char** pointer	is a pointer to the pathname of the destination file.
NULL	indicates that the function was not successful.

The **mi_file_to_file( )** function does *not* throw an MI_Exception event when it encounters a runtime error. Therefore, it does not cause callbacks to be invoked.

Related Topics

See also the descriptions of **mi_lo_filename( ), mi_lo_from_file( ), mi_lo_from_file_by_lofd( ),** and **mi_lo_to_file( ).**

mi_file_unlink( )

The **mi_file_unlink( )** function unlinks (removes) a file that **mi_file_open( )** previously opened.

Syntax

```
mi_integer mi_file_unlink(fd)
   mi_integer fd;
```

fd	is the file descriptor of the file to remove. The file descriptor is obtained by a previous call to **mi_file_open( ).**

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_file_unlink( )** function removes the operating-system file that *fd* identifies. This function is a destructor function for an operating-system file descriptor.

Client Only

A call to **mi_file_unlink( )** from a client LIBMI application raises an exception.

End of Client Only

Server Only

This function can only unlink a file that resides on the server computer.

End of Server Only

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

The **mi_file_unlink( )** function does *not* throw an MI_Exception event when it encounters a runtime error. Therefore, it does not cause callbacks to be invoked.

Related Topics

See also the description of **mi_file_open( ).**

mi_file_write( )

The **mi_file_write( )** function writes *a specified number of* bytes to an open operating-system file.

Syntax

```
mi_integer mi_file_write(fd, buf, nbytes)
   mi_integer fd;
   const char *buf;
   mi_integer nbytes;
```

fd	is the file descriptor of the file to write to. The file descriptor is obtained by a previous call to **mi_file_open( ).**
buf	is a pointer to a user-allocated character buffer of at least *nbytes* bytes that contains the data to write to the file.
nbytes	is the maximum number of bytes to write to the file.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

The **mi_file_write( )** function writes *nbytes* bytes of data to the file that the *fd* file descriptor identifies. The write begins at the current file seek position for *fd*. You can use the **mi_file_tell( )** function to obtain the current seek position. The function writes this data from the user-allocated buffer that *buf* references.

Server Only

In a C UDR, this function can write to files on either the server or client computer. You specify the location of the file when you open it with **mi_file_open( ).**

End of Server Only

Return Values

>=0	is the actual number of bytes that the function has written from the *h* to the open file.
MI_ERROR	indicates that the function was not successful.

The **mi_file_write( )** function does *not* throw an MI_Exception event when it encounters a runtime error. Therefore, it does not cause callbacks to be invoked.

Related Topics

See also the descriptions of **mi_file_open( ), mi_file_read( ), mi_file_seek( ),** and **mi_file_tell( ).**

mi_fix_integer( )

The **mi_fix_integer( )** function converts the specified 4-byte integer to or from the byte order of the client computer.

Syntax

```
mi_unsigned_integer mi_fix_integer(val)
   mi_unsigned_integer val;
```

val is the 4-byte integer value on which to fix the byte order.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_fix_integer( )** function is useful in send and receive support functions for an opaque data type when the opaque-type structure contains an **mi_integer** component that needs to be converted to and from the client byte order.

Return Values

An **mi_unsigned_integer** value
is the value in the desired byte order.

MI_ERROR indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_fix_smallint( ), mi_get_bytes( ), mi_get_double_precision( ), mi_get_integer( ), mi_get_smallint( ), mi_put_bytes( ), mi_put_double_precision( ), mi_put_integer( ),** and **mi_put_smallint( )**.

mi_fix_smallint( )

The **mi_fix_smallint( )** function converts the specified 2-byte integer to or from the byte order of the client computer.

Syntax

```
mi_unsigned_integer mi_fix_smallint (val)
   mi_unsigned_integer val;
```

val	is the 2-byte integer on which to fix the byte order.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_fix_smallint( )** function is useful in send and receive support functions for an opaque data type when the opaque type contains **mi_smallint** components that need to be converted to and from the client byte order.

For maximum portability, this function accepts and returns fully promoted **mi_integer** values instead of **mi_smallint** values. Arguments and return values might therefore require casting.

Return Values

An **mi_unsigned_integer** value	is the value in the desired byte order.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_fix_integer( ), mi_get_bytes( ), mi_get_double_precision( ), mi_get_integer( ), mi_get_smallint( ), mi_put_bytes( ), mi_put_double_precision( ), mi_put_integer( ),** and **mi_put_smallint( )**.

mi_fp_argisnull( )

The **mi_fp_argisnull( )** accessor function determines whether the argument of a user-defined routine is an SQL NULL value from its associated **MI_FPARAM** structure.

Syntax

```
mi_unsigned_char1 mi_fp_argisnull(fparam_ptr, arg_pos)
   MI_FPARAM *fparam_ptr;
   mi_integer arg_pos;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
arg_pos	is the index position into the null-argument array for the argument to check for a NULL value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fp_argisnull( )** function determines whether the routine argument at position *arg_pos* in the **MI_FPARAM** structure that *fparam_ptr* references contains the SQL NULL value. The **MI_FPARAM** structure stores information about whether routine arguments contain the NULL value in the zero-based null-argument array. Therefore, to obtain information about the *n*th argument, use an *arg_pos* value of *n*-1. For example, the following call to **mi_fp_argisnull( )** determines whether the third argument of the **my_func( )** UDR is NULL:

```
mi_integer my_func(..., fparam1)
   ...
   MI_FPARAM *fparam1;
{
   ...
   if ( mi_fp_argisnull(fparam1, 2) == MI_TRUE )
      /* code to handle NULL argument */
```

Routines that handle NULL arguments must be registered with the HANDLESNULLS routine modifier of the CREATE FUNCTION or CREATE PROCEDURE statement.

Return Values

MI_TRUE	indicates that the argument at position *arg_pos* is NULL.
MI_FALSE	indicates that the argument at position *arg_pos* is not NULL.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_fp_arglen( ), mi_fp_argprec( ), mi_fp_argscale( ), mi_fp_argtype( ), mi_fp_returnisnull( ), mi_fp_setargisnull( ),** and **mi_fp_setreturnisnull( ).**

For more information about argument information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_arglen( )

The **mi_fp_arglen( )** accessor function obtains the length of an argument of a user-defined routine from its associated **MI_FPARAM** structure.

Syntax

```
mi_integer mi_fp_arglen(fparam_ptr, arg_pos)
   MI_FPARAM *fparam_ptr;
   mi_integer arg_pos;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
arg_pos	is the index position into the argument-length array for the argument whose length you want.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fp_arglen( )** function obtains the length of the routine argument at position *arg_pos* from the **MI_FPARAM** structure that *fparam_ptr* references. The **MI_FPARAM** structure stores information about the lengths of routine arguments in the zero-based argument-length array. To obtain information about the *n*th argument, use an *arg_pos* value of *n*-1. For example, the following call to **mi_fp_arglen( )** obtains the length for the third argument of the **my_func( )** UDR, with which **fparam1** is associated:

```
mi_integer my_func(..., fparam1)
   ...
   MI_FPARAM *fparam1;
{
   mi_integer arg_len;
   ...
   arg_len = mi_fp_arglen(fparam1, 2);
```

Return Values

>=0	is the length, in bytes, of the argument at position *arg_pos*.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_fp_argisnull( ), mi_fp_argprec( ), mi_fp_argscale( ), mi_fp_argtype( ), mi_fp_retlen( ), mi_fp_setarglen( ),** and **mi_fp_setretlen( ).**

For more information about argument information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_argprec( )

The **mi_fp_argprec( )** accessor function obtains the precision of an argument of a user-defined routine from its associated **MI_FPARAM** structure.

Syntax

```
mi_integer mi_fp_argprec(fparam_ptr, arg_pos)
   MI_FPARAM *fparam_ptr;
   mi_integer arg_pos;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
arg_pos	is the index position into the argument-precision array for the argument whose precision you want.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fp_argprec( )** function obtains the precision of the routine argument at position *arg_pos* from the **MI_FPARAM** structure that *fparam_ptr* references. The precision is an attribute of the data type that represents the total number of digits the routine argument can hold, as follows.

Data Type	Meaning
DECIMAL, MONEY	Number of significant digits in the fixed-point or floating-point (DECIMAL) column
DATETIME, INTERVAL	Number of digits that are stored in the date and/or time column with the specified qualifier
Character, Varying-character	Maximum number of characters in the column

If you call **mi_fp_argprec( )** on some other data type, the function returns zero (`0`).

The **MI_FPARAM** structure stores information about the precision of routine arguments in the zero-based argument-precision array. To obtain information about the *n*th argument, use an *arg_pos* value of *n*-1. For example, the following call to **mi_fp_argprec( )** obtains the precision for the third argument of the **my_func( )** UDR, with which **fparam1** is associated:

```
mi_integer my_func(..., fparam1)
   ...
   MI_FPARAM *fparam1;
{
   mi_integer arg_prec;
   ...
   arg_prec = mi_fp_argprec(fparam1, 2);
```

Return Values

>=0	is the precision, in number of digits, of the fixed-point or floating-point argument at position *arg_pos*.

MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_fp_argisnull( ), mi_fp_arglen( ), mi_fp_argscale( ), mi_fp_argtype( ), mi_fp_retprec( ), mi_fp_setargprec( ),** and **mi_fp_setretprec( ).**

For more information about argument information in an **MI_FPARAM** structure or about the precision of a fixed-point data type, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_argscale( )

The **mi_fp_argscale( )** accessor function obtains the scale of an argument of a user-defined routine from its associated **MI_FPARAM** structure.

Syntax

```
mi_integer mi_fp_argscale(fparam_ptr, arg_pos)
   MI_FPARAM *fparam_ptr;
   mi_integer arg_pos;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
arg_pos	is the index position into the argument-scale array for the argument whose scale you want.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fp_argscale( )** function obtains the scale of the routine argument at position *arg_pos* from the **MI_FPARAM** structure that *fparam_ptr* references. The scale is an attribute of the data type. The meaning of the scale depends on the associated data type, as the following table shows.

Data Type	Meaning of Scale
DECIMAL (fixed-point), MONEY	The number of digits to the right of the decimal point
DECIMAL (floating-point)	The value 255
DATETIME, INTERVAL	The encoded integer value for the end qualifier of the data type; *end_qual* in the qualifier: *start_qual* TO *end_qual*

If you call **mi_fp_argscale( )** on some other data type, the function returns zero (`0`).

The **MI_FPARAM** structure stores information about the scale of routine arguments in the zero-based argument-scale array. To obtain information about the *n*th argument, use an *arg_pos* value of *n*-1.

For example, the following call to **mi_fp_argscale( )** obtains the scale for the third argument of the **my_func( )** UDR, with which **fparam1** is associated:

```
mi_integer my_func(..., fparam1)
   ...
   MI_FPARAM *fparam1;
{
   mi_integer arg_scale;
   ...
   arg_scale = mi_fp_argscale(fparam1, 2);
```

Return Values

>=0	is the scale, in number of digits, of the fixed-point or floating-point argument at position *arg_pos*.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_fp_argisnull( ), mi_fp_arglen( ), mi_fp_argprec( ), mi_fp_argtype( ), mi_fp_retscale( ), mi_fp_setargscale( ),** and **mi_fp_setretscale( ).**

For more information about argument information in an **MI_FPARAM** structure or about the precision of a fixed-point data type, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_argtype( )

The **mi_fp_argtype( )** accessor function obtains the type identifier for the data type of an argument of a user-defined routine from the argument's associated **MI_FPARAM** structure.

Syntax

```
MI_TYPEID *mi_fp_argtype(fparam_ptr, arg_pos)
   MI_FPARAM *fparam_ptr;
   mi_integer arg_pos;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
arg_pos	is the index position into the argument-type array for the argument whose type identifier you want.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

The **mi_fp_argtype( )** function obtains the type identifier (MI_TYPEID) of the routine argument at position *arg_pos* from the **MI_FPARAM** structure that *fparam_ptr* references. The type identifier is an integer value that indicates a particular data type. The **MI_FPARAM** structure stores information about the type identifiers of routine arguments in the zero-based argument-type array. To obtain information about the *n*th argument, use an *arg_pos* value of *n*-1. For example, the following call to **mi_fp_argtype( )** obtains the type identifier for the third argument of the **my_func( )** UDR, with which **fparam1** is associated:

```
mi_integer my_func(..., fparam1)
   ...
   MI_FPARAM *fparam1;
{
   MI_TYPEID *arg_type;
   ...
   arg_type = mi_fp_argtype(fparam1, 2);
```

Return Values

An **MI_TYPEID** pointer	is a pointer to the type identifier of the argument at position *arg_pos*.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_fp_argisnull( ), mi_fp_arglen( ), mi_fp_argprec( ), mi_fp_argscale( ), mi_fp_rettype( ), mi_fp_setargtype( ),** and **mi_fp_setrettype( ).**

For more information about argument information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_funcname( )

The **mi_fp_funcname( )** function obtains the name of a user-defined routine (UDR) using its associated **MI_FPARAM** structure.

Syntax

```
mi_string *mi_fp_funcname (fparam_ptr)
   MI_FPARAM *fparam_ptr;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
No	Yes

Usage

The **mi_fp_funcname( )** function obtains the routine name for the UDR that is associated with the *fparam_ptr* **MI_FPARAM** structure. The routine name is the SQL name of the UDR, stored in the **procname** column of the **sysprocedures** system catalog table. It is not the name of the C function that implements it. The function allocates memory (in the current memory duration) for the copy of the routine name that it returns.

This function is useful for UDRs that need to determine the routine name at runtime in an efficient way.

Return Values

An **mi_string** pointer	is a pointer to a string that contains the name of the user-defined routine.
NULL	indicates that the function was not successful.

Related Topics

For more information about UDR information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_funcstate( )

The **mi_fp_funcstate( )** accessor function obtains user-state information for the user-defined routine from its associated **MI_FPARAM** structure.

Syntax

```
void *mi_fp_funcstate(fparam_ptr)
   MI_FPARAM *fparam_ptr;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

The **mi_fp_funcstate( )** function obtains a pointer to the user-state information for the user-defined routine that is associated with the *fparam_ptr* **MI_FPARAM** structure. In the first invocation of the UDR, the database server sets the user-state pointer to NULL. Use the **mi_fp_funcstate( )** function to return the user-state pointer so that you can access the state information within a UDR.

Cast this user-state pointer to match the structure of the user-defined buffer. For example, the following call to **mi_fp_funcstate( )** casts the user-state pointer as a structure called **udr_info** and uses this pointer to access the **count_fld** of the **udr_info** structure:

```
MI_FPARAM *my_fparam;
struct udr_info *fi_ptr;
mi_integer count;
...
fi_ptr = (udr_info *)mi_fp_funcstate( my_fparam );
count = fi_ptr->count_fld;
```

Return Values

A user-state pointer	is a pointer that references the user-state information in a user-defined buffer. Cast this pointer to match the structure of the user-state information.
NULL	indicates that the user-state pointer is uninitialized. The user-state pointer has a NULL value the first time a UDR is invoked.

Related Topics

See also the descriptions of **mi_fp_argisnull( ), mi_fp_arglen( ), mi_fp_argprec( ), mi_fp_argscale( ), mi_fp_argtype( ), mi_fp_retlen( ), mi_fp_retprec( ), mi_fp_retscale( ), mi_fp_rettype( ), mi_fp_returnisnull( ),** and **mi_fp_setfuncstate( ).**

For more information about UDR information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_getcolid( )

The **mi_fp_getcolid( )** accessor function obtains the column identifier of the column that is associated with the user-defined routine from its **MI_FPARAM** structure.

Syntax

```
mi_integer mi_fp_getcolid(fparam_ptr)
   MI_FPARAM *fparam_ptr;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_fp_getcolid( )** function obtains the column identifier for the user-defined routine that is associated with the *fparam_ptr* **MI_FPARAM** structure. The column identifier is the location of the column within the row structure (with the first column starting at offset 0). The column identifier and row structure identify the column with which the UDR invocation is associated. To obtain the row structure, use the **mi_fp_getrow( )** function.

This function is *only* valid within an **assign( )**, **destroy( )**, or import support function for an opaque data type that contains smart large objects and for multirepresentational opaque types. Before executing the **assign( )**, **destroy( )**, or import function of an opaque data type, the database server automatically obtains the column identifier and row structure and stores them in the **MI_FPARAM** structure.

With the **mi_fp_getcolid( )** function, you can implement delayed creation or removal of a smart large object:

- Delayed creation of a smart large object within the **assign( )** support function

  This function returns the column identifier for the column into which you want to store the opaque type.
- Delayed removal of a smart large object within the **destroy( )** support function

  This function obtains the column identifier for the column from which you want to remove the opaque type.

Important: The **mi_fp_getcolid( )** function is valid only when called from within an **assign( )**, **destroy( )**, or import support function of an opaque data type. Outside the context of an **assign( )** or **destroy( )** function, **mi_fp_getcolid( )** always returns MI_ERROR. For more information, see the description of the **mi_lo_colinfo_by_ids( )** function.

Return Values

>=0	is the column identifier of the column with which the UDR is associated.
MI_ERROR	indicates that the function was not successful or that it was *not* called from within an **assign( )**, **destroy( )**, or import support function of an opaque data type.

Related Topics

See also the descriptions of **mi_fp_getrow( ), mi_fp_setcolid( ),** and **mi_lo_colinfo_by_ids( ).**

For more information about UDR information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_getfuncid( )

The **mi_fp_getfuncid( )** accessor function obtains the routine identifier for a user-defined routine (UDR) in its associated **MI_FPARAM** structure.

Syntax

```
mi_funcid mi_fp_getfuncid(fparam_ptr)
   MI_FPARAM *fparam_ptr;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
No	Yes

Usage

The **mi_fp_getfuncid( )** function obtains the routine identifier for the user-defined routine that is associated with the *fparam_ptr* **MI_FPARAM** structure. The routine identifier uniquely identifies the UDR within the database. This function is useful to determine the name of the UDR that is currently executing or that is about to be called.

Return Values

>=0	is the routine identifier of the UDR associated with the specified **MI_FPARAM** structure.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the description of **mi_fp_setfuncid( ).**

For more information about UDR information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_getrow( )

The **mi_fp_getrow( )** accessor function obtains the row structure that is associated with the user-defined routine from its **MI_FPARAM** structure.

Syntax

```
MI_ROW *mi_fp_getrow(fparam_ptr)
    MI_FPARAM *fparam_ptr;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_fp_getrow( )** function obtains the row structure for the user-defined routine that is associated with the *fparam_ptr* **MI_FPARAM** structure. The row structure contains the column values of the row with which the UDR invocation is associated. The row structure and column identifier identify the column with which the UDR invocation is associated. To get the column identifier of the column within the row structure, use the **mi_fp_getcolid( )** function.

This function is valid *only* within an **assign( )**, **destroy( )**, or import support function for an opaque data type that contains smart large objects and for multirepresentational opaque types. Before executing the **assign( )**, **destroy( )**, or import function of an opaque data type, the database server automatically obtains the row structure and column identifier and stores them in the **MI_FPARAM** structure.

With the **mi_fp_getrow( )** function, you can implement delayed creation or removal of a smart large object:

- Delayed creation of a smart large object within the **assign( )** support function

  This function can obtain the row structure into which you want to store the opaque type.
- Delayed removal of a smart large object within the **destroy( )** support function

  This function can obtain the row structure from which you want to remove the opaque type.

Important: The **mi_fp_getrow( )** function is valid only when called from within an **assign( )**, **destroy( )**, or import support function of an opaque data type. Outside the context of an **assign( )** or **destroy( )** function, **mi_fp_getrow( )** always returns a NULL-valued pointer. For more information, see the description of the **mi_lo_colinfo_by_ids( )** function.

Return Values

An **MI_ROW** pointer	is a pointer to the row that is associated with the UDR.
NULL	indicates that the function was not successful or that it was *not* called from within an **assign( )**, **destroy( )**, or import support function of an opaque data type.

Related Topics

See also the descriptions of **mi_fp_getcolid( ), mi_fp_setrow( ),** and **mi_lo_colinfo_by_ids( ).**

For more information about UDR information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_nargs( )

The **mi_fp_nargs( )** accessor function obtains the number of arguments for the UDR routine from its associated **MI_FPARAM** structure.

Syntax

```
mi_integer mi_fp_nargs(fparam_ptr)
   MI_FPARAM *fparam_ptr;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

You can use the **mi_fp_nargs( )** function with the DataBlade API functions that obtain information about each argument of the UDR routine (such as **mi_fp_argtype( )** and **mi_fp_argisnull( )**). For example, the following call to **mi_fp_nargs( )** obtains the number of arguments from the **MI_FPARAM** structure that **fparam1** identifies and uses the value in a loop to obtain the length of each argument:

```
arg_count = mi_fp_nargs(fparam1);
for (i = 0; i < arg_count; i++)
   {
   arg_len[i] = mi_fp_arglen(fparam1, i);
```

Return Values

>=0	is the number of arguments with which the UDR was called.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_fp_argisnull( ), mi_fp_arglen( ), mi_fp_argprec( ), mi_fp_argscale( ), mi_fp_argtype( ), mi_fp_nrets( ),** and **mi_fp_setnargs( ).**

For more information about argument information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_nrets( )

The **mi_fp_nrets( )** accessor function obtains the number of return values for the UDR from its associated **MI_FPARAM** structure.

Syntax

```
mi_integer mi_fp_nrets(fparam_ptr)
   MI_FPARAM *fparam_ptr;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

You can use the **mi_fp_nrets( )** function with the DataBlade API functions that obtain information about each return value of the UDR (such as **mi_fp_rettype( )** and **mi_fp_returnisnull( )**). For example, the following call to **mi_fp_nrets( )** obtains the number of return values from the **MI_FPARAM** structure that **fparam1** identifies and uses the value in a loop to obtain the length of each return value:

```
ret_count = mi_fp_nrets(fparam1);
for (i = 0; i < ret_count; i++)
   {
   ret_len[i] = mi_fp_retlen(fparam1, i);
   ...
```

Important: C user-defined functions have only one return value.

Return Values

>=0	is the number of values that the UDR returns.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_fp_nargs( ), mi_fp_retlen( ), mi_fp_retprec( ), mi_fp_retscale( ), mi_fp_rettype( ), mi_fp_returnisnull( ),** and **mi_fp_setnrets( ).**

For more information about return-value information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_request( )

The **mi_fp_request( )** accessor function obtains the iterator status for an iterator function from an associated **MI_FPARAM** structure.

Syntax

```
MI_SETREQUEST mi_fp_request(fparam_ptr)
   MI_FPARAM *fparam_ptr;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The database server sets the iterator-status field in an **MI_FPARAM** structure when the associated user-defined routine is an iterator function. The iterator status is one of three possible **MI_SETREQUEST** values.

Iterator-Status Constant	Meaning	Use
SET_INIT	This is the *first* time that the iterator function is called.	Initialize the user state for the iterator function.
SET_RETONE	This is an actual iteration of the iterator function.	Return items of the active set, one per iteration.
SET_END	This is the last time that the iterator function is called.	Free any resources associated with the user state.

Use the **mi_fp_request( )** function in an iterator function to determine which of the preceding actions to perform for a given iteration.

Return Values

An MI_SETREQUEST constant	is the iterator-status constant of SET_INIT, SET_RETONE, or SET_END to indicate the current iterator status of the iteration function.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the description of **mi_fp_setisdone( ).**

For more information on how to create and call iterator functions, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_retlen( )

The **mi_fp_retlen( )** accessor function obtains the length of a return value of a user-defined function from its associated **MI_FPARAM** structure.

Syntax

```
mi_integer mi_fp_retlen(fparam_ptr, ret_pos)
   MI_FPARAM *fparam_ptr;
   mi_integer ret_pos;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
ret_pos	is the index position into the return-length array for the return value whose length you want. For user-defined functions, the only valid value is `0`.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fp_retlen( )** function obtains the length of the routine return value at position *ret_pos* from the **MI_FPARAM** structure that *fparam_ptr* references. The **MI_FPARAM** structure stores information about return-value lengths in the zero-based return-length array. To obtain information about the *n*th return value, use a *ret_pos* value of *n*-1. For example, the following call to **mi_fp_retlen( )** obtains the length for the first return value of the **my_func( )** user-defined function, with which **fparam1** is associated:

```
mi_integer my_func(..., fparam1)
   ...
   MI_FPARAM *fparam1;
{
   mi_integer ret_len;
   ...
   ret_len = mi_fp_retlen(fparam1, 0);
```

Important: C user-defined functions always have only one return value.

Return Values

>=0	is the length, in bytes, of the return value at position *ret_pos*.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_fp_arglen( ), mi_fp_retprec( ), mi_fp_retscale( ), mi_fp_rettype( ), mi_fp_returnisnull( ), mi_fp_setarglen( ),** and **mi_fp_setretlen( ).**

For more information about return-value information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_retprec( )

The **mi_fp_retprec( )** accessor function obtains the precision of a return value of a user-defined function from its associated **MI_FPARAM** structure.

Syntax

```
mi_integer mi_fp_retprec(fparam_ptr, ret_pos)
   MI_FPARAM *fparam_ptr;
   mi_integer ret_pos;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
ret_pos	is the index position into the return-precision array for the return value whose precision you want. For user-defined functions, the only valid value is 0.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fp_retprec( )** function obtains the precision of the routine return value at position *ret_pos* from the **MI_FPARAM** structure that *fparam_ptr* references.

The precision is an attribute of the data type that represents the total number of digits the routine return value can hold, as follows.

Data Type	Meaning
DECIMAL, MONEY	Number of significant digits in the fixed-point or floating-point (DECIMAL) column
DATETIME, INTERVAL	Number of digits that are stored in the date and/or time column with the specified qualifier
Character, Varying-character	Maximum number of characters in the column

If you call **mi_fp_retprec( )** on some other data type, the function returns zero (0).

The **MI_FPARAM** structure stores information about the precision of function return values in the zero-based return-precision array. To obtain information about the *n*th return value, use a *ret_pos* value of *n*-1. For example, the following call to **mi_fp_retprec( )** obtains the precision for the first return value of the **my_func( )** user-defined function, with which **fparam1** is associated:

```
mi_integer my_func(..., fparam1)
   ...
   MI_FPARAM *fparam1;
{
   mi_integer ret_prec;
   ...
   ret_prec = mi_fp_retprec(fparam1, 0);
```

Important: C user-defined functions always have only one return value.

Return Values

>=0	is the precision, in number of digits, of the fixed-point or floating-point return value at position *ret_pos*.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_fp_argprec( ), mi_fp_retlen( ), mi_fp_retscale( ), mi_fp_rettype( ), mi_fp_returnisnull( ), mi_fp_setargprec( ),** and **mi_fp_setretprec( ).**

For more information about return-value information in an **MI_FPARAM** structure or about the precision of a fixed-point data type, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_retscale( )

The **mi_fp_retscale( )** accessor function obtains the scale of a return value of a user-defined function from its associated **MI_FPARAM** structure.

Syntax

```
mi_integer mi_fp_retscale(fparam_ptr, ret_pos)
   MI_FPARAM *fparam_ptr;
   mi_integer ret_pos;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
ret_pos	is the index position into the return-scale array for the return value whose scale you want. For user-defined functions, the only valid value is 0.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fp_retscale( )** function obtains the scale of the routine return value at position *ret_pos* from the **MI_FPARAM** structure that *fparam_ptr* references. The scale is an attribute of the return-value data type. The meaning of the scale depends on the associated data type, as the following table shows.

Data Type	Meaning of Scale
DECIMAL (fixed-point), MONEY	The number of digits to the right of the decimal point
DECIMAL (floating-point)	The value 255
DATETIME, INTERVAL	The encoded integer value for the end qualifier of the data type; *end_qual* in the qualifier: *start_qual* TO *end_qual*

If you call **mi_fp_retscale( )** on some other data type, the function returns zero (0).

The **MI_FPARAM** structure stores information about the scale of function return values in the zero-based return-scale array. To obtain information about the *n*th return value, use a *ret_pos* value of *n*-1. For example, the following call to **mi_fp_retscale( )** obtains the scale for the first return value of the **my_func( )** user-defined function, with which **fparam1** is associated:

```
mi_integer my_func(..., fparam1)
   ...
   MI_FPARAM *fparam1;
{
   mi_integer ret_scale;
   ...
   ret_scale = mi_fp_retscale(fparam1, 0);
```

Important: C user-defined functions always have only one return value.

Return Values

>=0	is the scale, in number of digits, of the fixed-point or floating-point return value at position *ret_pos*.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_fp_argscale( ), mi_fp_retlen( ), mi_fp_retprec( ), mi_fp_rettype( ), mi_fp_returnisnull( ), mi_fp_setargscale( ),** and **mi_fp_setretscale( ).**

For more information about return-value information in an **MI_FPARAM** structure or about the scale of a fixed-point data type, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_rettype( )

The **mi_fp_rettype( )** accessor function obtains the type identifier for the data type of a return value of a user-defined function from its associated **MI_FPARAM** structure.

Syntax

```
MI_TYPEID *mi_fp_rettype(fparam_ptr, ret_pos)
   MI_FPARAM *fparam_ptr;
   mi_integer ret_pos;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
ret_pos	is the index position into the return-type array for the return value whose type identifier you want. For user-defined functions, the only valid value is `0`.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fp_rettype( )** function obtains the type identifier of the routine return value at position *ret_pos* from the **MI_FPARAM** structure that *fparam_ptr* references. The type identifier is an integer value that indicates a particular data type. The **MI_FPARAM** structure stores information about the type identifiers of function return values in the zero-based return-type array. To obtain information about the *n*th return value, use a *ret_pos* value of *n*-1. For example, the following call to **mi_fp_rettype( )** sets the type identifier for the first return value of the **my_func( )** user-defined function, with which **fparam1** is associated:

```
mi_integer my_func(..., fparam1)
   ...
   MI_FPARAM *fparam1;
{
   MI_TYPEID *ret_type;
   ...
   ret_type = mi_fp_rettype(fparam1, 0);
```

Important: C user-defined functions always have only one return value.

Return Values

An **MI_TYPEID** pointer	is a pointer to the type identifier of the return value at position *ret_pos*.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_fp_argtype( ), mi_fp_retlen( ), mi_fp_retprec( ), mi_fp_retscale( ), mi_fp_returnisnull( ), mi_fp_setargtype( ),** and **mi_fp_setrettype( ).**

For more information about return-value information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_returnisnull( )

The **mi_fp_returnisnull( )** accessor function determines whether the return value of a user-defined function is an SQL NULL from its associated **MI_FPARAM** structure.

Syntax

```
mi_boolean mi_fp_returnisnull(fparam_ptr, ret_pos)
   MI_FPARAM *fparam_ptr;
   mi_integer ret_pos;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
ret_pos	is the index position into the null-return array for the return value to check for a NULL value. For user-defined functions, the only valid value is 0.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fp_returnisnull( )** function determines whether the routine return value at position *ret_pos* in the **MI_FPARAM** structure that *fparam_ptr* references contains the SQL NULL value. The **MI_FPARAM** structure stores information about whether function return values are NULL in the zero-based null-return array. To obtain information about the *n*th return value, use a *ret_pos* value of *n*-1. For example, the following call to **mi_fp_returnisnull( )** determines whether the first return value of the **my_func( )** user-defined function, with which **fparam1** is associated is NULL:

```
mi_integer my_func(..., fparam1)
   ...
   MI_FPARAM *fparam1;
{
   if ( mi_fp_returnisnull(fparam1, 0) == MI_TRUE )
      /* code to handle NULL return value */
```

Important: C user-defined functions always have only one return value.

Return Values

MI_TRUE	indicates that the return value at position *ret_pos* is NULL.
MI_FALSE	indicates that the return value at position *ret_pos* is not NULL.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_fp_argisnull( ), mi_fp_retlen( ), mi_fp_retprec( ), mi_fp_retscale( ), mi_fp_rettype( ), mi_fp_setargisnull( ),** and **mi_fp_setreturnisnull( ).**

For more information about return-value information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_setargisnull( )

The **mi_fp_setargisnull( )** accessor function sets the value of an argument of a user-defined routine to an SQL NULL in its associated **MI_FPARAM** structure.

Syntax

```
void mi_fp_setargisnull(fparam_ptr, arg_pos, is_null)
   MI_FPARAM *fparam_ptr;
   mi_integer arg_pos;
   mi_integer is_null;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
arg_pos	is the index position into the null-argument array for the argument that you want set to NULL.
is_null	is the value that determines whether the *arg_pos*+1 argument is NULL.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fp_setargisnull( )** function sets the routine argument at position *arg_pos* in the **MI_FPARAM** structure that *fparam_ptr* references to the appropriate SQL NULL value. The **MI_FPARAM** structure stores information about whether routine arguments are NULL in the zero-based null-argument array. To set the *n*th argument, use an *arg_pos* value of *n*-1. For example, the following call to **mi_fp_setargisnull( )** sets to NULL the third argument of the **my_func( )** UDR, with which **fparam1** is associated:

```
mi_integer my_func(..., fparam1)
   ...
   MI_FPARAM *fparam1;
{
   ...
   mi_fp_setargisnull(fparam1, 2, MI_TRUE);
```

You can specify the following values for the *is_null* argument.

Value of *is_null*	Meaning
MI_FALSE	The argument that *arg_pos* identifies is not NULL.
MI_TRUE	The argument that *arg_pos* identifies is NULL.

Return Values

None.

Related Topics

See also the descriptions of **mi_fp_argisnull( ), mi_fp_setarglen( ), mi_fp_setargprec( ), mi_fp_setargscale( ), mi_fp_setargtype( ), mi_fp_returnisnull( ),** and **mi_fp_setreturnisnull( ).**

For more information about argument information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_setarglen( )

The **mi_fp_setarglen( )** accessor function sets the length of an argument of a user-defined routine in its associated **MI_FPARAM** structure.

Syntax

```
void mi_fp_setarglen(fparam_ptr, arg_pos, arg_len)
   MI_FPARAM *fparam_ptr;
   mi_integer arg_pos;
   mi_integer arg_len;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
arg_pos	is the index position into the argument-length array for the argument whose length you want to set.
arg_len	is the length, in bytes, to set for the *arg_pos*+1 argument.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

The **mi_fp_setarglen( )** function sets the length of the routine argument at position *arg_pos* in the **MI_FPARAM** structure that *fparam_ptr* references. The **MI_FPARAM** structure stores information about the lengths of routine arguments in the zero-based argument-length array. To set information for the *n*th argument, use an *arg_pos* value of *n*-1. For example, the following call to **mi_fp_setarglen( )** sets the length for the third argument of the **my_func( )** UDR, with which **fparam1** is associated:

```
mi_integer my_func(..., fparam1)
   ...
   MI_FPARAM *fparam1;
{
   ...
   mi_fp_setarglen(fparam1, 2, 4);
```

Return Values

None.

Related Topics

See also the descriptions of **mi_fp_arglen( ), mi_fp_setargisnull( ), mi_fp_setargprec( ), mi_fp_setargscale( ), mi_fp_setargtype( ), mi_fp_retlen( ),** and **mi_fp_setretlen( ).**

For more information about argument information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_setargprec( )

The **mi_fp_setargprec( )** accessor function sets the precision of a fixed-point or floating-point argument of a user-defined routine in its associated **MI_FPARAM** structure.

Syntax

```
void mi_fp_setargprec(fparam_ptr, arg_pos, arg_prec)
   MI_FPARAM *fparam_ptr;
   mi_integer arg_pos;
   mi_integer arg_prec;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
arg_pos	is the index position into the argument-precision array for the argument whose precision you want to set.
arg_prec	is the integer precision, in number of digits, to set for the *arg_pos*+1 argument.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fp_setargprec( )** function sets the precision of the routine argument at position *arg_pos* in the **MI_FPARAM** structure that *fparam_ptr* references.

The precision is an attribute of the data type that represents the total number of digits the routine return value can hold, as follows.

Data Type	Meaning
DECIMAL, MONEY	Number of significant digits in the fixed-point or floating-point (DECIMAL) column
DATETIME, INTERVAL	Number of digits that are stored in the date and/or time column with the specified qualifier
Character, Varying-character	Maximum number of characters in the column

The **MI_FPARAM** structure stores information about the precision of routine arguments in the zero-based argument-precision array. To set information for the *n*th argument, use an *arg_pos* value of *n*-1. For example, the following call to **mi_fp_setargprec( )** sets the precision for the third argument of the **my_func( )** UDR, with which **fparam1** is associated:

```
mi_integer my_func(..., fparam1)
   ...
   MI_FPARAM *fparam1;
{
   ...
   mi_fp_setargprec(fparam1, 2, 10);
```

Return Values

None.

Related Topics

See also the descriptions of **mi_fp_argprec( ), mi_fp_setargisnull( ), mi_fp_setarglen( ), mi_fp_setargscale( ), mi_fp_setargtype( ), mi_fp_retprec( ),** and **mi_fp_setretprec( ).**

For more information about argument information in an **MI_FPARAM** structure or about the precision of a fixed-point data type, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_setargscale( )

The **mi_fp_setargscale( )** accessor function sets the scale of an argument of a user-defined routine in its associated **MI_FPARAM** structure.

Syntax

```
void mi_fp_setargscale(fparam_ptr, arg_pos, arg_scale)
   MI_FPARAM *fparam_ptr;
   mi_integer arg_pos;
   mi_integer arg_scale;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
arg_pos	is the index position into the argument-scale array for the argument whose scale you want to set.
arg_scale	is the integer scale, in number of digits, to set for the *arg_pos*+1 argument.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fp_setargscale( )** function sets the scale of the routine argument at position *arg_pos* in the **MI_FPARAM** structure that *fparam_ptr* references.

The scale is an attribute of the data type. The meaning of the scale depends on the associated data type, as the following table shows.

Data Type	**Meaning of Scale**
DECIMAL (fixed-point), MONEY	The number of digits to the right of the decimal point
DECIMAL (floating-point)	The value 255
DATETIME, INTERVAL	The encoded integer value for the end qualifier of the data type; *end_qual* in the qualifier: *start_qual* TO *end_qual*

The **MI_FPARAM** structure stores information about the scale of routine arguments in the zero-based argument-scale array. To set information for the *n*th argument, use an *arg_pos* value of *n*-1. For example, the following call to **mi_fp_setargscale( )** sets the scale for the third argument of the **my_func( )** UDR, with which **fparam1** is associated:

```
mi_integer my_func(..., fparam1)
   ...
   MI_FPARAM *fparam1;
{
   ...
   mi_fp_setargscale(fparam1, 2, 4);
```

Return Values

None.

Related Topics

See also the descriptions of **mi_fp_argscale( ), mi_fp_setargisnull( ), mi_fp_setarglen( ), mi_fp_setargprec( ), mi_fp_setargtype( ), mi_fp_retscale( ),** and **mi_fp_setretscale( ).**

For more information about argument information in an **MI_FPARAM** structure or about the precision of a fixed-point data type, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_setargtype( )

The **mi_fp_setargtype( )** accessor routine sets the type identifier for the data type of an argument of a user-defined routine in its associated **MI_FPARAM** structure.

Syntax

```
void mi_fp_setargtype(fparam_ptr, arg_pos, arg_typeid)
   MI_FPARAM *fparam_ptr;
   mi_integer arg_pos;
   MI_TYPEID *arg_typeid;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
arg_pos	is the index position into the argument-type array for the argument whose type identifier you want to set.
arg_typeid	is a pointer to the type identifier that specifies the data type to set for the *arg_pos* + 1 argument.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fp_setargtype( )** function sets the type identifier of the routine argument at position *arg_pos* in the **MI_FPARAM** structure that *fparam_ptr* references. The type identifier indicates a particular data type. The **MI_FPARAM** structure stores information about the type identifiers of routine arguments in the zero-based argument-type array. To set information about the *n*th argument, use an *arg_pos* value of *n*-1.

For example, the following call to **mi_fp_setargtype( )** obtains the type identifier for the third argument of the **my_func( )** UDR, with which **fparam1** is associated:

```
mi_integer my_func(..., fparam1)
   ...
   MI_FPARAM *fparam1;
{
   MI_TYPEID *arg_type;
   ...
   arg_type = mi_type_
   mi_fp_setargtype(fparam1, 2, arg_type);
```

Return Values

None.

Related Topics

See also the descriptions of **mi_fp_argtype( ), mi_fp_setargisnull( ), mi_fp_setarglen( ), mi_fp_setargprec( ), mi_fp_setargscale( ), mi_fp_rettype( ),** and **mi_fp_setrettype( ).**

For more information about argument information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_setcolid( )

The **mi_fp_setcolid( )** accessor function sets the column identifier of the column that is associated with the user-defined routine from its **MI_FPARAM** structure.

Syntax

```
void mi_fp_setcolid(fparam_ptr, value)
   MI_FPARAM *fparam_ptr;
   mi_integer value;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
value	is the intended value of the column.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Warning: This advanced function can adversely affect your UDR if you use the function incorrectly. Use it only when no regular DataBlade API function can perform the task you need done.

Usage

The **mi_fp_setcolid( )** function sets the column identifier for the user-defined routine that is associated with the *fparam_ptr* **MI_FPARAM** structure. The column identifier is the location of the column within the row structure (with the first column starting at offset `0`). The column identifier and row structure identify the column with which the UDR invocation is associated. To set the row structure, use the **mi_fp_setrow( )** function.

This function is valid *only* when you need to create a smart large object in either of the following UDRs:

- Another iteration of the UDR
- A UDR that is called through the Fastpath interface

For either case, you can use **mi_fp_setcolid( )** to set the column identifier in the **MI_FPARAM** structure of the UDR *before* the UDR is called. When this UDR executes, it can obtain the column identifier from its **MI_FPARAM** structure and use it in conjunction with the **mi_lo_colinfo_by_ids( )** function to obtain the correct storage characteristics for the smart large object it needs to create.

Return Values

None.

Related Topics

See also the descriptions of **mi_fp_getcolid( ), mi_fp_setrow( ),** and **mi_lo_colinfo_by_ids( ).**

For more information about UDR information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_setfuncid( )

The **mi_fp_setfuncid( )** accessor function sets the routine identifier for a user-defined routine in its associated **MI_FPARAM** structure.

Syntax

```
void mi_fp_setfuncid(fparam_ptr, routine_id)
   MI_FPARAM *fparam_ptr;
   mi_funcid routine_id;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
routine_id	is the integer routine identifier to set in the **MI_FPARAM** structure.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fp_setfuncid( )** function sets the routine identifier for the UDR associated with the **MI_FPARAM** structure that *fparam_ptr* references. The routine identifier uniquely identifies the UDR within the database.

Tip: The DataBlade API provides the **mi_funcid** data type for routine identifiers. The **mi_funcid** data type has the same structure as the **mi_integer** data type. For compatibility with earlier versions, some DataBlade API functions still assume that routine identifiers are of type **mi_integer**.

Return Values

None.

Related Topics

See also the description of **mi_fp_getfuncid( ).**

For more information about UDR information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_setfuncstate( )

The **mi_fp_setfuncstate( )** accessor function sets the user-state pointer for the user-defined routine in its associated **MI_FPARAM** structure.

Syntax

```
void mi_fp_setfuncstate(fparam_ptr, state_ptr)
   MI_FPARAM *fparam_ptr;
   void *state_ptr;
```

fparam_ptr	is the pointer to the associated **MI_FPARAM** structure.
state_ptr	is a pointer to the user-state information that is stored as the user-state pointer.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fp_setfuncstate( )** function sets the user-state pointer for the user-defined routine that is associated with the *fparam_ptr* **MI_FPARAM** structure. The user-state pointer points to a user-defined buffer or structure that contains private state information for the user-defined routine. In the first invocation of the UDR, the database server sets the user-state pointer to NULL. Use the **mi_fp_funcstate( )** function to return the user-state pointer so that you can access the private state information within a UDR.

Cast the *state_ptr* pointer to "**void** *" before you store it as the user-state pointer. For example, the following call to **mi_fp_setfuncstate( )** casts a pointer to a structure called **udr_info** before it stores it as the user-state pointer in an **MI_FPARAM** structure:

```
MI_FPARAM *my_fparam;
struct udr_info *fi_ptr;
...
fi_ptr = (udr_info *)mi_dalloc(sizeof(udr_info),
   PER_COMMAND);
mi_fp_setfuncstate(my_fparam, (void *)fi_ptr);
```

Return Values

None.

Related Topics

See also the descriptions of **mi_fp_funcstate( ), mi_fp_setargisnull( ), mi_fp_setarglen( ), mi_fp_setargprec( ), mi_fp_setargscale( ), mi_fp_setargtype( ), mi_fp_setretlen( ), mi_fp_setretprec( ), mi_fp_setretscale( ), mi_fp_setrettype( ),** and **mi_fp_setreturnisnull( ).**

For more information about UDR information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_setisdone( )

The **mi_fp_setisdone( )** accessor function sets the iterator-completion flag for an iterator function in its associated **MI_FPARAM** structure.

Syntax

```
void mi_fp_setisdone(fparam_ptr, flag)
   MI_FPARAM *fparam_ptr;
   mi_integer flag;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
flag	is the integer iterator-completion flag to store in the **MI_FPARAM** structure, which indicates whether the end condition for the iterator function was reached.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

Use the **mi_fp_setisdone( )** function to tell the database server whether the current iterator function has reached its end condition. An *end condition* indicates that the generation of the active set is complete. The database server calls the iterator function with the SET_RETONE iterator-status value as long as the end condition has *not* been set.

Valid values for the iterator-completion *flag* argument are as follows.

Valid Value	Meaning	Description
1	The end condition was reached.	Once iterations are complete, the database server calls the iterator function one final time, with the iterator status of SET_END.
0	The end condition has *not* been reached.	The database server sets the iterator status to SET_RETONE and continues to call the iterator function.

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 iterations, which causes it to iterate the function in an infinite loop.

The iterator function does not return a value into the active set once the iterator-completion flag is set to 1.

Return Values

None.

Related Topics

See also the description of **mi_fp_request( ).**

For more information on how to create and call iterator functions, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_setnargs( )

The **mi_fp_setnargs( )** accessor function sets the number of arguments for the user-defined routine in its associated **MI_FPARAM** structure.

Syntax

```
void mi_fp_setnargs(fparam_ptr, arg_num)
   MI_FPARAM *fparam_ptr;
   mi_integer arg_num;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
arg_num	is the integer number of arguments for which the **MI_FPARAM** structure holds information.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fp_setnargs( )** function sets the number of arguments for the C UDR associated with the **MI_FPARAM** structure that *fparam_ptr* references to the value in *arg_num*.

Return Values

None.

Related Topics

See also the descriptions of **mi_fp_nargs( ), mi_fp_setargisnull( ), mi_fp_setarglen( ), mi_fp_setargprec( ), mi_fp_setargscale( ), mi_fp_setargtype( ),** and **mi_fp_setnrets( ).**

For more information about argument information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_setnrets( )

The **mi_fp_setnrets( )** accessor function sets the number of return values for the user-defined function in its associated **MI_FPARAM** structure.

Syntax

```
void mi_fp_setnrets(fparam_ptr, ret_num)
   MI_FPARAM *fparam_ptr;
   mi_integer ret_num;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
ret_num	is the integer number of return values to store in the **MI_FPARAM** structure.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fp_setnrets( )** function sets the number of return values for a C UDR associated with the **MI_FPARAM** structure that *fparam_ptr* references to the value in *ret_num*.

Important: C user-defined functions have only one return value.

Return Values

None.

Related Topics

See also the descriptions of **mi_fp_nrets( ), mi_fp_setnargs( ), mi_fp_setreturnisnull( ), mi_fp_setretlen( ), mi_fp_setretprec( ), mi_fp_setretscale( ),** and **mi_fp_setrettype( ).**

For more information about return-value information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_setretlen( )

The **mi_fp_setretlen( )** accessor function sets the length of a return value of a user-defined function from its associated **MI_FPARAM** structure.

Syntax

```
void mi_fp_setretlen(fparam_ptr, ret_pos, ret_len)
   MI_FPARAM *fparam_ptr;
   mi_integer ret_pos;
   mi_integer ret_len;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
ret_pos	is the index position into the return-length array for the return value whose length you want to set. For C user-defined functions, the only valid value is 0.
ret_len	is the length, in bytes, to set for the *ret_pos*+1 return value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fp_setretlen( )** function sets the length of the function return value at position *ret_pos* in the **MI_FPARAM** structure that *fparam_ptr* references. The **MI_FPARAM** structure stores information about return-value lengths in the zero-based return-length array.

To set information for the *n*th return value, use a *ret_pos* value of *n*-1. For example, the following call to **mi_fp_setretlen( )** sets the length to 4 bytes for the first return value of the **my_func( )** UDR, with which **fparam1** is associated:

```
mi_integer my_func(..., fparam1)
   ...
   MI_FPARAM *fparam1;
{
   ...
   mi_fp_setretlen(fparam1, 0, 4);
```

Important: C user-defined functions always have only one return value.

Return Values

None.

Related Topics

See also the descriptions of **mi_fp_arglen( ), mi_fp_setarglen( ), mi_fp_setretprec( ), mi_fp_setretscale( ), mi_fp_setrettype( ),** and **mi_fp_setreturnisnull( ).**

For more information about return-value information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_setretprec( )

The **mi_fp_setretprec( )** accessor function sets the precision of a return value of a user-defined function in its associated **MI_FPARAM** structure.

Syntax

```
void mi_fp_setretprec(fparam_ptr, ret_pos, ret_prec)
   MI_FPARAM *fparam_ptr;
   mi_integer ret_pos;
   mi_integer ret_prec;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
ret_pos	is the index position into the return-precision array for the return value whose precision you want to set. For C user-defined functions, the only valid value is 0.
ret_prec	is the precision, in number of digits, to set for the *ret_pos*+1 return value.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

The **mi_fp_setretprec( )** function sets the precision of the function return value at position *ret_pos* in the **MI_FPARAM** structure that *fparam_ptr* references.

The precision is an attribute of the data type that represents the total number of digits the routine return value can hold, as follows.

Data Type	**Meaning**
DECIMAL, MONEY	Number of significant digits in the fixed-point or floating-point (DECIMAL) column
DATETIME, INTERVAL	Number of digits that are stored in the date and/or time column with the specified qualifier
Character, Varying-character	Maximum number of characters in the column

The **MI_FPARAM** structure stores information about the precision of function return values in the zero-based return-precision array. To set information for the *n*th return value, use a *ret_pos* value of *n*-1.

For example, the following call to **mi_fp_setretprec( )** sets the precision of 10 for the first return value of the **my_func( )** UDR, with which **fparam1** is associated:

```
mi_integer my_func(..., fparam1)
   ...
   MI_FPARAM *fparam1;
{
   ...
   mi_fp_setretprec(fparam1, 0, 10);
```

Important: C user-defined functions always have only one return value.

Return Values

None.

Related Topics

See also the descriptions of **mi_fp_argprec( ), mi_fp_setargprec( ), mi_fp_setretlen( ), mi_fp_setretscale( ), mi_fp_setrettype( ),** and **mi_fp_setreturnisnull( ).**

For more information about return-value information in an **MI_FPARAM** structure or about the precision of a fixed-point data type, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_setretscale( )

The **mi_fp_setretscale( )** accessor function sets the scale of a return value of a user-defined function in its associated **MI_FPARAM** structure.

Syntax

```
void mi_fp_setretscale(fparam_ptr, ret_pos, ret_scale)
   MI_FPARAM *fparam_ptr;
   mi_integer ret_pos;
   mi_integer ret_scale;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
ret_pos	is the index position into the return-scale array for the return value whose scale you want to set. For C user-defined functions, the only valid value is 0.
ret_scale	is the scale, in number of digits, to set for the *ret_pos*+1 return value.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

The **mi_fp_setretscale( )** function sets the scale of the function return value at position *ret_pos* in the **MI_FPARAM** structure that *fparam_ptr* references.

The scale is an attribute of the return-value data type. The meaning of the scale depends on the associated data type, as the following table shows.

Data Type	**Meaning of Scale**
DECIMAL (fixed-point), MONEY	The number of digits to the right of the decimal point
DECIMAL (floating-point)	The value 255
DATETIME, INTERVAL	The encoded integer value for the end qualifier of the data type; *end_qual* in the qualifier: *start_qual* TO *end_qual*

The **MI_FPARAM** structure stores information about the scale of function return values in the zero-based return-scale array.

To set information for the *n*th return value, use a *ret_pos* value of *n*-1. For example, the following call to **mi_fp_setretscale( )** sets the scale to 4 for the first return value of the **my_func( )** UDR, with which **fparam1** is associated:

```
mi_integer my_func(..., fparam1)
   ...
   MI_FPARAM *fparam1;
{
   ...
   mi_fp_setretscale(fparam1, 0, 4);
```

Important: C user-defined functions always have only one return value.

Return Values

None.

Related Topics

See also the descriptions of **mi_fp_argscale( ), mi_fp_setargscale( ), mi_fp_setretlen( ), mi_fp_setretprec( ), mi_fp_setrettype( ),** and **mi_fp_setreturnisnull( ).**

For more information about return-value information in an **MI_FPARAM** structure or about the precision of a fixed-point data type, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_setrettype( )

The **mi_fp_setrettype( )** accessor function sets the type identifier for the data type of a return value of a user-defined function in its associated **MI_FPARAM** structure.

Syntax

```
void mi_fp_setrettype(fparam_ptr, ret_pos, ret_typeid)
   MI_FPARAM *fparam_ptr;
   mi_integer ret_pos;
   MI_TYPEID *ret_typeid;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
ret_pos	is the index position into the return-type array for the return value whose type identifier you want to set. For C user-defined functions, the only valid value is `0`.
ret_typeid	is a pointer to the integer type identifier that specifies the data type to set for the *ret_pos* + 1 return value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fp_setrettype( )** function sets the type identifier of the function return value at position *ret_pos* in the **MI_FPARAM** structure that *fparam_ptr* references. The type identifier is an integer value that indicates a particular data type. The *ret_typeid* value must be a valid type identifier. The **MI_FPARAM** structure stores information about the type identifiers of function return values in the zero-based return-type array.

To set information about the *n*th return value, use a *ret_pos* value of *n*-1. For example, the following call to **mi_fp_setrettype( )** sets the type identifier to **mi_integer** for the first return value of the **my_func( )** UDR, with which **fparam1** is associated:

```
mi_integer my_func(..., fparam1)
   ...
   MI_FPARAM *fparam1;
{
   MI_TYPEID *type_id;
   ...
   type_id = mi_typestring_to_id(conn, "integer");
   mi_fp_setrettype(fparam1, 0, type_id);
```

Important: C user-defined functions always have only one return value.

Return Values

None.

Related Topics

See also the descriptions of **mi_fp_argtype( ), mi_fp_setargtype( ), mi_fp_setretlen( ), mi_fp_setretprec( ), mi_fp_setretscale( ),** and **mi_fp_setreturnisnull( ).**

For more information about return-value information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_setreturnisnull( )

The **mi_fp_setreturnisnull( )** accessor function sets the value of a return value of a user-defined function to an SQL NULL in its associated **MI_FPARAM** structure.

Syntax

```
void mi_fp_setreturnisnull(fparam_ptr, ret_pos, is_null)
   MI_FPARAM *fparam_ptr;
   mi_integer ret_pos;
   mi_integer is_null;
```

fparam_ptr is a pointer to the associated **MI_FPARAM** structure.

ret_pos is the index position into the null-return array for the return value to set to NULL. For C user-defined functions, the only valid value is 0.

is_null is the value that determines whether the *ret_pos*+1 return value is NULL. You can specify the following values for the *is_null* argument:

MI_FALSE
The return value that *ret_pos* identifies is not NULL.

MI_TRUE
The return value that *ret_pos* identifies is NULL.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fp_setreturnisnull( )** function sets the function return value at position *ret_pos* in the **MI_FPARAM** structure that *fparam_ptr* references to the appropriate SQL NULL value. The **MI_FPARAM** structure stores information about whether function return values are NULL in the zero-based null-return array.

To set the *n*th return value, use a *ret_pos* value of *n*-1.

For example, the following call to **mi_fp_setreturnisnull( )** sets to NULL the first return value of the **my_func( )** UDR, with which **fparam1** is associated:

```
mi_integer my_func(..., fparam1)
...
   MI_FPARAM *fparam1;
{
   ...
   mi_fp_setreturnisnull(fparam1, 0, MI_TRUE);
```

Important: C user-defined functions always have only one return value.

Return Values

None.

Related Topics

See also the descriptions of **mi_fp_argisnull( ), mi_fp_setargisnull( ), mi_fp_setretlen( ), mi_fp_setretprec( ), mi_fp_setretscale( ),** and **mi_fp_setrettype( ).**

For more information about return-value information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_setrow( )

The **mi_fp_setrow( )** accessor function sets the row structure that is associated with the user-defined routine in its associated **MI_FPARAM** structure.

Syntax

```
void mi_fp_setrow(fparam_ptr, row_struc)
   MI_FPARAM *fparam_ptr;
   MI_ROW *row_struct
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.
row_struct	is a pointer to the row structure to store in the **MI_FPARAM** structure.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Warning: This advanced function can adversely affect your UDR if you use the function incorrectly. Use it only when no regular DataBlade API function can perform the task you need done.

Usage

The **mi_fp_setrow( )** function sets the row structure for the user-defined routine that is associated with the *fparam_ptr* **MI_FPARAM** structure. The row structure contains the column values of the row with which the UDR invocation is associated. The row structure and column identifier identify the column with which the UDR invocation is associated. To set the column identifier to a column within the row structure, use the **mi_fp_setcolid( )** function.

This function is valid *only* to create a smart large object either in another iteration of a UDR or in a UDR that is called through the Fastpath interface. In either case, you can use **mi_fp_setrow( )** to set the row structure in the **MI_FPARAM** structure of a UDR *before* the UDR is called. When the UDR executes, it can obtain the row structure from its **MI_FPARAM** structure and use this row structure in conjunction with the **mi_lo_colinfo_by_ids( )** function to obtain the correct storage characteristics for the creation of a smart large object.

Return Values

None.

Related Topics

See also the descriptions of **mi_fp_getrow( ), mi_fp_setcolid( ),** and **mi_lo_colinfo_by_ids( ).**

For more information about UDR information in an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fp_usr_fparam( )

The **mi_fp_usr_fparam( )** accessor function determines whether the database server or the developer has allocated the specified **MI_FPARAM** structure.

Syntax

```
mi_boolean mi_fp_usr_fparam (MI_FPARAM fparam_ptr)
   MI_FPARAM *fparam_ptr;
```

fparam_ptr	is a pointer to the associated **MI_FPARAM** structure.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The Fastpath interface uses an **MI_FPARAM** structure to hold information about the UDR or cast function to execute. This **MI_FPARAM** structure can be allocated in one of two ways:

- By a Fastpath look-up function (**mi_cast_get( ), mi_func_desc_by_typeid( ), mi_routine_get( ), mi_routine_get_by_typeid( ),** or **mi_td_cast_get( )**), as part of the function descriptor for the routine to execute
- By the developer, with the **mi_fparam_allocate( ))** or **mi_fparam_copy( )** function

The **mi_fp_usr_fparam( )** function determines which of these two methods was used to allocate the **MI_FPARAM** structure that *fparam_ptr* references.

Return Values

MI_TRUE	indicates that the **MI_FPARAM** structure, which *fparam_ptr* references, is a user-allocated structure.
MI_FALSE	indicates that the **MI_FPARAM** structure, which *fparam_ptr* references, was allocated by the database server.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_cast_get( ), mi_fparam_allocate( ), mi_fparam_copy( ), mi_fparam_free( ), mi_func_desc_by_typeid( ), mi_routine_exec( ), mi_routine_get( ), mi_routine_get_by_typeid( ),** and **mi_td_cast_get( ).**

For more information about how to use an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fparam_allocate( )

The **mi_fparam_allocate( )** function allocates an **MI_FPARAM** structure and returns a pointer to this structure.

Syntax

```
MI_FPARAM *mi_fparam_allocate(nargs)
   mi_integer nargs;
```

nargs	is the number of arguments that the new **MI_FPARAM** structure can hold.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fparam_allocate( )** function is a constructor function for the **MI_FPARAM** structure. It allocates an **MI_FPARAM** structure that holds information for *nargs* arguments. Use this function for the **MI_FPARAM** structure that Fastpath look-up functions allocate.

Server Only

The **mi_fparam_allocate( )** function allocates a new **MI_FPARAM** structure in the PER_COMMAND memory duration.

End of Server Only

Return Values

An **MI_FPARAM** pointer	is a pointer to the **MI_FPARAM** structure for which **mi_fparam_allocate( )** has allocated memory.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_fp_usr_fparam( ), mi_fparam_copy( ),** and **mi_fparam_free( )**.

For more information about how to use an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fparam_copy( )

The **mi_fparam_copy( )** function copies an existing **MI_FPARAM** structure to a newly allocated **MI_FPARAM** structure field by field.

Syntax

```
MI_FPARAM *mi_fparam_copy(source_fparam_ptr)
   MI_FPARAM *source_fparam_ptr;
```

source_fparam_ptr	is a pointer to the **MI_FPARAM** structure to copy.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fparam_copy( )** function is a constructor function for the **MI_FPARAM** structure. It performs the following tasks:

1. It allocates a new **MI_FPARAM** structure to hold the copy.
2. It copies the fields of the source **MI_FPARAM** structure, which *source_fparam_ptr* indicates, to the newly allocated **MI_FPARAM** structure.
3. It returns a pointer to the newly allocated **MI_FPARAM** structure.

Server Only

The **mi_fparam_copy( )** function allocates a new **MI_FPARAM** structure in the PER_COMMAND memory duration.

End of Server Only

Use the **mi_fparam_copy( )** function when your user-defined routine needs to reuse an **MI_FPARAM** structure that the database server originally allocated for a particular user-defined routine. You can make any modifications that the DataBlade API module requires to this new **MI_FPARAM** structure.

The **mi_fparam_copy( )** function performs a deep copy of every value from the source **MI_FPARAM** structure to the target **MI_FPARAM** structure *except* for the following information:

- The user-state pointer from the source **MI_FPARAM** structure is *not* copied.
- The row-descriptor pointer from the source **MI_FPARAM** structure is *not* copied.

The **mi_fparam_copy( )** function sets a field in the target **MI_FPARAM** structure to indicate that this **MI_FPARAM** structure is a user-allocated structure, not one that the database server allocated. In the target **MI_FPARAM** structure, this field value is independent of its value in the source **MI_FPARAM** structure. Use the **mi_fp_usr_fparam( )** function to determine the value of this field in an **MI_FPARAM** structure.

Return Values

An **MI_FPARAM** pointer	is a pointer to the newly allocated **MI_FPARAM** structure, which has the information from the source **MI_FPARAM** structure.

NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_fp_usr_fparam( ), mi_fparam_allocate( ),** and **mi_fparam_free( ).**

For more information about how to use an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fparam_free( )

The **mi_fparam_free( )** function deallocates resources used by an **MI_FPARAM** structure allocated on behalf of a user-defined routine.

Syntax

```
mi_integer mi_fparam_free(fparam_ptr)
   MI_FPARAM *fparam_ptr;
```

fparam_ptr	is a pointer to the **MI_FPARAM** structure for which to deallocate resources.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fparam_free( )** function is the destructor function for the **MI_FPARAM** structure. Use the **mi_fparam_free( )** function to free *only* those **MI_FPARAM** structures that have been allocated on behalf of user-defined routines with **mi_fparam_allocate( )** or **mi_fparam_copy( ).**

Important: The **mi_fparam_free( )** function generates an error if you attempt to free an **MI_FPARAM** structure that the database server has allocated.

It is an error to call this function to free an **MI_FPARAM** structure that the database server allocated internally.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_fp_usr_fparam( ), mi_fparam_allocate( ),** and **mi_fparam_copy( )**.

For more information about how to use an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fparam_get( )

The **mi_fparam_get( )** function retrieves a pointer to the **MI_FPARAM** structure that is associated with the function descriptor of a user-defined routine.

Syntax

```
MI_FPARAM *mi_fparam_get(conn, funcdesc_ptr)
   MI_CONNECTION *conn;
   MI_FUNC_DESC *funcdesc_ptr;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
funcdesc_ptr	is a pointer to the function descriptor whose **MI_FPARAM** structure **mi_fparam_get( )** is to return.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_fparam_get( )** function returns a pointer to the **MI_FPARAM** structure that is part of the function descriptor *funcdesc_ptr* references. Once you obtain the **MI_FPARAM** structure for a UDR you want to call with the Fastpath interface, you can modify this **MI_FPARAM** before you execute the UDR with Fastpath. The **mi_fparam_get( )** function is one of the functions of the Fastpath interface.

Return Values

An **MI_FPARAM** pointer	is a pointer to the **MI_FPARAM** structure that is associated with the UDR that the specified function descriptor identifies.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_fparam_get_current( ), mi_func_handlesnulls( ), mi_func_isvariant( ), mi_func_negator( ),** and **mi_routine_id_get( ).**

For more information about how to use an **MI_FPARAM** structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_fparam_get_current( )

The **mi_fparam_get_current( )** function obtains the **MI_FPARAM** structure for the currently running user-defined routine (UDR).

Syntax

```
MI_FPARAM *mi_fparam_get_current(void)
```

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Warning: This advanced function can adversely affect your UDR if you use the function incorrectly. Use it only when no regular DataBlade API function can perform the task you need done.

Usage

The **mi_fparam_get_current( )** function obtains the **MI_FPARAM** structure for the current UDR. When the routine manager invokes a UDR, it passes this **MI_FPARAM** structure as that last argument to the UDR. This function is valid only within a UDR call. It returns the NULL-valued pointer if called from a callback (such as an end-of-statement callback) that is not strictly called from a UDR.

The database server's routine manager allocates an **MI_FPARAM** structure for every UDR, regardless of whether the UDR explicitly declares one or not. If a UDR does not declare an **MI_FPARAM** structure but dynamically determines that it requires **MI_FPARAM** information, the UDR can use the **mi_fparam_get_current( )** function to obtain a pointer to its **MI_FPARAM** structure. This function is useful when the C function that implements a UDR needs to determine how many arguments the UDR was registered with (or called with).

Important: If you know that a UDR needs information in the **MI_FPARAM** structure, declare an **MI_FPARAM** structure as the final argument for the UDR. Restrict use of **mi_fparam_get_current( )** to UDRs that must dynamically determine that they need **MI_FPARAM** information.

Return Values

An **MI_FPARAM** pointer	is a pointer that references the **MI_FPARAM** structure for the current UDR.
NULL	indicates that the function was not successful or that it was called from a callback that was not strictly called from a UDR.

Related Topics

For more information about how to declare an **MI_FPARAM** argument or how to obtain routine information for a UDR, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_free( )

The **mi_free( )** routine frees the user memory that was previously allocated with the **mi_alloc( ), mi_dalloc( ),** or **mi_zalloc( )** function.

Syntax

```
void mi_free(ptr)
   void *ptr;
```

ptr	is a pointer to memory that **mi_alloc( ), mi_dalloc( ),** or **mi_zalloc( )** previously allocated.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_free( )** function frees the user memory that *ptr* references. This function behaves just like the **free( )** system call, except that it frees memory that one of the DataBlade API memory-management routines allocated. The **mi_free( )** function does *not* free memory allocated with **malloc( )**. To conserve resources, use the **mi_free( )** function to deallocate the user memory explicitly when your DataBlade API module no longer needs it. The **mi_free( )** function is the destructor function for user memory. If you do not explicitly free user memory, the database server frees it when its memory duration expires.

Server Only

In a C UDR, the **mi_free( )** function does *not* assume that the memory that *ptr* references is in the current memory duration. Instead, the function figures out the memory duration of the memory to deallocate.

End of Server Only

Client Only

In client LIBMI applications, you *must* call **mi_free( )** to free memory that it has allocated with **mi_alloc( ), mi_dalloc( ),** or **mi_zalloc( ).** Otherwise, this memory is not freed until the client LIBMI application exits. The database server does *not* automatically free memory for client LIBMI applications.

End of Client Only

Return Values

None.

Related Topics

See also the descriptions of **mi_alloc( ), mi_dalloc( ), mi_switch_mem_duration( ),** and **mi_zalloc( ).**

mi_func_commutator( )

The **mi_func_commutator( )** function obtains the name of a commutator function for a user-defined function.

Syntax

```
char *mi_func_commutator(funcdesc_ptr)
   MI_FUNC_DESC *funcdesc_ptr;
```

funcdesc_ptr	is a pointer to a function descriptor for a user-defined function.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_func_commutator( )** function obtains the name of the commutator function for the user-defined function associated with the *funcdesc_ptr* function descriptor. When you register a user-defined function, you can use the COMMUTATOR routine modifier of the CREATE FUNCTION statement to associate a commutator function with the user-defined function. To be a commutator function, a user-defined function must have either of the following similarities to the registered user-defined function:

- The commutator function takes the same arguments as the registered user-defined function but in opposite order.
- The commutator function returns the same result as the registered user-defined function.

The **mi_func_commutator( )** function is one of the functions of the Fastpath interface.

Return Values

A **char** pointer	is a pointer to the name of the commutator function for the user-defined function that the *funcdesc_ptr* function descriptor identifies.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_cast_get( ), mi_fparam_get( ), mi_func_handlesnulls( ), mi_func_isvariant( ), mi_func_negator( ), mi_routine_get( ), mi_routine_get_by_typeid( ), mi_routine_id_get( ),** and **mi_td_cast_get( ).**

mi_func_desc_by_typeid( )

The **mi_func_desc_by_typeid( )** function looks up a registered user-defined routine by its routine identifier and creates its function descriptor.

Syntax

```
MI_FUNC_DESC *mi_func_desc_by_typeid(conn, routine_id)
   MI_CONNECTION *conn;
   mi_funcid *routine_id;;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
	can be a pointer to a session-duration connection descriptor established by a previous call to **mi_get_session_connection( ).** Use of a session-duration connection descriptor is an *advanced* feature of the DataBlade API.
routine_id	is the routine identifier that uniquely identifies the UDR within the **sysprocedures** system catalog table.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_func_desc_by_typeid( )** function creates a function descriptor for the UDR that the *routine_id* argument specifies. The *routine_id* argument provides the routine identifier of the UDR. The function is one of the functions of the Fastpath interface. It is a constructor function for the function descriptor.

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 compatibility with earlier versions, some DataBlade API functions still assume that routine identifiers are of type **mi_integer**.

This function performs the following tasks:

1. Looks for a user-defined routine that matches the *routine_id* routine identifier in the **sysprocedures** system catalog table
2. Allocates a function descriptor for the UDR and saves the routine sequence in this descriptor
3. Allocates an **MI_FPARAM** structure for the routine and saves the argument and return-value information in this structure
4. Returns a pointer to the function descriptor that it has allocated for the user-defined routine

Server Only

When you pass a public connection descriptor (from **mi_open( )**), the **mi_func_desc_by_typeid( )** function allocates the new function descriptor in the PER_COMMAND memory duration. If you pass a session-duration connection descriptor (from **mi_get_session_connection( )**), **mi_func_desc_by_typeid( )** allocates the new function descriptor in the PER_SESSION memory duration. This function descriptor is called a session-duration function descriptor. For more information, see the *IBM Informix DataBlade API Programmer's Guide*.

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. Use them only when a regular connection descriptor or function descriptor cannot perform the task you need done.

End of Server Only

Return Values

An **MI_FUNC_DESC** pointer	is a pointer to the function descriptor for the UDR that *routine_id* identifies.
NULL	indicates that no matching user-defined routine was found or that the specified user-defined routine has multiple return values, which is possible with the following routines: • SPL routines that include the WITH RESUME clause in the RETURN statement • Iterator functions

Related Topics

See also the descriptions of **mi_fparam_get( ), mi_routine_end( ), mi_routine_exec( ), mi_routine_get( ), mi_routine_get_by_typeid( ),** and **mi_routine_id_get( ).**

mi_func_handlesnulls( )

The **mi_func_handlesnulls( )** function determines whether a user-defined routine (UDR) can handle SQL NULL values.

Syntax

```
mi_integer mi_func_handlesnulls(funcdesc_ptr)
   MI_FUNC_DESC *funcdesc_ptr;
```

funcdesc_ptr	is a pointer to a function descriptor for a UDR.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
No	Yes

Usage

The **mi_func_handlesnulls( )** function determines whether the UDR associated with the *funcdesc_ptr* function descriptor handles SQL NULL values as arguments. You can register a UDR that handles NULL arguments with the HANDLESNULLS routine modifier of the CREATE FUNCTION or CREATE PROCEDURE statement.

The **mi_func_handlesnulls( )** function is one of the functions of the Fastpath interface.

Return Values

1	indicates that the UDR that the *funcdesc_ptr* function descriptor identifies can handle NULL values.
2	indicates that the UDR that the *funcdesc_ptr* function descriptor identifies *cannot* handle NULL values.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_cast_get( ), mi_fparam_get( ), mi_func_commutator( ), mi_func_isvariant( ), mi_func_negator( ), mi_routine_get( ), mi_routine_get_by_typeid( ), mi_routine_id_get( ),** and **mi_td_cast_get( ).**

mi_func_isvariant( )

The **mi_func_isvariant( )** function determines whether a user-defined function is variant.

Syntax

```
mi_integer mi_func_isvariant(funcdesc_ptr)
   MI_FUNC_DESC *funcdesc_ptr;
```

funcdesc_ptr	is a pointer to a function descriptor for a user-defined function.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_func_isvariant( )** function determines whether the user-defined function associated with the *funcdesc_ptr* function descriptor is variant or nonvariant. A user-defined function is variant if it behaves in either of the following ways:

- Returns different results when it is invoked with the same arguments
- Modifies a database or variable state

You can register a user-defined function as variant with the VARIANT routine modifier. If you do not specify the VARIANT or NOT VARIANT routine modifier with the CREATE FUNCTION statement, the user-defined function is variant.

The **mi_func_isvariant( )** function is one of the functions of the Fastpath interface.

Return Values

1	indicates that the UDR that the *funcdesc_ptr* function descriptor identifies is variant.
2	indicates that the UDR that the *funcdesc_ptr* function descriptor identifies is *not* variant.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_cast_get( ), mi_fparam_get( ), mi_func_commutator( ), mi_func_handlesnulls( ), mi_func_negator( ), mi_routine_get( ), mi_routine_get_by_typeid( ), mi_routine_id_get( ),** and **mi_td_cast_get( ).**

mi_func_negator( )

The **mi_func_negator( )** function obtains the name of a negator function for a user-defined function.

Syntax

```
char *mi_func_negator(funcdesc_ptr)
   MI_FUNC_DESC *funcdesc_ptr;
```

funcdesc_ptr	is a pointer to a function descriptor for a user-defined function.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_func_negator( )** function obtains the name of the negator function for the user-defined function associated with the *funcdesc_ptr* function descriptor. A negator function evaluates the Boolean NOT condition for its associated Boolean user-defined function. When you register a Boolean user-defined function, you can specify the NEGATOR routine modifier in the CREATE FUNCTION statement to associate a negator function with the user-defined function.

The **mi_func_negator( )** function is one of the functions of the Fastpath interface.

Return Values

A **char** pointer	is a pointer to the name of the negator function for the user-defined function that the *funcdesc_ptr* function descriptor identifies.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_cast_get( ), mi_fparam_get( ), mi_func_commutator( ), mi_func_handlesnulls( ), mi_func_isvariant( ), mi_routine_get( ), mi_routine_get_by_typeid( ), mi_routine_id_get( ),** and **mi_td_cast_get( ).**

mi_funcarg_get_argtype( )

The **mi_funcarg_get_argtype( )** function returns the argument type for the companion-UDR argument of a cost or selectivity function.

Syntax

```
MI_FUNCARG_TYPE mi_funcarg_get_argtype(funcarg_ptr)
   MI_FUNCARG *funcarg_ptr;
```

funcarg_ptr	is a pointer to the **MI_FUNCARG** structure that describes the companion-UDR argument.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_funcarg_get_argtype( )** function returns the argument type from the **MI_FUNCARG** structure that *funcarg_ptr* references. The **MI_FUNCARG** structure describes an argument of a companion UDR to its cost or selectivity function. This argument type is represented by one of the following argument-type constants.

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

Use the **mi_funcarg_get_argtype( )** function in a cost or selectivity function to determine the kind of argument passed into the companion UDR.

Return Values

MI_FUNCARG_COLUMN	indicates that the companion-UDR argument is a constant value.
MI_FUNCARG_CONSTANT	indicates that the companion-UDR argument is a column value.
MI_FUNCARG_PARAM	indicates that the companion-UDR argument is a parameter.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_funcarg_get_colno( ), mi_funcarg_get_constant( ), mi_funcarg_get_datalen( ), mi_funcarg_get_datatype( ), mi_funcarg_get_distrib( ), mi_funcarg_get_routine_id( ), mi_funcarg_get_routine_name( ), mi_funcarg_get_tabid( ),** and **mi_funcarg_isnull( ).**

mi_funcarg_get_colno( )

The **mi_funcarg_get_colno( )** function returns the column number for the column associated with the companion-UDR argument of a cost or selectivity function.

Syntax

```
mi_integer mi_funcarg_get_colno(funcarg_ptr)
   MI_FUNCARG *funcarg_ptr;
```

funcarg_ptr	is a pointer to the **MI_FUNCARG** structure that describes the companion-UDR argument.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_funcarg_get_colno( )** function returns the column number of the argument from the **MI_FUNCARG** structure that *funcarg_ptr* references. The **MI_FUNCARG** structure describes an argument of a companion UDR to its cost or selectivity function. Use the **mi_funcarg_get_colno( )** function *only* for companion-UDR arguments that are column values; that is, only arguments for which the **mi_funcarg_get_argtype( )** function returns the MI_FUNCARG_COLUMN value. The column number is the value from the **colid** column of the **syscolumns** system catalog table.

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.

Use the **mi_funcarg_get_colno( )** function in a cost or selectivity function to obtain the column number for the column with which an argument passed into the companion UDR is associated.

Return Values

>= 0	is the column number for the column associated with the companion-UDR argument.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_funcarg_get_argtype( ), mi_funcarg_get_datalen( ), mi_funcarg_get_datatype( ), mi_funcarg_get_distrib( ), mi_funcarg_get_routine_id( ), mi_funcarg_get_routine_name( ),** and **mi_funcarg_get_tabid( ).**

mi_funcarg_get_constant( )

The **mi_funcarg_get_constant( )** function returns the constant value of a companion-UDR argument of a cost or selectivity function.

Syntax

```
MI_DATUM mi_funcarg_get_constant(funcarg_ptr)
   MI_FUNCARG *funcarg_ptr;
```

funcarg_ptr	is a pointer to the **MI_FUNCARG** structure that describes the companion-UDR argument.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_funcarg_get_constant( )** function returns the constant value of the argument from the **MI_FUNCARG** structure that *funcarg_ptr* references. The **MI_FUNCARG** structure describes an argument of a companion UDR to its cost or selectivity function. Use the **mi_funcarg_get_constant( )** function *only* for companion-UDR arguments that are constant values; that is, only arguments for which the **mi_funcarg_get_argtype( )** function returns the MI_FUNCARG_CONSTANT value.

The **mi_funcarg_get_constant( )** function returns the column value in an **MI_DATUM** structure. Make sure you use the passing mechanism appropriate for the data type of the value to obtain it from the **MI_DATUM** structure. Use the **mi_funcarg_get_constant( )** function in a cost or selectivity function to obtain the value of an argument passed into the companion UDR.

Return Values

An **MI_DATUM** value	is the value for the constant companion-UDR argument.
NULL	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_funcarg_get_argtype( ), mi_funcarg_get_datalen( ), mi_funcarg_get_datatype( ), mi_funcarg_get_routine_id( ), mi_funcarg_get_routine_name( ),** and **mi_funcarg_isnull( ).**

mi_funcarg_get_datalen( )

The **mi_funcarg_get_datalen( )** function returns the data length of an argument for the companion UDR of a cost or selectivity function.

Syntax

```
mi_integer mi_funcarg_get_datalen(funcarg_ptr)
   MI_FUNCARG *funcarg_ptr;
```

funcarg_ptr	is a pointer to the **MI_FUNCARG** structure that describes the companion-UDR argument.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_funcarg_get_datalen( )** function returns the data length of the argument from the **MI_FUNCARG** structure that *funcarg_ptr* references. The **MI_FUNCARG** structure describes an argument of a companion UDR to its cost or selectivity function. Use the **mi_funcarg_get_datalen( )** function in a cost or selectivity function to obtain the data length of an argument passed into the expensive UDR.

Return Values

>= 0	is the data length for the companion-UDR argument.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_funcarg_get_argtype( ), mi_funcarg_get_colno( ), mi_funcarg_get_constant( ), mi_funcarg_get_datatype( ), mi_funcarg_get_distrib( ), mi_funcarg_get_routine_id( ), mi_funcarg_get_routine_name( ), mi_funcarg_get_tabid( ),** and **mi_funcarg_isnull( ).**

mi_funcarg_get_datatype( )

The **mi_funcarg_get_datatype( )** function returns the type identifier for the data type of an argument for the companion UDR of a cost or selectivity function.

Syntax

```
MI_TYPEID *mi_funcarg_get_datatype(funcarg_ptr)
   MI_FUNCARG *funcarg_ptr;
```

funcarg_ptr	is a pointer to the **MI_FUNCARG** structure that describes the companion-UDR argument.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_funcarg_get_datatype( )** function returns the data type of the argument from the **MI_FUNCARG** structure that *funcarg_ptr* references. The **MI_FUNCARG** structure describes an argument of a companion UDR to its cost or selectivity function. The function returns this data type as a type identifier (**MI_TYPEID**). Use the **mi_funcarg_get_datatype( )** function in a cost or selectivity function to obtain the data type of an argument passed into the companion UDR.

Return Values

An **MI_TYPEID** pointer	points to a type identifier for the data type of the companion-UDR argument.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_funcarg_get_argtype( ), mi_funcarg_get_colno( ), mi_funcarg_get_constant( ), mi_funcarg_get_datalen( ), mi_funcarg_get_distrib( ), mi_funcarg_get_routine_id( ), mi_funcarg_get_routine_name( ), mi_funcarg_get_tabid( ),** and **mi_funcarg_isnull( ).**

mi_funcarg_get_distrib( )

The **mi_funcarg_get_distrib( )** function returns the data-distribution information for the column associated with the companion-UDR argument of a cost or selectivity function.

Syntax

```
mi_bitvarying *mi_funcarg_get_distrib(funcarg_ptr)
   MI_FUNCARG *funcarg_ptr;
```

funcarg_ptr	is a pointer to the **MI_FUNCARG** structure that describes the companion-UDR argument.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_funcarg_get_distrib( )** function returns data-distribution information from the **MI_FUNCARG** structure that *funcarg_ptr* references. The **MI_FUNCARG** structure describes an argument of a companion UDR to its cost or selectivity function. Use the **mi_funcarg_get_distrib( )** function *only* for companion-UDR arguments that are column values; that is, only arguments for which the **mi_funcarg_get_argtype( )** function returns the MI_FUNCARG_COLUMN value.

The data-distribution information is either an ASCII histogram that divides the column values into a prescribed number of bins or it is user-defined statistics. It is the value from the **encdat** column of the **sysdistrib** system catalog table. The **mi_funcarg_get_distrib( )** function returns the data-distribution information as an **mi_bitvarying** varying-length structure. The varying-length data is an **mi_statret** structure, which contains the actual data distribution.

Use the **mi_funcarg_get_distrib( )** function in a cost or selectivity function to obtain the distribution information for the column with which an argument passed into the expensive UDR is associated.

Return Values

An **mi_bitvarying** pointer	is a pointer to a varying-length structure that contains the data-distribution information for the column associated with the companion-UDR argument.
NULL	indicates one of the following conditions: • No data-distribution information exists for the column. • The companion-UDR argument is not of type MI_FUNCARG_COLUMN. • The function was *not* successful.

Related Topics

See also the descriptions of **mi_funcarg_get_argtype( ), mi_funcarg_get_colno( ), mi_funcarg_get_datalen( ), mi_funcarg_get_datatype( ), mi_funcarg_get_routine_id( ), mi_funcarg_get_routine_name( ),** and **mi_funcarg_get_tabid( ).**

For more information on cost and selectivity, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_funcarg_get_routine_id( )

The **mi_funcarg_get_routine_id( )** function returns the routine identifier for the companion UDR of a cost or selectivity function.

Syntax

```
mi_integer mi_funcarg_get_routine_id(funcarg_ptr)
   MI_FUNCARG *funcarg_ptr;
```

funcarg_ptr	is a pointer to the **MI_FUNCARG** structure that describes the companion UDR.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_funcarg_get_routine_id( )** function returns the routine identifier from the **MI_FUNCARG** structure that *funcarg_ptr* references. The **MI_FUNCARG** structure provides information about an argument of a companion UDR to its cost or selectivity function.

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. However, this DataBlade API function still assumes that routine identifiers are of type **mi_integer**.

Use the **mi_funcarg_get_routine_id( )** function in a cost or selectivity function to obtain the routine identifier for the companion UDR.

Return Values

>= 0	is the routine identifier for the companion UDR.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_funcarg_get_argtype( ), mi_funcarg_get_colno( ), mi_funcarg_get_constant( ), mi_funcarg_get_datalen( ), mi_funcarg_get_datatype( ), mi_funcarg_get_distrib( ), mi_funcarg_get_routine_name( ), mi_funcarg_get_tabid( ),** and **mi_funcarg_isnull( ).**

mi_funcarg_get_routine_name( )

The **mi_funcarg_get_routine_name( )** function returns the routine name for the companion UDR of a cost or selectivity function.

Syntax

```
mi_string *mi_funcarg_get_routine_name(funcarg_ptr)
   MI_FUNCARG *funcarg_ptr;
```

funcarg_ptr	is a pointer to the **MI_FUNCARG** structure that describes the companion UDR.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_funcarg_get_routine_name( )** function returns the routine name from the **MI_FUNCARG** structure that *funcarg_ptr* references. The **MI_FUNCARG** structure describes an argument of a companion UDR to its cost or selectivity function. Use the **mi_funcarg_get_routine_name( )** function in a cost or selectivity function to obtain the routine name for the companion UDR.

Return Values

An **mi_string** pointer	is a pointer to the routine name of the companion UDR.
NULL	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_funcarg_get_argtype( ), mi_funcarg_get_colno( ), mi_funcarg_get_constant( ), mi_funcarg_get_datalen( ), mi_funcarg_get_datatype( ), mi_funcarg_get_distrib( ), mi_funcarg_get_routine_id( ), mi_funcarg_get_tabid( ),** and **mi_funcarg_isnull( ).**

mi_funcarg_get_tabid( )

The **mi_funcarg_get_tabid( )** function returns the table identifier for the column associated with the companion-UDR argument of a cost or selectivity function.

Syntax

```
mi_integer mi_funcarg_get_tabid(funcarg_ptr)
   MI_FUNCARG *funcarg_ptr;
```

funcarg_ptr	is a pointer to the **MI_FUNCARG** structure that describes a companion-UDR argument.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_funcarg_get_tabid( )** function returns the table identifier from the **MI_FUNCARG** structure that *funcarg_ptr* references. The **MI_FUNCARG** structure describes an argument of a companion UDR to its cost or selectivity function. Use the **mi_funcarg_get_tabid( )** function *only* for companion-UDR arguments that are column values; that is, only arguments for which the **mi_funcarg_get_argtype( )** function returns the MI_FUNCARG_COLUMN value. The table identifier identifies the table that contains the column associated with the *funcarg_ptr* companion-UDR argument. It is the value from the **tabid** column of the **systables** system catalog table.

Use the **mi_funcarg_get_tabid( )** function in a cost or selectivity function to obtain the table identifier for the column with which an argument passed into the companion UDR is associated.

Return Values

>= 0	is the table identifier for the column associated with the companion-UDR argument.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_funcarg_get_argtype( ), mi_funcarg_get_colno( ), mi_funcarg_get_datalen( ), mi_funcarg_get_datatype( ), mi_funcarg_get_distrib( ), mi_funcarg_get_routine_id( ),** and **mi_funcarg_get_routine_name( ).**

mi_funcarg_isnull( )

The **mi_funcarg_isnull( )** function determines whether the companion-UDR argument of a cost or selectivity function contains the SQL NULL value.

Syntax

```
mi_boolean mi_funcarg_isnull(funcarg_ptr)
   MI_FUNCARG *funcarg_ptr;
```

funcarg_ptr	is a pointer to the **MI_FUNCARG** structure that describes the companion-UDR argument.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_funcarg_isnull( )** function determines whether the argument from the **MI_FUNCARG** structure that *funcarg_ptr* references contains the SQL NULL value. The **MI_FUNCARG** structure describes an argument of an expensive UDR to its cost or selectivity function. Use the **mi_funcarg_isnull( )** function *only* for companion-UDR arguments that are constant values; that is, only arguments for which the **mi_funcarg_get_argtype( )** function returns the MI_FUNCARG_CONSTANT value.

Use the **mi_funcarg_isnull( )** function in a cost or selectivity function to determine if the value of an argument passed into the companion UDR is the SQL NULL value.

Return Values

MI_TRUE	indicates that the companion-UDR argument contains the SQL NULL value.
MI_FALSE	indicates that the companion-UDR argument does *not* contain the SQL NULL value.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_funcarg_get_argtype( ), mi_funcarg_get_constant( ), mi_funcarg_get_datalen( ), mi_funcarg_get_datatype( ), mi_funcarg_get_routine_id( ),** and **mi_funcarg_get_routine_name( ).**

mi_get_bytes( )

The **mi_get_bytes( )** function copies the given number of bytes, converting any difference in alignment or byte order on the client computer to that of the server computer.

Syntax

```
mi_unsigned_char1 *mi_get_bytes(data_ptr, val_ptr, nbytes)
   mi_unsigned_char1 *data_ptr;
   char *val_ptr;
   mi_integer nbytes;
```

data_ptr	is a pointer to the buffer from which to copy bytes of data.
val_ptr	is a pointer to the buffer to which to copy bytes of data.
nbytes	is the number of bytes to copy.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_get_bytes( )** function copies *nbytes* bytes of data from the user-defined buffer that *data_ptr* references to the buffer that *val_ptr* references. Upon completion, **mi_get_bytes( )** returns the address of the next position from which data can be copied from the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *nbytes* bytes, ready for a subsequent copy. In other words, if *nbytes* is the length of the value that *val_ptr* identifies, the returned address is *nbytes* bytes advanced from the original buffer address in *data_ptr*.

The **mi_get_bytes( )** function is useful in a receive support function of an opaque data type that contains byte data. Use this function to receive untyped data (such as **void** *) within an opaque-type internal structure from a client application (which possibly has unaligned data buffers).

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_fix_integer( ), mi_fix_smallint( ), mi_get_date( ), mi_get_datetime( ), mi_get_decimal( ), mi_get_double_precision( ), mi_get_int8( ), mi_get_integer( ), mi_get_interval( ), mi_get_lo_handle( ), mi_get_money( ), mi_get_real( ), mi_get_smallint( ), mi_get_string( ),** and **mi_put_bytes( ).**

For more information on the use of **mi_get_bytes( )** in a receive support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_get_connection_info( )

The **mi_get_connection_info( )** function populates a connection-information descriptor with current connection parameters for an open connection.

Syntax

```
mi_integer mi_get_connection_info(conn, conn_info)
   MI_CONNECTION *conn;
   MI_CONNECTION_INFO *conn_info;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
conn_info	is a pointer to a user-provided connection-information descriptor, to store the current connection parameters.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_get_connection_info( )** function puts the current connection parameters for the connection that *conn* references into the descriptor (**MI_CONNECTION_INFO** structure) that *conn_info* references. The connection parameters in this structure include the name of the database server, the number of the server port, and a GLS locale:

```
typedef struct mi_connection_info
    {
    char   *server_name; /* INFORMIXSERVER */
    mi_integer server_port;  /* SERVERNUM */
    char   *locale;      /* Processing locale */
    mi_integer reserved1;    /* unused */
    mi_integer reserved2;    /* unused */
    } MI_CONNECTION_INFO;
```

Tip: You must allocate this connection-information descriptor *before* you call **mi_get_connection_info( )**.

You can pass the connection-information descriptor to **mi_server_connect( )** to specify the connection parameters for a connection from the client LIBMI application to the database server.

The **mi_get_connection_info( )** function initializes the DataBlade API when it is the first DataBlade API function in a client LIBMI application or a C user-defined routine.

Global Language Support

The GLS locale in the connection parameters refers to the *server* locale in a C UDR or to the *database* locale in a client LIBMI application. For more information about GLS locales, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_get_connection_user_data( ), mi_get_database_info( ), mi_get_default_connection_info( ), mi_get_parameter_info( ), mi_get_serverenv( ), mi_server_connect( ),** and **mi_set_default_connection_info( ).**

For a description of the connection-information descriptor, more information on how to examine the connection-information descriptor, or more information on ways to interact with the session environment, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_get_connection_option( )

The **mi_get_connection_option( )** function returns information about the database of the current connection.

Syntax

```
mi_integer mi_get_connection_option(conn, conn_option, result_ptr)
   MI_CONNECTION *conn;
   const mi_integer conn_option;
   mi_integer *result_ptr;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
conn_option	is an integer constant to indicate which connection attribute to obtain. For a list of valid constants, see the following “Usage” section.
result_ptr	is a pointer to the connection information that **mi_get_connection_option( )** obtains.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_get_connection_option( )** function puts information about the connection attribute that *conn* specifies in the location that *result_ptr* references.

The following table lists the valid values for *conn_option* and the associated possible values that *result_ptr* references.

Description	Connection-Option Constant	Result
Is the open database an ANSI-compliant database?	MI_IS_ANSI_DB	MI_TRUE or MI_FALSE
Does the open database use a transaction log?	MI_IS_LOGGED_DB	MI_TRUE or MI_FALSE
Is the database in exclusive mode? (Has the DATABASE statement included the EXCLUSIVE keyword?)	MI_IS_EXCLUSIVE_DB	MI_TRUE or MI_FALSE

This function copies this value into the buffer that *result_ptr* references. The function allocates memory for the result in the current memory duration. When you no longer need this result, free this memory.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful or the *conn_option* argument is unknown.

The **mi_get_connection_option( )** function raises an exception for the following conditions:

- Parameter problems, such as bad or misaligned result pointer
- An invalid connection

mi_get_connection_user_data( )

The **mi_get_connection_user_data( )** function obtains the address of user data associated with an open connection.

Syntax

```
mi_integer mi_get_connection_user_data(conn, user_data_ptr)
   MI_CONNECTION *conn;
   void **user_data_ptr;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).** can be a pointer to a session-duration connection descriptor established by a previous call to **mi_get_session_connection( ).** Use of a session-duration connection descriptor is an *advanced* feature of the DataBlade API.
user_data_ptr	is the address of the user-data pointer that is associated with the specified connection.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_get_connection_user_data( )** function obtains the address of the user-data pointer, which *user_data_ptr* references, from the connection descriptor that *conn* references. The user-data pointer is the address to a user-defined buffer or structure that contains private information you want to keep with the specified connection.

The DataBlade API does not interpret or touch the associated user-data pointer, other than to retrieve it from the connection descriptor. Cast the *user_data_ptr* pointer from **void **** to the address of the user-data pointer for the data structure before you use it as the user-data pointer in a DataBlade API module.

You can set the user-data pointer with the **mi_set_connection_user_data( )** function.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_get_connection_info( ), mi_get_database_info( ), mi_get_parameter_info( ),** and **mi_set_connection_user_data( ).**

mi_get_cursor_table( )

The **mi_get_cursor_table( )** function obtains the name of the database table that is associated with a specified cursor.

Syntax

```
mi_lvarchar *mi_get_cursor_table(cursor_name)
   mi_lvarchar *cursor_name;
```

cursor_name	is a pointer to the varying-length structure that contains the internal representation of the cursor name.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_get_cursor_table( )** function obtains the name of the table in the query that is associated with the cursor that *cursor_name* references. The function expects the cursor name to be in a varying-length (**mi_lvarchar**) structure. If the table is not present, **mi_get_cursor_table( )** returns an error. Otherwise, it returns the table name associated with the cursor. If the query is performing a join (which involves two tables), **mi_get_cursor_table( )** returns the name of the *first* table in the query.

When you create a generic UDR, you can use **mi_get_cursor_table( )** to obtain the name of the table from the associated cursor, as the following code fragment shows:

```
MI_CONNECTION *conn;
MI_STATEMENT *stmt_desc1;
mi_lvarchar *tbl_name;

stmt_desc1 = mi_prepare(conn, "select * from systables;", 0);
if ( (mi_open_prepared_statement(stmt_desc1, 0, 0, 0,
   0, 0, 0, 0, "curs1", 0, 0)) == MI_OK )
   tbl_name =
      mi_get_cursor_table(mi_string_to_lvarchar("curs1"));
   ...
```

In the preceding code fragment, `tbl_name` points to an **mi_lvarchar** structure whose data is the non-null-terminated string `systables`. The table name is useful to include in error messages.

Return Values

An **mi_lvarchar** pointer	is a pointer to a varying-length structure that contains the name of the table associated with the specified cursor.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_open_prepared_statement( )** and **mi_prepare( ).**

For information on how to pass a cursor name to a prepared statement, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_get_database_info( )

The **mi_get_database_info( )** function populates a database-information descriptor with current database parameters for an open connection.

Syntax

```
mi_integer mi_get_database_info(conn, db_info)
   MI_CONNECTION *conn;
   MI_DATABASE_INFO *db_info;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
db_info	is a pointer to a user-provided database-information descriptor, which stores the current database parameters.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_get_database_info( )** function puts the current database parameters for the connection that *conn* references into the database-information descriptor (**MI_DATABASE_INFO** structure) that *db_info* references. The database parameters include the name of the database, the user account, and the account password. The following table shows the 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
password	**char ***	The account password, as defined by the operating system

Tip: You must allocate this database-information descriptor *before* you call **mi_get_database_info( )**.

The **mi_get_database_info( )** function also initializes the DataBlade API when it is the first DataBlade API function in a client LIBMI application or a user-defined routine.

Server Only

In a user-defined routine, **mi_get_database_info( )** retrieves the same information as **mi_get_default_database_info( ).**

End of Server Only

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_connection_info( ), mi_get_default_database_info( ), mi_get_parameter_info( ), mi_get_serverenv( ),** and **mi_set_default_database_info( ).**

For a description of the database-information descriptor, more information on how to examine the database-information descriptor, or more information on ways to interact with the session environment, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_get_date( )

The **mi_get_date( )** function copies an **mi_date** (DATE) value, converting any difference in alignment or byte order on the client computer to that of the server computer.

Syntax

```
mi_unsigned_char1 *mi_get_date(data_ptr, date_ptr)
   mi_unsigned_char1 *data_ptr;
   mi_date *date_ptr;
```

data_ptr	is a pointer to the buffer from which to copy the **mi_date** value.
date_ptr	is a pointer to the buffer to which to copy the **mi_date** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_get_date( )** function copies a value from the user-defined buffer that *data_ptr* references into the buffer that *date_ptr* references. Upon completion, **mi_get_date( )** returns the address of the next position from which data can be copied from the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *n* bytes, ready for a subsequent copy. In other words, if *n* is the length of the value that *date_ptr* references, the returned address is *n* bytes advanced from the original buffer address in *data_ptr*.

The **mi_get_date( )** function is useful in a receive support function of an opaque data type that contains an **mi_date** value. Use this function to receive an **mi_date** field of an opaque-type internal structure from a client application (which possibly has unaligned data buffers).

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_bytes( ), mi_get_datetime( ), mi_get_decimal( ), mi_get_double_precision( ), mi_get_int8( ), mi_get_integer( ), mi_get_interval( ), mi_get_lo_handle( ), mi_get_money( ), mi_get_real( ), mi_get_smallint( ), mi_get_string( ),** and **mi_put_date( ).**

For more information on the use of **mi_get_date( )** in a receive support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_get_datetime( )

The **mi_get_datetime( )** function copies an **mi_datetime** (DATETIME) value, converting any difference in alignment or byte order on the client computer to that of the server computer.

Syntax

```
mi_unsigned_char1 *mi_get_datetime(data_ptr, dtime_ptr)
   mi_unsigned_char1 *data_ptr;
   mi_datetime *dtime_ptr;
```

data_ptr	is a pointer to the buffer from which to copy the **mi_datetime** value.
dtime_ptr	is a pointer to the buffer to which to copy the **mi_datetime** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_get_datetime( )** function copies a value from the user-defined buffer that *data_ptr* references into the buffer that *dtime_ptr* references. Upon completion, **mi_get_datetime( )** returns the address of the next position from which data can be copied from the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *n* bytes, ready for a subsequent copy. In other words, if *n* is the length of the value that *dtime_ptr* identifies, the returned address is *n* bytes advanced from the original buffer address in *data_ptr*.

The **mi_get_datetime( )** function is useful in a receive support function of an opaque data type that contains an **mi_datetime** value. Use this function to receive an **mi_datetime** field of an opaque-type internal structure from a client application (which possibly has unaligned data buffers).

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_bytes( ), mi_get_date( ), mi_get_decimal( ), mi_get_double_precision( ), mi_get_int8( ), mi_get_integer( ), mi_get_interval( ), mi_get_lo_handle( ), mi_get_money( ), mi_get_real( ), mi_get_smallint( ), mi_get_string( ),** and **mi_put_datetime( ).**

For more information on the use of **mi_get_datetime( )** in a receive support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_get_db_locale( )

The **mi_get_db_locale( )** function returns the value of the current database locale.

Syntax

```
mi_char *mi_get_db_locale(void);
```

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_get_db_locale( )** function returns the value of the current database locale, which indicates the IBM Informix GLS locale that the database server uses. The server-locale name gives information about language, territory, code set, and optionally other information about the database server. For more information about the server locale, see the *IBM Informix GLS User's Guide*.

The database server locale can be one of the following:

- The current session locale, if it is different than the value of the **DB_LOCALE** environment variable
- The value of DB_LOCALE
- `en_us.8859-1`, if DB_LOCALE is not set

Return Values

An **mi_char** pointer	is a pointer to the name of the current database locale

mi_get_dbnames( )

The **mi_get_dbnames( )** function retrieves the names of all databases available on the database server, corresponding to the logged-in connection.

Syntax

```
mi_integer mi_get_dbnames(conn, dbnameps, dbnamepssize, dbnamesb, dbnamesbsize)
   MI_CONNECTION *conn;
   char *dbnameps[];
   mi_integer dbnamepssize;
   char *dbnamesb;
   mi_integer dbnamesbsize;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
dbnameps	contains pointers to database names.
dbnamepssize	is the size of *dbnameps*.
dbnamesb	is a pointer to the result list of database names.
dbnamesbsize	is the size of *dbnamesb*.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	No

Usage

The **mi_get_dbnames( )** function copies database names into *dbnamesb* (up to *dbnamesbsize*) and copies database-name pointers into *dbnameps* (up to *dbnamepssize*). The user allocates *dbnamesb* and *dbnamesps*.

If the connection is not logged in, this function retrieves the names of databases available on **$INFORMIXSERVER**. If the connection is not provided (*conn* argument is NULL), this function retrieves the names of databases on all database servers.

Return Values

>=0	is the number of retrieved database names.
MI_ERROR	indicates that the function was not successful.

mi_get_decimal( )

The **mi_get_decimal( )** function copies an **mi_decimal** (DECIMAL) value, converting any difference in alignment or byte order on the client computer to that of the server computer.

Syntax

```
mi_unsigned_char1 *mi_get_decimal(data_ptr, dec_ptr)
   mi_unsigned_char1 *data_ptr;
   mi_decimal *dec_ptr;
```

data_ptr	is a pointer to the buffer from which to copy the **mi_decimal** value.
dec_ptr	is a pointer to the buffer to which to copy the **mi_decimal** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_get_decimal( )** function copies a value from the user-defined buffer that *data_ptr* references into the buffer that *dec_ptr* references. Upon completion, **mi_get_decimal( )** returns the address of the next position from which data can be copied from the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *n* bytes, ready for a subsequent copy. In other words, if *n* is the length of the value that *dec_ptr* identifies, the returned address is *n* bytes advanced from the original buffer address in *data_ptr*.

The **mi_get_decimal( )** function is useful in a receive support function of an opaque data type that contains an **mi_decimal** value. Use this function to receive an **mi_decimal** field of an opaque-type internal structure from a client application (which possibly has unaligned data buffers).

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_bytes( ), mi_get_date( ), mi_get_datetime( ), mi_get_double_precision( ), mi_get_int8( ), mi_get_integer( ), mi_get_interval( ), mi_get_lo_handle( ), mi_get_money( ), mi_get_real( ), mi_get_smallint( ), mi_get_string( ),** and **mi_put_decimal( ).**

For more information on the use of **mi_get_decimal( )** in a receive support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_get_default_connection_info( )

The **mi_get_default_connection_info( )** function populates a connection-information descriptor with the default connection parameters.

Syntax

```
mi_integer mi_get_default_connection_info(conn_info)
   MI_CONNECTION_INFO *conn_info;
```

conn_info	is a pointer to a user-allocated connection-information descriptor in which to store the default connection parameters.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_get_default_connection_info( )** function retrieves default connection parameters into the connection-information descriptor (**MI_CONNECTION_INFO** structure) that *conn_info* references. The connection parameters include the name of the database server and a GLS locale.

Server Only

In a C UDR, the GLS locale in the default connection parameters refers to the *server* locale.

End of Server Only

Client Only

In a client LIBMI application, the GLS locale in the default connection parameters refers to the *database* locale.

End of Client Only

For more information about GLS locales, see the *IBM Informix GLS User's Guide*.

You can pass the *conn_info* descriptor to **mi_server_connect( )** to specify the default connection parameters for a connection.

Tip: You must allocate this connection-information descriptor *before* you call **mi_get_default_connection_info( )**.

If no default value exists for a particular field, the **mi_get_default_connection_info( )** function sets string fields to a NULL-valued pointer and integer fields to `0`.

When **mi_get_default_connection_info( )** is the first DataBlade API function in a client LIBMI application or a user-defined routine, it also initializes the DataBlade

API.

Server Only

In a C UDR, the **mi_get_default_connection_info( )** function returns the same information as the **mi_get_connection_info( )** function.

End of Server Only

Client Only

In a client LIBMI application, you can set the connection parameters with the **mi_set_default_connection_info( )** function.

End of Client Only

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_get_connection_info( ), mi_server_connect( ),** and **mi_set_default_connection_info( ).**

For a description of the connection-information descriptor, more information on how to use the connection-information descriptor, or more information on ways to interact with the session environment, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_get_default_database_info( )

The **mi_get_default_database_info( )** function populates a database-information descriptor with the default database parameters.

Syntax

```
mi_integer mi_get_default_database_info(db_info)
   MI_DATABASE_INFO *db_info;
```

db_info	is a pointer to a user-allocated database-information descriptor in which to store the default database parameters.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_get_default_database_info( )** function returns default database parameters into the database-information descriptor (**MI_DATABASE_INFO** structure) that *db_info* references. The database parameters include the name of the database, the user account, and the account password. The **mi_open( )** function can use these default database parameters when it establishes a connection.

Tip: You must allocate this database-information descriptor *before* you call **mi_get_default_database_info( )**.

If no default value exists for a particular field, the **mi_get_default_database_info( )** function sets string fields to a NULL-valued pointer and integer fields to `0`.

The **mi_get_default_database_info( )** function also initializes the DataBlade API when it is the first DataBlade API function in a client LIBMI application or a user-defined routine.

Server Only

In a user-defined routine, **mi_get_default_database_info( )** retrieves the same information as **mi_get_database_info( ).**

End of Server Only

Client Only

In a client LIBMI application, you can set the application database parameters with the **mi_set_default_database_info( )** function.

End of Client Only

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_get_database_info( ), mi_open( ),** and **mi_set_default_database_info( ).**

For more information on the database-information descriptor, more information on how to use the database-information descriptor, or more information on ways to interact with the session environment, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_get_double_precision( )

The **mi_get_double_precision( )** function copies an **mi_double_precision** (FLOAT) value, converting any difference in alignment or byte order on the client computer to that of the server computer.

Syntax

```
mi_unsigned_char1 *mi_get_double_precision(data_ptr, dbl_ptr)
   mi_unsigned_char1 *data_ptr;
   mi_double_precision *dbl_ptr;
```

data_ptr	is a pointer to the buffer from which to copy the **mi_double_precision** value.
dbl_ptr	is a pointer to the buffer to which to copy the **mi_double_precision** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_get_double_precision( )** function copies a value from the user-defined buffer that *data_ptr* references into the buffer that *dbl_ptr* references. Upon completion, **mi_get_double_precision( )** returns the address of the next position from which data can be copied from the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *n* bytes, ready for a subsequent copy. In other words, if *n* is the length of the value that *dbl_ptr* indicates, the returned address is advanced *n* bytes from the original buffer address in *data_ptr*.

The **mi_get_double_precision( )** function is useful in a receive support function of an opaque data type that contains an **mi_double_precision** value. Use this function to receive an **mi_double_precision** field of an opaque-type internal structure from a client application (which possibly has unaligned data buffers).

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_bytes( ), mi_get_date( ), mi_get_datetime( ), mi_get_decimal( ), mi_get_int8( ), mi_get_integer( ), mi_get_interval( ), mi_get_lo_handle( ), mi_get_money( ), mi_get_real( ), mi_get_smallint( ), mi_get_string( ),** and **mi_put_double_precision( ).**

For more information on the use of **mi_get_double_precision( )** in a receive support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_get_duration_size( )

The **mi_get_duration_size( )** function returns the TOTAL pool size of the specified memory duration.

Syntax

```
mi_integer MI_PROC_EXPORT
mi_get_duration_size(MI_MEMORY_DURATION md)
```

md	is the specified memory duration.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_get_duration_size( )** function lets you determine the size of a memory pool.

Return Values

size	is the size of the memory pool.

Related Topics

See also the description of **mi_get_memptr_duration( ).**

mi_get_id( )

The **mi_get_id( )** function enables you to obtain statement identifiers or session identifiers.

Syntax

```
mi_integer mi_get_id(*conn, id_type)
   MI_CONNECTION *conn;
   MI_ID id_type;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
id_type	is an integer that specifies the type of identifier to obtain. Valid values for *id_type* follow: **MI_STATEMENT_ID** obtains a statement identifier. **MI_SESSION_ID** obtains a session identifier.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_get_id( )** function returns a statement or session identifier, depending on the value of the *id_type* parameter. To specify the identifier to obtain, **mi_get_id( )** has an *id_type* argument of type **MI_ID**, which supports the cursor-action constants in the following table.

Value of id_type	Description
MI_STATEMENT_ID	The *statement identifier* is an integer value that identifies the prepared statement that most recently executed. The **mi_get_id( )** function validates the connection that *conn* identifies and raises an error if the connection is invalid.
MI_SESSION_ID	The *session identifier* is an integer value that identifies the current client session. In a client LIBMI application, the session identifier corresponds to a connection. The **mi_get_id( )** function does *not* validate the connection that *conn* identifies.

The **mi_get_id( )** function is useful when you are debugging a DataBlade API module. You can use it with the MI_STATEMENT_ID argument to determine which prepared statement last executed. When you have many sessions that write to the same trace file, you can use **mi_get_id( )** with the MI_SESSION_ID argument to identify the session.

Return Values

0	indicates that no statement or session identifier currently exists.
>0	is a one- or two-digit number that represents the session or statement identifier.

The function raises an exception if *id_type* is an invalid type.

Related Topics

For more information on statement identifiers, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_get_int8( )

The **mi_get_int8( )** function copies an **mi_int8** (INT8) value, converting any difference in alignment or byte order on the client computer to that of the server computer.

Syntax

```
mi_unsigned_char1 *mi_get_int8(data_ptr, int8_ptr)
   mi_unsigned_char1 *data_ptr;
   mi_int8 *int8_ptr;
```

data_ptr	is a pointer to the buffer from which to copy the **mi_int8** value.
int8_ptr	is a pointer to the buffer to which to copy the **mi_int8** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_get_int8( )** function copies a value from the user-defined buffer that *data_ptr* references into the buffer that *int8_ptr* references. Upon completion, **mi_get_int8( )** returns the address of the next position from which data can be copied from the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *n* bytes, ready for a subsequent copy. In other words, if *n* is the length of the value that *int8_ptr* identifies, the returned address is *n* bytes advanced from the original buffer address in *data_ptr*.

The **mi_get_int8( )** function is useful in a receive support function of an opaque data type that contains an **mi_int8** value. Use this function to receive an **mi_int8** field of an opaque-type internal structure from a client application (which possibly has unaligned data buffers).

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_bytes( ), mi_get_date( ), mi_get_datetime( ), mi_get_decimal( ), mi_get_double_precision( ), mi_get_integer( ), mi_get_interval( ), mi_get_lo_handle( ), mi_get_money( ), mi_get_real( ), mi_get_smallint( ), mi_get_string( ),** and **mi_put_int8( ).**

For more information on the use of **mi_get_int8( )** in a receive support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_get_integer( )

The **mi_get_integer( )** function copies an **mi_integer** (INTEGER) value, converting any difference in alignment or byte order on the client computer to that of the server computer.

Syntax

```
mi_unsigned_char1 *mi_get_integer (data_ptr, int_ptr)
   mi_unsigned_char1 *data_ptr;
   mi_integer *int_ptr;
```

data_ptr	is a pointer to the buffer from which to copy the **mi_integer** value.
int_ptr	is a pointer to the buffer to which to copy the **mi_integer** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_get_integer( )** function copies a value from the user-defined buffer that *data_ptr* references into the buffer that *int_ptr* references. Upon completion, **mi_get_integer( )** returns the address of the next position from which data can be copied from the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *n* bytes, ready for a subsequent copy. In other words, if *n* is the length of the value that *int_ptr* identifies, the returned address is advanced *n* bytes from the original buffer address in *data_ptr*.

The **mi_get_integer( )** function is useful in a receive support function of an opaque data type that contains an **mi_integer** value. Use this function to receive an **mi_integer** field of an opaque-type internal structure from a client application (which possibly has unaligned data buffers).

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_fix_integer( ), mi_get_bytes( ), mi_get_date( ), mi_get_datetime( ), mi_get_decimal( ), mi_get_double_precision( ), mi_get_int8( ), mi_get_interval( ), mi_get_lo_handle( ), mi_get_money( ), mi_get_real( ), mi_get_smallint( ), mi_get_string( ),** and **mi_put_integer( ).**

For more information on the use of **mi_get_integer( )** in a receive support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_get_interval( )

The **mi_get_interval( )** function copies an **mi_interval** (INTERVAL) value, converting any difference in alignment or byte order on the client computer to that of the server computer.

Syntax

```
mi_unsigned_char1 *mi_get_interval(data_ptr, intrvl_ptr)
   mi_unsigned_char1 *data_ptr;
   mi_interval *intrvl_ptr;
```

data_ptr	is a pointer to the buffer from which to copy the **mi_interval** value.
intrvl_ptr	is a pointer to the buffer to which to copy the **mi_interval** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_get_interval( )** function copies a value from the user-defined buffer that *data_ptr* references into the buffer that *intrvl_ptr* references. Upon completion, **mi_get_interval( )** returns the address of the next position from which data can be copied from the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *n* bytes, ready for a subsequent copy. In other words, if *n* is the length of the value that *intrvl_ptr* identifies, the returned address is *n* bytes advanced from the original buffer address in *data_ptr*.

The **mi_get_interval( )** function is useful in a receive support function of an opaque data type that contains an **mi_interval** value. Use this function to receive an **mi_interval** field of an opaque-type internal structure from a client application (which possibly has unaligned data buffers).

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_bytes( ), mi_get_date( ), mi_get_datetime( ), mi_get_decimal( ), mi_get_double_precision( ), mi_get_int8( ), mi_get_integer( ), mi_get_lo_handle( ), mi_get_money( ), mi_get_real( ), mi_get_smallint( ), mi_get_string( ),** and **mi_put_interval( ).**

For more information on the use of **mi_get_interval( )** in a receive support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_get_lo_handle( )

The **mi_get_lo_handle( )** function copies an LO handle, converting any difference in alignment or byte order on the client computer to that of the server computer.

Syntax

```
mi_unsigned_char1 *mi_get_lo_handle(data_ptr, LO_hdl)
   mi_unsigned_char1 *data_ptr;
   MI_LO_HANDLE *LO_hdl;
```

data_ptr	is a pointer to the buffer from which to copy the LO handle.
LO_hdl	is a pointer to the LO handle to which to copy the buffer value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_get_lo_handle( )** function copies a value from the user-defined buffer that *data_ptr* references into a new smart large object that LO*_hdl* references. It is a constructor function for an LO handle. The function allocates a new LO handle in the current memory duration.

Upon completion, **mi_get_lo_handle( )** returns the address of the next position from which data can be copied from the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *n* bytes, ready for a subsequent copy. In other words, if *n* is the length of the LO handle that LO*_hdl* identifies, the returned address is *n* bytes advanced from the original buffer address in *data_ptr*.

The **mi_get_lo_handle( )** function is useful in a receive support function of an opaque data type that contains a smart large object. Use this function to receive an LO-handle field of an opaque-type internal structure from a client application (which possibly has unaligned data buffers).

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_bytes( ), mi_get_date( ), mi_get_datetime( ), mi_get_decimal( ), mi_get_double_precision( ), mi_get_int8( ), mi_get_integer( ), mi_get_interval( ), mi_get_money( ), mi_get_real( ), mi_get_smallint( ), mi_get_string( ),** and **mi_put_lo_handle( ).**

For more information on the use of **mi_get_lo_handle( )** in a receive support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_get_memptr_duration( )

The **mi_get_memptr_duration( )** function returns the memory duration of a specified pointer.

Syntax

```
MI_MEMORY_DURATION
mi_get_memptr_duration(void * memptr)
```

memptr	is the specified pointer.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_get_memptr_duration( )** function lets you determine the memory duration of memory allocated with one of the `mi_*` functions.

Return Values

md	is the memory duration of the specified pointer.

Related Topics

See also the description of **mi_get_duration_size( ).**

mi_get_money( )

The **mi_get_money( )** function copies an **mi_money** (MONEY) value, converting any difference in alignment or byte order on the client computer to that of the server computer.

Syntax

```
mi_unsigned_char1 *mi_get_money(data_ptr, money_ptr)
   mi_unsigned_char1 *data_ptr;
   mi_money *money_ptr;
```

data_ptr	is a pointer to the buffer from which to copy the **mi_money** value.
money_ptr	is a pointer to the buffer to which to copy the **mi_money** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_get_money( )** function copies a value from the user-defined buffer that *data_ptr* references into the buffer that *money_ptr* references. Upon completion, **mi_get_money( )** returns the address of the next position from which data can be copied from the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *n* bytes, ready for a subsequent copy. In other words, if *n* is the length of the value that *money_ptr* identifies, the returned address is *n* bytes advanced from the original buffer address in *data_ptr*.

The **mi_get_money( )** function is useful in a receive support function of an opaque data type that contains an **mi_money** value. Use this function to receive an **mi_money** field of an opaque-type internal structure from a client application (which possibly has unaligned data buffers).

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_bytes( ), mi_get_date( ), mi_get_datetime( ), mi_get_decimal( ), mi_get_double_precision( ), mi_get_int8( ), mi_get_integer( ), mi_get_interval( ), mi_get_lo_handle( ), mi_get_real( ), mi_get_smallint( ), mi_get_string( ),** and **mi_put_money( ).**

For more information on the use of **mi_get_money( )** in a receive support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_get_next_sysname( )

The **mi_get_next_sysname( )** function retrieves the next database server name.

Syntax

```
mi_integer mi_get_next_sysname(name_ptr, name_buf, name_len)
   mi_integer *name_ptr;
   char *name_buf;
   mi_integer name_len;
```

name_ptr	is a pointer to the next database server name.
name_buf	is a pointer to the buffer to contain the next database server name.
name_len	is the size of the *name_buf* buffer.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	No

Usage

The **mi_get_next_sysname( )** function obtains the next database server name and puts it into the buffer that *name_buf* references. It returns the length of this name in the *name_len* argument.

UNIX/Linux Only

On UNIX or Linux, the **sqlhosts** file defines system names.

End of UNIX/Linux Only

Windows Only

On Windows, the Registry defines system names.

End of Windows Only

The following code fragment uses **mi_get_next_sysname( )** to display the contents of the server-definition file:

```
main( )
{
   mi_integer handle;
   char nameb[32];

   handle = 0;
   while( mi_get_next_sysname(&handle, nameb, sizeof(nameb) )
      puts(nameb);
   return 0;
}
```

The **mi_get_next_sysname( )** function initializes the DataBlade API when it is the first DataBlade API function in a client LIBMI application.

Return Values

MI_TRUE	indicates that the function was successful.
MI_FALSE	indicates that the function was *not* successful.

Related Topics

See also the description of **mi_sysname( ).**

mi_get_parameter_info( )

The **mi_get_parameter_info( )** function populates a parameter-information descriptor with the current session parameters.

Syntax

```
mi_integer mi_get_parameter_info (parameter_info)
   MI_PARAMETER_INFO *parameter_info;
```

parameter_info	is a pointer to a user-provided parameter-information descriptor to store the current session parameters.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_get_parameter_info( )** function returns current session parameters into the parameter-information descriptor (**MI_PARAMETER_INFO** structure) that *parameter_info* references. The parameter information structure controls whether callbacks are enabled and whether pointers are checked during the session.

Tip: You must allocate this parameter-information descriptor *before* you call **mi_get_parameter_info( )**.

The **mi_get_parameter_info( )** function also initializes the DataBlade API when it is the first DataBlade API function in a client LIBMI application.

To set the session parameters, use the **mi_set_parameter_info( )** function.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_connection_info( ), mi_get_database_info( ), mi_get_serverenv( ),** and **mi_set_parameter_info( ).**

For more information on the parameter-information descriptor, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_get_real( )

The **mi_get_real( )** function copies an **mi_real** (SMALLFLOAT) value, converting any difference in alignment or byte order on the client computer to that of the server computer.

Syntax

```
mi_unsigned_char1 *mi_get_real(data_ptr, real_ptr)
   mi_unsigned_char1 *data_ptr;
   mi_real *real_ptr;
```

data_ptr	is a pointer to the buffer from which to copy the **mi_real** value.
real_ptr	is a pointer to the buffer to which to copy the **mi_real** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_get_real( )** function copies a value from the user-defined buffer that *data_ptr* references into the buffer that *real_ptr* references. Upon completion, **mi_get_real( )** returns the address of the next position from which data can be copied from the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *n* bytes, ready for a subsequent copy. In other words, if *n* is the length of the value that *real_ptr* identifies, the returned address is *n* bytes advanced from the original buffer address in *data_ptr*.

The **mi_get_real( )** function is useful in a receive support function of an opaque data type that contains an **mi_real** value. Use this function to receive an **mi_real** field of an opaque-type internal structure from a client application (which possibly has unaligned data buffers).

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_bytes( ), mi_get_date( ), mi_get_datetime( ), mi_get_decimal( ), mi_get_double_precision( ), mi_get_int8( ), mi_get_integer( ), mi_get_interval( ), mi_get_lo_handle( ), mi_get_money( ), mi_get_smallint( ), mi_get_string( ),** and **mi_put_real( ).**

For more information on the use of **mi_get_real( )** in a receive support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_get_result( )

The **mi_get_result( )** function returns the status of the current statement.

Syntax

```
mi_integer mi_get_result(conn)
   MI_CONNECTION *conn;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_get_result( )** function provides the statement status for the current statement on the connection that *conn* references. The current statement is the most recently executed SQL statement sent to the database server on that connection. One of the DataBlade API statement-execution functions sent the statement to the database server: **mi_exec( ), mi_exec_prepared_statement( ), or mi_open_prepared_statement( ).**

The **mi_get_result( )** function classifies SQL statements as follows.

Type of SQL Statement	Statement-Status Constant
Statement is a data manipulation language (DML) statement (DELETE, INSERT, UPDATE)	MI_DML
Statement is a data definition language (DDL) statement (such as ALTER TABLE, CREATE TABLE, or DROP TABLE)	MI_DDL
Statement produced rows of data that the DataBlade API program must fetch (SELECT or EXECUTE FUNCTION)	MI_ROWS, MI_DML

The **mi_get_result( )** function is typically executed in a loop that terminates when this function returns MI_NO_MORE_RESULTS. If **mi_get_result( )** returns MI_ROWS, a query has executed and a cursor is ready to be accessed by the **mi_next_row( )** function.

Return Values

MI_NO_MORE_RESULTS	indicates no more results pending.
MI_ROWS	indicates that the query has executed successfully and the cursor is available.
MI_DML	indicates that a Data Manipulation Language (DML) statement has completed.
MI_DDL	indicates that a Data Definition Language (DDL) statement has completed.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_exec( ), mi_exec_prepared_statement( ), mi_fetch_statement( ), mi_next_row( ), mi_open_prepared_statement( ), mi_result_command_name( ),** and **mi_result_row_count( ).**

For more information about how to obtain results, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_get_row_desc( )

The **mi_get_row_desc( )** function obtains a pointer to the row descriptor for the specified row.

Syntax

```
MI_ROW_DESC *mi_get_row_desc(row)
   MI_ROW *row;
```

row	is a pointer to the row for which to obtain a descriptor.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_get_row_desc( )** function returns a row descriptor for the row structure that *row* references. This function is useful for processing returned row data, especially when not all the rows that a query returns have the same size and structure.

The row descriptor that **mi_get_row_desc( )** returns is valid while the row it came from is valid. For information about row validity, see the description of the **mi_next_row( )** function.

Return Values

An **MI_ROW_DESC** pointer	is a pointer to the row descriptor for the row.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_row_desc_from_type_desc( ), mi_get_row_desc_without_row( ),** and **mi_get_statement_row_desc( ).**

mi_get_row_desc_from_type_desc( )

The **mi_get_row_desc_from_type_desc( )** function obtains a row descriptor that is associated with the specified type descriptor.

Syntax

```
MI_ROW_DESC *mi_get_row_desc_from_type_desc(type_desc)
   MI_TYPE_DESC *type_desc;
```

type_desc	is a pointer to the type descriptor.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_get_row_desc_from_type_desc( )** function is useful for processing returned row data, especially when not all the rows that a query returns have the same size and structure.

Return Values

An **MI_ROW_DESC** pointer	is a pointer to the row descriptor for the specified type descriptor.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_row_desc( ), mi_get_row_desc_without_row( ),** and **mi_get_statement_row_desc( ).**

For more information on how to obtain row descriptors or on type descriptors, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_get_row_desc_without_row( )

The **mi_get_row_desc_without_row( )** function obtains the row descriptor for the current statement.

Syntax

```
MI_ROW_DESC *mi_get_row_desc_without_row(conn)
   MI_CONNECTION *conn;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_get_row_desc_without_row( )** function obtains the row descriptor for the current statement on the connection that *conn* references. The current statement is the most recently executed SQL statement sent to the database server on that connection.

Use **mi_get_row_desc_without_row( )** to obtain a row descriptor for a query when the current statement was sent with the **mi_exec( )** function. The **mi_get_row_desc_without_row( )** function is useful when all the rows that the query returns are of the same type or when the statement is expected to return row data but no rows qualified.

The row descriptor is valid until one of the following conditions occurs:
- The query finishes.
- The **mi_close( )** function is called on the connection.

Important: Do not use the **mi_row_desc_free( )** function to free the row descriptor that **mi_get_row_desc_without_row( )** allocates. Use **mi_row_desc_free( )** only for row descriptors that you allocate with **mi_row_desc_create( ).**

After you obtain a row descriptor for the current statement, you can obtain the number of columns in the row with the **mi_column_count( )** function. The number of columns determines the number of times to call the **mi_value( )** or **mi_value_by_name( )** function to obtain column values.

Return Values

An **MI_ROW_DESC** pointer	is a pointer to the row descriptor for the last query that returned rows on the specified connection.
NULL	indicates that the function was not successful or if the current statement is not a statement that returns rows.

Related Topics

See also the descriptions of **mi_column_count( ), mi_close( ), mi_get_row_desc_from_type_desc( ),** and **mi_get_statement_row_desc( ).**

mi_get_serverenv( )

The **mi_get_serverenv( )** function obtains information from the server environment.

Syntax

```
mi_integer mi_get_serverenv(name, value)
   const char *name;
   char **value;
```

name	is a pointer to the name of the server-environment information to obtain.
value	is a pointer to the value to obtain from the server-environment information that *name* references.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_get_serverenv( )** function obtains a copy of the value assigned to the server-environment information that *name* references. The server environment describes the environment of the database server in which the UDR executes. The *name* buffer can contain any of the following server-environment information:

- The name of an Informix environment variable
- The name of a configuration parameter in the database server ONCONFIG file

Tip: The **mi_get_serverenv( )** function is similar in functionality to the operating-system call **getenv( )**, except that **mi_get_serverenv( )** can only retrieve the value of an Informix environment variable.

If the item in the *name* buffer has multiple occurrences in the server environment, **mi_get_serverenv( )** returns the first occurrence from the following list:

Windows Only

1. The **InetLogin** login structure
2. The Windows Registry

End of Windows Only

3. The ONCONFIG file (for configuration parameters)
4. The default value (if one exists)

Global Language Support

The **mi_get_serverenv( )** function performs any necessary code-set conversion on the values of server-environment information from the client code set to the code set of the current processing locale.

End of Global Language Support

The **mi_get_serverenv( )** function copies the current value into the buffer that *value* references. The function allocates memory for the *value* buffer in the current memory duration. When you no longer need this buffer, free this memory. If the *name* buffer specifies some entity that is undefined in the server environment, the **mi_get_serverenv( )** function takes the following actions:

- It sets *value* to a NULL-valued pointer.
- It returns a value of MI_OK.

Return Values

MI_OK	indicates that the function was successful or the specified server-environment information is undefined.
MI_ERROR	indicates that the function was not successful. The arguments are not valid pointers or memory cannot be allocated for the value that *name* references.

Related Topics

See also the descriptions of **mi_get_connection_info( ), mi_get_connection_option( ), mi_get_database_info( ), mi_get_parameter_info( ), mi_set_connection_user_data( ), mi_set_default_connection_info( ), mi_set_default_database_info( ),** and **mi_set_parameter_info( ).**

mi_get_session_connection( )

The **mi_get_session_connection( )** function obtains a session-duration connection descriptor.

Syntax

```
MI_CONNECTION *mi_get_session_connection( )
```

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Warning: This advanced function can adversely affect your UDR if you use the function incorrectly. Use it only when no regular DataBlade API function can perform the task you need done.

Usage

The **mi_get_session_connection( )** function obtains a session-duration connection, which provides access to the connection for the session (which the client application begins). A UDR connection that **mi_open( )** establishes is private to the UDR; that is, it is valid until the UDR completes. A session-duration connection is valid until the end of the session.

This function is a constructor function for a session-duration connection descriptor, although it does not actually allocate a new connection descriptor. Instead, it obtains a copy of the session context for the client application and stores it in PER_SESSION memory.

Tip: You can use the session-duration connection descriptor to obtain session-duration function descriptors. For more information, see the *IBM Informix DataBlade API Programmer's Guide*.

The following restrictions apply to the use of **mi_get_session_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.

Return Values

An **MI_CONNECTION** pointer	is a pointer to the session-duration connection descriptor for the session.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_cast_get( ), mi_close( ), mi_func_desc_by_typeid( ), mi_open( ), mi_routine_get( ), mi_routine_get_by_typeid( ),** and **mi_td_cast_get( ).**

mi_get_smallint( )

The **mi_get_smallint( )** function copies an **mi_smallint** (SMALLINT) value, converting any difference in alignment or byte order on the client computer to that of the server computer.

Syntax

```
mi_unsigned_char1 *mi_get_smallint (data_ptr, smallint_ptr)
   mi_unsigned_char1 *data_ptr;
   mi_smallint *smallint_ptr;
```

data_ptr	is the address of the buffer from which to copy the *promoted* **mi_smallint** value.
smallint_ptr	is a pointer to the buffer to which to copy the **mi_smallint** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_get_smallint( )** function copies a value from the user-defined buffer that *data_ptr* references into the buffer that *smallint_ptr* references. Upon completion, **mi_get_smallint( )** returns the address of the next position from which data can be copied from the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *n* bytes, ready for a subsequent copy. In other words, if *n* is the length of the value that *smallint_ptr* identifies, the returned address is *n* bytes advanced from the original buffer address in *data_ptr*.

For maximum portability, this function accepts a fully promoted **mi_integer** value instead of an **mi_smallint** value. This argument might therefore require casting.

The **mi_get_smallint( )** function is useful in a receive support function of an opaque data type that contains an **mi_smallint** value. Use this function to obtain an **mi_smallint** field of an opaque-type internal structure from a client application (which possibly has unaligned data buffers).

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_fix_smallint( ), mi_get_bytes( ), mi_get_date( ), mi_get_datetime( ), mi_get_decimal( ), mi_get_double_precision( ), mi_get_int8( ), mi_get_integer( ), mi_get_interval( ), mi_get_lo_handle( ), mi_get_money( ), mi_get_real( ), mi_get_string( ),** and **mi_put_smallint( ).**

For more information on the use of **mi_get_smallint( )** in a receive support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_get_statement_row_desc( )

The **mi_get_statement_row_desc( )** function obtains the row descriptor for a prepared statement.

Syntax

```
MI_ROW_DESC *mi_get_statement_row_desc(stmt_desc)
   MI_STATEMENT *stmt_desc;
```

stmt_desc	is a pointer to the statement descriptor for the prepared statement.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_get_statement_row_desc( )** function obtains the row descriptor from the statement descriptor that *stmt_desc* references. This row descriptor contains information for the columns in the prepared statement. From this row descriptor, you can use the row-accessor functions to obtain information about the prepared-statement columns.

The row descriptor is valid until one of the following conditions occurs:

- The query finishes.
- The **mi_close( )** function is called on the connection.

Important: Do not use the **mi_row_desc_free( )** function to free the row descriptor that **mi_get_statement_row_desc( )** allocates. Use **mi_row_desc_free( )** only for row descriptors that you allocate with **mi_row_desc_create( ).**

Return Values

An **MI_ROW_DESC** pointer	is a pointer that corresponds to the input statement.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_column_count( ), mi_column_id( ), mi_column_name( ), mi_column_nullable( ), mi_column_precision( ), mi_column_scale( ), mi_column_type_id( ), mi_column_typedesc( ), mi_get_row_desc( ), mi_get_row_desc_from_type_desc( ), mi_get_row_desc_without_row( ),** and **mi_prepare( ).**

mi_get_string( )

The **mi_get_string( )** function copies an **mi_string** (CHAR(x)) value from a buffer.

Syntax

```
mi_unsigned_char1 *mi_get_string(data_ptr, string_dptr, srcbytes)
   mi_unsigned_char1 *data_ptr;
   mi_string **string_dptr;
   mi_integer srcbytes;
```

data_ptr	is a pointer to the buffer from which to copy the **mi_string** value.
string_dptr	is a pointer to the buffer to which to copy the **mi_string** value.
srcbytes	is the number of source bytes to copy.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_get_string( )** function copies *srcbytes* bytes from the user-defined buffer that *data_ptr* references into the buffer that *string_ptr* references. Upon completion, **mi_get_string( )** returns the address of the next position from which data can be copied from the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *srcbytes* bytes, ready for a subsequent copy. In other words, if *srcbytes* is the length of the value in *string_buf*, the returned address is *srcbytes* bytes advanced from the original buffer address in *data_dptr*.

The **mi_get_string( )** function is useful in a receive support function of an opaque data type that contains an **mi_string** value. Use this function to receive an **mi_string** field of an opaque-type internal structure from a client application (which possibly has unaligned data buffers).

Global Language Support

If code-set conversion is required, the **mi_get_string( )** function converts the **mi_string** value from the code set of the client locale to that of the serverprocessing locale. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Tip: While other **mi_get** functions accept a preallocated buffer, **mi_get_string( )** allocates memory for data to be copied. This allocation is why the function accepts a pointer to the address as the buffer argument.

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_bytes( ), mi_get_date( ), mi_get_datetime( ), mi_get_decimal( ), mi_get_double_precision( ), mi_get_int8( ), mi_get_integer( ), mi_get_interval( ), mi_get_lo_handle( ), mi_get_money( ), mi_get_real( ), mi_get_smallint( ),** and **mi_put_string( ).**

For more information on the use of **mi_get_string( )** in a receive support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_get_type_source_type( )

The **mi_get_type_source_type( )** function obtains a type descriptor for the source of a distinct type.

Syntax

```
MI_TYPE_DESC *mi_get_type_source_type(typedesc_ptr)
   MI_TYPE_DESC *typedesc_ptr;
```

typedesc_ptr	is a pointer to a type descriptor of the distinct type.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_get_type_source_type( )** function obtains a type descriptor for the source type of the data type that *typedesc_ptr* references.

Return Values

An **MI_TYPE_DESC** pointer	is a pointer to a type descriptor for the source type of the specified distinct type.
NULL	indicates that the function was not successful.

mi_get_transaction_id( )

The **mi_get_transaction_id( )** function obtains the current internal transaction ID.

Syntax

```
mi_integer mi_get_transaction_id(void);
```

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_get_transaction_id( )** function obtains the internal transaction ID of the current transaction. This transaction ID is not the same as the thread ID or the session ID. If **mi_get_transaction_id( )** is called while not inside a transaction or with a non-logging database, it returns MI_ERROR. The value of the MI_ERROR -1.

Return Values

>0	is a transaction ID.
-1	indicates no transaction

mi_get_vardata( )

The **mi_get_vardata( )** accessor function returns a pointer to the data contained in a varying-length structure (such as **mi_lvarchar**).

Syntax

```
char *mi_get_vardata(varlen_ptr)
   mi_lvarchar *varlen_ptr;
```

varlen_ptr	is a pointer to a varying-length structure from which to retrieve the data.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_get_vardata( )** function obtains a pointer to the data portion of the varying-length structure that *varlen_ptr* references.

Important: The varying-length structure that "*varlen_ptr*" references is an opaque structure. Do not access the fields of this structure directly. Instead, use **mi_get_vardata( )** to obtain the data pointer from this structure.

The data in a varying-length structure is *not* null terminated. Do *not* use null termination to determine end of the data that the **mi_get_vardata( )** returns. Instead, use the **mi_get_varlen( )** function to obtain the actual data length, which you can then use to access the varying-length data.

Although the *varlen_ptr* argument is declared as a pointer to an **mi_lvarchar** structure, you can also use the **mi_get_vardata( )** function to obtain data from other varying-length data types, such as **mi_sendrecv**.

Return Values

A **char** pointer	is a pointer to the data in the varying-length structure.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_vardata_align( ), mi_get_varlen( ), mi_new_var( ), mi_set_vardata( ),** and **mi_var_free( ).**

mi_get_vardata_align( )

The **mi_get_vardata_align( )** accessor function obtains a pointer to the data in a varying-length structure (such as **mi_lvarchar**) and adjusts for any initial padding required to align the data.

Syntax

```
char *mi_get_vardata_align(varlen_ptr, align)
   mi_lvarchar *varlen_ptr,
   mi_integer align;
```

varlen_ptr	is a pointer to a variable-length structure.
align	is the alignment boundary value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_get_vardata_align( )** function aligns the data on the nearest *align*-byte boundary. The **mi_get_vardata_align( )** function is useful for data types whose alignment is not on a 4-byte boundary, such as arrays that are stored in varying-length structures. Array elements might have more stringent alignment requirements than the 4-byte alignment guaranteed by the varying-length structure. For opaque data types, this value must match the **align** column of the **sysxtdtypes** system catalog table.

Important: The varying-length structure that "*varlen_ptr*" references is an opaque structure. Do not access the fields of this structure directly. Instead, use **mi_get_vardata_align( )** to obtain the data from this structure in an aligned format.

The data in a varying-length structure is *not* null terminated. Do *not* use null termination to determine end of the data that the **mi_get_vardata_align( )** returns. Instead, use the **mi_get_varlen( )** function to obtain the actual data length, which you can then use to access the varying-length data.

Although the *varlen_ptr* argument is declared as a pointer to an **mi_lvarchar**, you can also use the **mi_get_vardata_align( )** function to obtain aligned data from other varying-length data types, such as **mi_sendrecv**.

Return Values

A **char** pointer	is a pointer to the data in the varying-length structure.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_vardata( ), mi_get_varlen( ), mi_new_var( ), mi_get_vardata_align( ),** and **mi_var_free( ).**

mi_get_varlen( )

The **mi_get_varlen( )** accessor function returns the length of the data stored in a varying-length structure (such as **mi_lvarchar**).

Syntax

```
mi_integer mi_get_varlen(varlen_ptr)
   mi_lvarchar *varlen_ptr;
```

varlen_ptr	is a pointer to a varying-length structure.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_get_varlen( )** function obtains the length of the varying-length data from the varying-length structure that *varlen_ptr* references. This returned length is the actual length of the varying-length structure. It does not include the length of the other fields of the varying-length structure.

Important: The varying-length structure that "*varlen_ptr*" references is an opaque structure. Do not access the fields of this structure directly. Instead, use **mi_get_varlen( )** to obtain the data length from this structure.

The data in a varying-length structure is *not* null terminated. Use the **mi_get_varlen( )** function to obtain the data length, which you can then use to access the varying-length data.

Although the *varlen_ptr* argument is declared as a pointer to an **mi_lvarchar**, you can also use the **mi_get_varlen( )** function to obtain data length from other varying-length data types, such as **mi_sendrecv**.

Return Values

>=0	is the length of the data in the variable-length structure.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_vardata( ), mi_get_vardata_align( ), mi_new_var( ), mi_set_varlen( ),** and **mi_var_free( ).**

mi_hdr_status( )

The **mi_hdr_status( )** function returns the high-availability cluster replication status of the current server.

Syntax

```
mi_integer mi_hdr_status(void)
```

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_hdr_status( )** function returns information about:

- the server type, such as primary server, SD secondary server, RS secondary server, or HDR secondary server,
- HDR status,
- the ability to update data on a secondary server (also known as proxy writes).

When C UDRs are executing as a part of SELECT, execute function, or execute procedure statements, they can call **mi_hdr_status( )** to determine the mode of the current server.

The return value of **mi_hdr_status( )** is of type mi_integer, which is interpreted as a bitmap. The meaning of each bit is defined in the **milib.h** file.

Return Values

MI_HDR_ON
indicates that the HDR environment is configured and is working.

MI_HDR_PRIMARY
indicates that the current server is a primary server.

MI_HDR_SECONDARY
indicates whether the server is any type of secondary server. Current secondary server types include HDR, SD secondary, and RS secondary servers. This return code has been preserved for compatibility with earlier versions. The return code MI_SECONDARY has the same return code as MI_HDR_SECONDARY. Use MI_SECONDARY instead of MI_HDR_SECONDARY for new application development. Use MI_HDR_SEC_NODE to determine whether the server is an HDR secondary server.

MI_SECONDARY
indicates that the current server is a secondary server.

MI_HDR_SEC_NODE
indicates that the current server is an HDR secondary server.

MI_RSS_SECONDARY
indicates that the current server is an RS (Remote Standalone) secondary server.

MI_SDS_SECONDARY
indicates that the current server is an SD (Shared Disk) secondary server.

MI_UPDATABLE_SECONDARY
indicates that the current server is a secondary server that is configured to accept updates.

Related Topics

For more information about High-Availability Data Replication and high-availability cluster environments, see the *IBM Informix Dynamic Server Administrator's Guide*.

mi_init_library( )

The **mi_init_library( )** function initializes the DataBlade API library.

Syntax

```
mi_integer mi_init_library(flags)
   mi_integer flags;
```

flags	should be `0`.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	No

Usage

The **mi_init_library( )** function initializes the DataBlade API when it is the first DataBlade API function in a client LIBMI application.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_client_locale( ), mi_get_default_connection_info( ), mi_get_default_database_info( ), mi_get_next_sysname( ), mi_get_parameter_info( ), mi_open( ), mi_register_callback( ), mi_server_connect( ), mi_set_default_connection_info( ), mi_set_default_database_info( ), mi_set_parameter_info( ),** and **mi_sysname( ).**

mi_interrupt_check( )

The **mi_interrupt_check( )** function checks for a user interrupt.

Syntax

```
mi_integer mi_interrupt_check( )
```

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_interrupt_check( )** function determines whether the client application has interrupted the current operation by sending a break signal to the database server. When **mi_interrupt_check( )** returns a nonzero value, the C UDR returns an appropriate error.

Return Values

0	indicates that the client application did *not* interrupt the current operation.
<> 0	indicates that the client application interrupted the current operation.

mi_interval_to_string( )

The **mi_interval_to_string( )** function creates an ANSI SQL standards text (string) representation of an interval value from its binary (internal) INTERVAL representation.

Syntax

```
mi_string *mi_interval_to_string(intvl_data)
   mi_interval *intvl_data;
```

intvl_data	is a pointer to the internal INTERVAL representation of the interval value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_interval_to_string( )** function converts the internal INTERVAL value that *intvl_data* references into an interval string. The interval string has the following ANSI SQL standards format:

```
"YYYY-MM-DD HH:mm:SS.FFFFF"
```

YYYY	is the 4-digit year.
MM	is the 2-digit month.
DD	is the 2-digit day.
HH	is the 2-digit hour.
mm	is the 2-digit minute.
SS	is the 2-digit second.
FFFFF	is the fraction of a second, in which the date, time, or date and time qualifier specifies the number of digits, with a maximum precision of 5 digits.

If the internal INTERVAL value contains only a subset of this range, **mi_interval_to_string( )** creates an interval string with the appropriate portion of the preceding format. For example, suppose *intvl_data* references the internal format of the interval *6 days, 5 hours, and 45 minutes*. The **mi_interval_to_string( )** function returns an **mi_string** value with the following interval string:

```
"06 05:45"
```

Global Language Support

The **mi_interval_to_string( )** function does *not* format the interval string in the date and time formats of the current processing locale.

End of Global Language Support

Return Values

An **mi_string** pointer	is a pointer to the interval string equivalent to *intvl_data*.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_date_to_string( ), mi_decimal_to_string( ), mi_datetime_to_string( ), mi_money_to_string( ),** and **mi_string_to_interval( ).**

For more information on how to convert internal INTERVAL format to interval strings, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_issmall_data( )

The **mi_issmall_data( )** macro determines whether storage of data is or is not in a smart large object.

Syntax

```
mi_boolean mi_issmall_data(size)
   MI_MULTIREP_SIZE size;
```

size	is the value of the threshold-tracking field in the internal representation of a multirepresentational opaque type. This field is set to either MI_MULTIREP_SMALL or MI_MULTIREP_LARGE.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_issmall_data( )** macro checks the *size* value and sets the return value to indicate whether data is or is not stored in a smart large object.

Return Values

MI_TRUE	indicates that data is not stored in a smart large object.
MI_FALSE	indicates that data is stored in a smart large object.

Related Topics

See also the description of **mi_set_large( ).**

mi_last_serial( )

The **mi_last_serial( )** function returns a SERIAL value that the database server has generated for the most recent INSERT statement on a SERIAL column.

Syntax

```
mi_integer mi_last_serial(conn, serial_val)
   MI_CONNECTION *conn;
   mi_integer *serial_val;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
serial_val	is a pointer to the new SERIAL value to obtain.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

The database server automatically generates a value for a SERIAL column when it executes an INSERT statement on the column.

Server Only

Use the **mi_last_serial( )** function to obtain the last system-generated SERIAL value for your C user-defined routine.

End of Server Only

Client Only

To use the **mi_last_serial( )** function in a client LIBMI application, call it after the insert operation is complete (when the **mi_get_result( )** function returns the MI_DML statement).

End of Client Only

You must also call **mi_last_serial( )** *before* any call to **mi_query_finish( )** or **mi_close( ).**

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_get_result( )** and **mi_last_serial8( ).**

For more information, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_last_serial8( )

The **mi_last_serial8( )** function obtains the SERIAL8 value that the database server generated for the most recent INSERT statement on a SERIAL8 column.

Syntax

```
mi_integer mi_last_serial8(conn, serial8_val)
   MI_CONNECTION *conn;
   mi_int8 *serial8_val;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
serial8_val	is a pointer to the new SERIAL8 value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The database server automatically generates a value for a SERIAL8 column when it executes an INSERT statement on the column.

Server Only

Use the **mi_last_serial8( )** function to obtain the last system-generated SERIAL8 value for your C user-defined routine.

End of Server Only

Client Only

To use the **mi_last_serial8( )** function in a client LIBMI application, call it after the insert operation is complete (when the **mi_get_result( )** function returns the MI_DML statement).

End of Client Only

You must also call **mi_last_serial8( )** *before* any call to **mi_query_finish( )** or **mi_close( ).**

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_get_result( )** and **mi_last_serial( ).**

mi_library_version( )

The **mi_library_version( )** function returns the version and version date of the DataBlade API currently being used.

Syntax

```
mi_integer mi_library_version(buf, buflen)
   char *buf;
   mi_integer buflen;
```

buf	is the user-allocated buffer for the version string.
buflen	is the length of *buf* in bytes.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_library_version( )** function copies the version and version date into the user-defined buffer that *buf* references.

Server Only

In a C UDR, **mi_library_version( )** returns the name of the database server and its version number.

Important: Beginning with Dynamic Server Version 10.0, use the new **mi_server_library_version( )** function to obtain the correct database server version. The **mi_library_version( )** function returns a string with the value ″IBM Informix Dynamic Server Version 9.50.UC1,″ which would still be applicable if your software uses string compare functionality.

End of Server Only

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_version_comparison( )** and **mi_server_library_version( )**.

mi_lo_alter( )

The **mi_lo_alter( )** function alters the storage characteristics of an existing smart large object.

Syntax

```
mi_integer mi_lo_alter(conn, LO_hdl, LO_spec)
   MI_CONNECTION *conn;
   MI_LO_HANDLE *LO_hdl;
   MI_LO_SPEC *LO_spec;
```

conn	is one of the following connection values:
	A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )**
	A NULL-valued pointer (database server only)
LO_hdl	is a pointer to an LO handle.
LO_*spec*	is a pointer to an LO-specification structure.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_alter( )** function updates the storage characteristics of an existing smart large object with the characteristics in the LO-specification structure that LO_*spec* references. With **mi_lo_alter( )**, you can change *only* the following storage characteristics:

- Logging characteristics

  You can specify the MI_LO_ATTR_LOG or MI_LO_ATTR_NO_LOG constant for the attributes flag with the **mi_lo_specget_flags( )** function.
- Last-access time characteristics

  You can specify the MI_LO_ATTR_KEEP_LASTACCESS_TIME or MI_LO_ATTR_NOKEEP_LASTACCESS_TIME constant for the attributes flag with the **mi_lo_specget_flags( )** function.
- Extent size

  You can store a new integer value for the allocation extent size with the **mi_lo_specset_extsz( )** function. The new extent size applies only to extents written *after* the **mi_lo_alter( )** function completes.
- Buffering mode

  You can use **mi_lo_alter( )** to alter the buffering mode of a smart large object that was created with light-weight I/O (MI_LO_NOBUFFER) to buffered I/O (MI_LO_BUFFER) as long as no open instances exist for that smart large object.

  However, **mi_lo_alter( )** generates an error if you attempt to change an open smart large object with buffered I/O to one with light-weight I/O.

The function obtains an exclusive lock for the entire smart large object before it proceeds with the update. It holds this lock until the update completes.

Server Only

In a C UDR, the **mi_lo_alter( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_lo_spec_init( ), mi_lo_colinfo_by_name( ),** and **mi_lo_colinfo_by_ids( ).**

mi_lo_close( )

The **mi_lo_close( )** function closes an open smart large object.

Syntax

```
mi_integer mi_lo_close(conn, LO_fd)
   MI_CONNECTION *conn;
   MI_LO_FD LO_fd;
```

conn	is one of the following connection values:
	A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )**
	A NULL-valued pointer (database server only)
LO_*fd*	is an LO file descriptor of the smart large object to close.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

The **mi_lo_close( )** function closes the smart large object that is associated with the LO file descriptor, LO_*fd*, then frees this file descriptor. This function is the destructor function for an LO file descriptor. A call to the **mi_open( ), mi_lo_copy( ), mi_lo_create( ), mi_lo_expand( ),** or **mi_lo_from_file( )** function returns an LO file descriptor for a smart large object. Once you free an LO file descriptor, you can reuse it for another smart large object. Any open smart large object that you do not explicitly close is automatically closed when the connection closes (when the client connection terminates or the C user-defined routine completes).

When the **mi_lo_close( )** function closes a smart large object, the database server attempts to unlock a locked smart large object if it has a share-mode or update-mode lock. For exclusive locks, the database server does not permit the release of the lock until the end of the transaction. The **mi_lo_close( )** function also deallocates any private-buffer memory that lightweight-I/O allocates.

Server Only

The **mi_lo_close( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful; the session is bad, or an argument is invalid.

Related Topics

See also the descriptions of **mi_lo_copy( ), mi_lo_create( ), mi_lo_expand( ), mi_lo_from_file( ),** and **mi_open( ).**

mi_lo_colinfo_by_ids( )

The **mi_lo_colinfo_by_ids( )** function sets the fields of an LO-specification structure to the column-level storage characteristics for specified row descriptor and column identifier.

Syntax

```
mi_integer mi_lo_colinfo_by_ids(conn, row_desc, column_id, LO_spec)
   MI_CONNECTION *conn;
   MI_ROW *row_desc;
   mi_integer column_id;
   MI_LO_SPEC *LO_spec;
```

conn	is one of the following connection values: A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )** A NULL-valued pointer (database server only)
row_desc	is a pointer to the row descriptor that contains the column.
column_id	is the integer column identifier of the column within the row structure, with the first column starting at offset `0`.
LO_*spec*	is a pointer to the LO-specification structure into which **mi_lo_colinfo_by_ids( )** is to put the storage characteristics for the specified column.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_lo_colinfo_by_ids( )** function sets the fields of the LO-specification structure that LO_*spec* references to the storage characteristics for a specified column. Identify this column with the following arguments:

- The *row_desc* argument references the row descriptor (the **MI_ROW** structure) that contains the column.
- The *column_id* argument is the offset of the column within the row descriptor that *row_desc* identifies.

If this specified column does *not* have column-level storage characteristics defined for it, the database server uses the storage characteristics that are inherited.

The **mi_lo_colinfo_by_ids( )** function is primarily intended for use within the **assign( )** or import support routines of an opaque data type that contains a smart large object. Within either of these support functions, the database server guarantees that the associated row and the column information can be determined from the **MI_FPARAM** structure with the accessor functions, **mi_fp_getrow( )** and **mi_fp_getcolid( ).** This function also works for any user-defined routine that is named or directly used in an INSERT, UPDATE, or SELECT statement. In all other contexts, the database server does *not* guarantee that the row and column information in the **MI_FPARAM** structure is valid.

Server Only

The **mi_lo_colinfo_by_ids( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

MI_OK	indicates that the function was successful; LO_*spec* points to the LO-specification structure with the storage characteristics of the specified column.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_fp_getcolid( ), mi_fp_getrow( ), mi_lo_colinfo_by_name( )** and **mi_lo_spec_init( ).**

mi_lo_colinfo_by_name( )

The **mi_lo_colinfo_by_name( )** function sets the fields of an LO-specification structure to the column-level storage characteristics for a specified database column.

Syntax

```
mi_integer mi_lo_colinfo_by_name(conn, column_spec, LO_spec)
   MI_CONNECTION *conn;
   const char *column_spec;
   MI_LO_SPEC *LO_spec;
```

conn	is one of the following connection values: A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )** A NULL-valued pointer (database server only)
column_spec	is a pointer to a buffer that contains the name of the database column whose column-level storage characteristics you want to use.
LO_*spec*	is a pointer to the LO-specification structure into which **mi_lo_colinfo_by_name( )** puts the storage characteristics for the *column_spec* column.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_colinfo_by_name( )** function sets the fields of the LO-specification structure that LO_*spec* references to the storage characteristics for the *column_spec* database column. If this specified column does not have column-level storage characteristics defined for it, the database server uses the storage characteristics that are inherited.

The *column_spec* buffer must specify the column name in the following format:

```
database@server_name:table.column
```

In the preceding format, the *database* and *server_name* arguments are optional. The following column specifications are all valid:

- Full column specification:

  ```
  emp_data@main_server:employee.resume
  ```
- Column specification that omits the *server_name* argument:

  ```
  emp_data:employee.resume
  ```

  The **mi_lo_colinfo_by_name( )** function assumes that the **employee** table resides in the specified database that the current database server manages.
- Column specification that omits the *database* and *server_name* arguments:

  ```
  employee.resume
  ```

  The **mi_lo_colinfo_by_name( )** function assumes that the **employee** table resides in the database that is currently open and that the current database server manages.

American National Standards Institute

If the column is in a database that is ANSI compliant, you can also include the owner name, as follows:

```
database@server_name:owner.table.column
```

End of American National Standards Institute

Server Only

The **mi_lo_colinfo_by_name( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

MI_OK	indicates that the function was successful; LO_*spec* points to the LO-specification structure with the storage characteristics of *column_spec*.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_lo_colinfo_by_ids( )** and **mi_lo_spec_init( ).**

mi_lo_copy( )

The **mi_lo_copy( )** function creates a copy of a smart large object and opens the copy.

Syntax

```
MI_LO_FD mi_lo_copy(conn, src_LOhdl, LO_spec, open_mode, target_LOhdl_dptr)
   MI_CONNECTION *conn;
   MI_LO_HANDLE *src_LOhdl;
   MI_LO_SPEC *LO_spec;
   mi_integer open_mode;
   MI_LO_HANDLE **target_LOhdl_dptr;
```

conn	is one of the following connection values: A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )** A NULL-valued pointer (database server only)
src_LOhdl	is a pointer to the LO handle for the existing smart large object that is to be copied.
LO_spec	is a pointer to the LO-specification structure that contains the storage characteristics to use for the new smart large object.
open_mode	is an integer bitmask that specifies the open mode for the new smart large object that *target*_LO*hdl_dptr* references.
target_LOhdl_dptr	is a doubly indirected pointer to the target LO handle that identifies the new smart large object where **mi_lo_copy( )** copies the data in the smart large object that *src*_LO*hdl* references.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_copy( )** function performs the following steps to create a new smart large object whose data is copied from the smart large object that *src*_LO*hdl* references:

1. It obtains an LO handle for the new smart large object and assigns a pointer to the LO handle that *target*_LO*hdl_dptr* references.

   If the *target*_LO*hdl_dptr* value points to NULL, **mi_lo_copy( )** allocates a new LO handle for the new smart large object and assigns a pointer to this handle to *target*_LO*hdl_dptr*. If *target*_LO*hdl_dptr* does not point to NULL, **mi_lo_copy( )** assumes that you have already allocated an LO handle and uses the *target*_LO*hdl_dptr* argument as a pointer to an existing LO handle.
2. It assigns the storage characteristics from the LO-specification structure, LO_*spec*, to the new smart large object.

   If the LO-specification structure has *not* been updated with storage characteristics (the associated fields are null), the **mi_lo_copy( )** function uses the system-specified storage characteristics.

If the LO-specification structure was updated with storage characteristics, **mi_lo_copy( )** uses the storage characteristics that the LO-specification structure defines for the new smart large object.

3. It opens the new smart large object in the open mode that the *open_mode* argument specifies.

   The bitmask value for the *open_mode* argument indicates the open mode of the smart large object after **mi_lo_copy( )** successfully completes. For more information on valid open-mode flags, see the *IBM Informix DataBlade API Programmer's Guide*.
4. It copies the contents of the data in the smart large object that *src*_LO*hdl* references into the new smart large object that *target*_LO*hdl_dptr* references.

   The **mi_lo_copy( )** function writes the source data to the sbspace of the new smart large object.
5. It returns an LO file descriptor that identifies the new smart large object and is positioned at the start of this smart large object.

   When the **mi_lo_copy( )** function is successful, it returns a valid LO file descriptor. You can then use this file descriptor to identify which smart large object to access in subsequent function calls, such as **mi_lo_read( )** and **mi_lo_write( ).**

The **mi_lo_copy( )** function is a constructor function for both an LO file descriptor and an LO handle.

Server Only

If the **mi_lo_copy( )** function allocates an LO handle, it allocates this LO handle in the current memory duration. The **mi_lo_copy( )** function does *not* allocate memory for the LO file descriptor. Your UDR can assign this LO file descriptor to user memory with a desired memory duration.

End of Server Only

Each **mi_lo_copy( )** call is implicitly associated with the current session. When this session ends, the database server deallocates any smart large objects that **mi_lo_copy( )** has created if its LO handle has not been stored in a column (its reference count is zero) and no open LO file descriptors exist for the smart large object.

Server Only

The **mi_lo_copy( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

An **MI_LO_FD** value	is the LO file descriptor for the open smart large object that *target*_LO*hdl_dptr* references. The function also initializes the LO handle that *target*_LO*hdl_dptr* references.

MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_lo_create( ), mi_lo_from_file( ), mi_lo_spec_init( ),** and **mi_lo_open( ).**

mi_lo_create( )

The **mi_lo_create( )** function creates a new smart large object and opens it for access within a *DataBlade API* module.

Syntax

```
MI_LO_FD mi_lo_create(conn, LO_spec, open_mode, LOhdl_dptr)
   MI_CONNECTION *conn;
   MI_LO_SPEC *LO_spec;
   mi_integer open_mode;
   MI_LO_HANDLE **LOhdl_dptr;
```

conn	is one of the following connection values:
	A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )**
	A NULL-valued pointer (database server only)
LO_spec	is a pointer to the LO-specification structure that contains the storage characteristics to use for the new smart large object.
open_mode	is an integer bitmask that specifies the open mode for the smart large object that LO*hdl_dptr* references.
LOhdl_dptr	is a doubly indirected pointer to an LO handle that identifies the new smart large object.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_create( )** function performs the following steps to create a new smart large object that LO*hdl_dptr* references:

1. It obtains an LO handle for the new smart large object and assigns a pointer to the handle to the argument that LO*hdl_dptr* references.

   If the LO*hdl_dptr* argument is NULL, **mi_lo_create( )** allocates a new LO handle for the new smart large object and assigns a pointer to this handle to LO*hdl_dptr*. If LO*hdl_dptr* is not NULL, **mi_lo_create( )** assumes that you have already allocated an LO handle and uses the LO*hdl_dptr* argument to locate the LO handle for the new smart large object.
2. It assigns the storage characteristics from the LO-specification structure, LO*_spec*, to the new smart large object.

   If the LO-specification structure has *not* been updated with storage characteristics (the associated fields are null), the **mi_lo_create( )** function uses the system-specified storage characteristics.

   If the LO-specification structure was updated with storage characteristics, **mi_lo_create( )** uses the storage characteristics that the LO-specification structure defines for the new smart large object.
3. It opens the new smart large object in the open mode that the *open_mode* argument specifies.

   The bitmask value for the *open_mode* argument indicates the open mode of the smart large object after **mi_lo_create( )** successfully completes. For more information on valid open-mode flags, see the *IBM Informix DataBlade API Programmer's Guide*.

4. It returns an LO file descriptor that identifies the new smart large object and is positioned at the start of this smart large object.

   When the **mi_lo_create( )** function is successful, it returns a valid LO file descriptor. You can then use this file descriptor to identify which smart large object to access in subsequent function calls, such as **mi_lo_read( )** and **mi_lo_write( ).**

The **mi_lo_create( )** function is a constructor function for both an LO file descriptor and an LO handle.

Server Only

If the **mi_lo_create( )** function allocates an LO handle, it allocates this LO handle in the current memory duration. The **mi_lo_create( )** function does *not* allocate memory for the LO file descriptor. Your UDR can assign this LO file descriptor to user memory with a desired memory duration.

End of Server Only

Each **mi_lo_create( )** call is implicitly associated with the current session. When this session ends, the database server deallocates any smart large objects that **mi_lo_create( )** has created if its LO handle has not been stored in a column (its reference count is zero) and no open LO file descriptors exist for the smart large object.

Server Only

The **mi_lo_create( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

An **MI_LO_FD** value	is the LO file descriptor for the open smart large object that LO*hdl_dptr* references. The function also initializes the LO handle that LO*hdl_dptr* references.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_lo_copy( ), mi_lo_colinfo_by_ids( ), mi_lo_colinfo_by_name( ), mi_lo_from_file( ), mi_lo_spec_init( ),** and **mi_lo_specget_flags( ).**

mi_lo_decrefcount( )

The **mi_lo_decrefcount( )** function decrements the reference count of a smart large object.

Syntax

```
mi_integer mi_lo_decrefcount(conn, LO_hdl)
   MI_CONNECTION *conn;
   MI_LO_HANDLE *LO_hdl;
```

conn	is one of the following connection values: A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )** A NULL-valued pointer (database server only)
LO_hdl	is a pointer to the LO handle.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
No	Yes

Usage

The **mi_lo_decrefcount( )** function is useful for manually tracking multiple references to the same smart large object. Typical use of this function is in the **destroy( )** support function of an opaque data type that contains smart large objects. Use the **mi_lo_decrefcount( )** function in the **destroy( )** support function to decrement the reference count of the smart large object that is being deleted by one, as follows:

- If the opaque type does *not* have an **lohandles( )** support function, you must use the **mi_lo_decrefcount( )** function in the **destroy( )** support function of the opaque type.
- If the opaque type has an **lohandles( )** support function, do *not* use the **mi_lo_decrefcount( )** function in the **destroy( )** support function. The database server automatically decrements the reference count when it executes the **lohandles( )** support function at destroy time.

Server Only

The **mi_lo_decrefcount( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

>=0	is the new reference count for the smart large object that LO_*hdl* references.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the description of **mi_lo_increfcount( ).**

mi_lo_delete_immediate( )

The **mi_lo_delete_immediate( )** function deletes a smart large object.

Syntax

```
mi_integer mi_lo_delete_immediate(conn, LO_hdl)
   MI_CONNECTION *conn;
   MI_LO_HANDLE *LO_hdl;
```

conn	is one of the following connection values: A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )** A NULL-valued pointer (database server only)
LO_hdl	is a pointer to the LO handle that identifies the smart large object to be deleted.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_delete_immediate( )** function deletes the smart large object that the *LO_hdl* LO handle identifies. Normally, the database server deletes all smart large objects that have a reference count of zero and no open LO file descriptors at the end of the transaction. Therefore, you do not usually need to issue explicit delete calls. The **mi_lo_delete_immediate( )** function is useful when you want to delete some transient smart large object right away and reuse its associated resources. This function can delete a transient smart large object *only* when no open LO file descriptors exist for the smart large object.

Server Only

The **mi_lo_delete_immediate( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

The **mi_lo_delete_immediate( )** function immediately deletes all the information related to the specified smart large object. After the call to **mi_lo_delete_immediate( )**, all released resources can be reused, and any operations attempted on this smart large object will fail.

Warning: This *mi_lo_delete_immediate( )* is not recoverable. Even if the transaction is rolled back, the deleted smart large object cannot be re-created.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the description of **mi_lo_release( ).**

mi_lo_expand( )

The **mi_lo_expand( )** function copies multirepresentational data to a new smart large object.

Syntax

```
MI_LO_FD mi_lo_expand(conn, LOhdl_dptr, multirep_ptr, multirep_len,
   open_mode, LO_spec)
   MI_CONNECTION *conn;
   MI_LO_HANDLE **LOhdl_dptr;
   MI_MULTIREP_DATA *multirep_ptr;
   mi_integer multirep_len;
   mi_integer open_mode;
   MI_LO_SPEC *LO_spec;
```

conn	is one of the following connection values: A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )** A NULL-valued pointer (database server only)
LOhdl_dptr	is a doubly indirected pointer to an LO handle that identifies the new smart large object into which **mi_lo_expand( )** copies the multirepresentational data.
multirep_ptr	is a pointer to a buffer that contains the multirepresentational data to store in the new smart large object.
multirep_len	is an integer that specifies the size of the *multirep_ptr* data, in bytes.
open_mode	is an integer bitmask that specifies the open mode for the smart large object that LO*hdl_dptr* references.
LO_spec	is a pointer to the LO-specification structure that contains the storage characteristics to use for the new smart large object.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_expand( )** function performs the following steps to create a new smart large object that LO*hdl_dptr* references and to copy multirepresentational data into this new smart large object:

1. It obtains an LO handle for the new smart large object and assigns a pointer to the handle to the argument that LO*hdl_dptr* references.

   If the LO*hdl_dptr* argument is NULL, **mi_lo_expand( )** allocates a new LO handle for the new smart large object and assigns a pointer to this handle to LO*hdl_dptr*. If LO*hdl_dptr* is not NULL, **mi_lo_expand( )** assumes that you have already allocated an LO handle and uses the LO*hdl_ptr* argument to locate the LO handle for the new smart large object.
2. It assigns the storage characteristics from the LO-specification structure, LO*_spec*, to the new smart large object.

   If the LO-specification structure has *not* been updated with storage characteristics (the associated fields are null), the **mi_lo_expand( )** function uses the system-specified storage characteristics.

If the LO-specification structure was updated with storage characteristics, **mi_lo_expand( )** uses the storage characteristics that the LO-specification structure defines for the new smart large object.

3. It opens the new smart large object in the open mode that the *open_mode* argument specifies.

   The bitmask value for the *open_mode* argument indicates the open mode of the smart large object after **mi_lo_expand( )** successfully completes. For more information on valid open-mode flags, see the *IBM Informix DataBlade API Programmer's Guide*.
4. It copies the contents of the multirepresentational data in the *multirep_ptr* buffer into the new smart large object that LO*hdl_dptr* references.

   The **mi_lo_expand( )** function copies *multirep_len* bytes of the multirepresentational data to the sbspace of the new smart large object.
5. It returns an LO file descriptor that identifies the new smart large object and is positioned at the start of this smart large object.

   When the **mi_lo_expand( )** function is successful, it returns a valid LO file descriptor. You can then use this file descriptor to identify which smart large object to access in subsequent function calls, such as **mi_lo_read( )** and **mi_lo_write( ).**

The **mi_lo_expand( )** function is a constructor function for both an LO file descriptor and an LO handle.

Server Only

If the **mi_lo_expand( )** function allocates an LO handle, it allocates this LO handle in the current memory duration. The **mi_lo_expand( )** function does *not* allocate memory for the LO file descriptor. Your UDR can assign this LO file descriptor to user memory with a desired memory duration.

End of Server Only

Each **mi_lo_expand( )** call is implicitly associated with the current session. When this session ends, the database server deallocates any smart large objects that **mi_lo_expand( )** has created if its LO handle has not been stored in a column (its reference count is zero) and no open LO file descriptors exist for the smart large object.

Server Only

The **mi_lo_expand( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Important: The DataBlade API provides this function only for compatibility with earlier applications. For new DataBlade API modules, use the **mi_lo_create( )** or **mi_lo_from_file( )** function.

Return Values

An **MI_LO_FD** value   is the LO file descriptor for the open smart large

	object that LO*hdl_dptr* references. The function also initializes the LO handle that LO*hdl_dptr* references.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_lo_copy( ), mi_lo_create( ), mi_lo_from_file( ), mi_lo_spec_init( ), mi_lo_open( ),** and **mi_set_large( ).**

mi_lo_filename( )

The **mi_lo_filename( )** function constructs a filename for smart-large-object data based on an LO handle and a filename specification.

Syntax

```
const char *mi_lo_filename(conn, LO_hdl, fname_spec)
   MI_CONNECTION *conn;
   MI_LO_HANDLE *LO_hdl;
   const char *fname_spec;
```

conn	is one of the following connection values: A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )** A NULL-valued pointer (database server only)
LO_hdl	is a pointer to a valid LO handle.
fname_spec	is a specification for the destination file pathname. It can include wildcard characters.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_filename( )** function generates a filename from the *fname_spec* argument that you provide. Use the **mi_lo_filename( )** function to determine the filename that the **mi_lo_to_file( )** function would create for its *fname_spec* argument. This function uses a template to specify the exact format of the result. By default, the **mi_lo_to_file( )** function generates a filename of the form:

fname.*hex_id*

However, you can specify wildcards in the *fname_spec* argument that can change this default filename. You can use these wildcards in the *fname_spec* argument of **mi_lo_filename( )** to see what filename these wildcards generate. For more information on the wildcards that are valid in the *fname_spec* argument, see the description of the **mi_lo_to_file( )** function.

You are responsible for freeing the memory that the return value occupies.

Server Only

The **mi_lo_filename( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

A **char** pointer	is the character string that is the filename that the **mi_lo_to_file( )** function would generate.

NULL	indicates that the function was not successful.

Related Topics

See also the description of **mi_lo_to_file( ).**

mi_lo_from_buffer( )

The **mi_lo_from_buffer( )** function copies a specified number of bytes from a user-defined buffer to an existing smart large object.

Syntax

```
mi_integer mi_lo_from_buffer(conn, LO_hdl, size, buffer)
   MI_CONNECTION *conn;
   MI_LO_HANDLE *LO_hdl;
   mi_integer size;
   char *buffer;
```

conn	is one of the following connection values: A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )** A NULL-valued pointer (database server only)
LO_hdl	is the LO handle for the smart large object into which you want to copy the buffer data.
size	is an integer that identifies the number of bytes to copy to the smart large object.
buffer	is a pointer to a user-defined buffer from which you want to copy the data.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_from_buffer( )** function copies up to *size* bytes from the user-defined *buffer* into a smart large object that the LO_*hdl* LO handle references. The write operation to the smart large object starts at a zero-byte offset. This function allows you to write data to a smart large object without opening the smart large object. To use the **mi_lo_from_buffer( )** function to copy data, the smart large object must already exist in an sbspace.

Server Only

The **mi_lo_from_buffer( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

>=0	is the number of bytes copied from the user-defined buffer to the smart large object. This value should always be equal to the *size* value.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_lo_from_file( )** and **mi_lo_to_buffer( ).**

mi_lo_from_file( )

The **mi_lo_from_file( )** function copies the contents of an operating-system file on the server or client computer to a new smart large object.

Syntax

```
MI_LO_FD mi_lo_from_file(conn, LO_dptr, fname_spec, open_mode, offset,amount,
   LO_spec)
   MI_CONNECTION *conn;
   MI_LO_HANDLE **LO_dptr;
   const char *fname_spec;
   mi_integer open_mode;
   mi_integer offset;
   mi_integer amount;
   MI_LO_SPEC *LO_spec;
```

conn	is one of the following connection values: A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )** A NULL-valued pointer (database server only)
LO_dptr	is a doubly indirected pointer to the LO handle that identifies the new smart large object. This smart large object is where **mi_lo_from_file( )** copies the file data.
fname_spec	is the full pathname to the operating-system file to copy into the new smart large object.
open_mode	is an integer bitmask to indicate how to open the operating-system file and where this file is located. For a list of valid file-mode constants, see the table in the following "Usage" section.
offset	is the point to begin the read in the operating-system file. The *offset* value is the number of bytes from the beginning of the file, starting at `0`.
amount	is the amount of data to read from the operating-system file, starting at the offset. An *amount* value of `-1` means read to the end of the file.
LO_spec	is a pointer to an LO-specification structure that contains the storage characteristics of the new smart large object.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_from_file( )** function performs the following steps to create a new smart large object that LO*hdl_dptr* references and copy data from an operating-system file to a new smart large object:

1. It creates an LO handle for the new smart large object and assigns a pointer to the handle to the argument that LO*hdl_dptr* references.

   If the LO*hdl_dptr* argument is NULL, **mi_lo_from_file( )** allocates a new LO handle for the new smart large object and assigns a pointer to this handle to LO*hdl_dptr*. If LO*hdl_dptr* is not NULL, **mi_lo_from_file( )** assumes that you have already allocated an LO handle and uses the LO*hdl_dptr* argument to locate the LO handle for the new smart large object.

2. It assigns the storage characteristics from the LO-specification structure, LO*spec*, to the new smart large object.

   If the LO-specification structure has *not* been updated with storage characteristics (the associated fields are null), the **mi_lo_from_file( )** function uses the system-specified storage characteristics.

   If the LO-specification structure was updated with storage characteristics, **mi_lo_from_file( )** uses the storage characteristics that the LO-specification structure defines for the new smart large object.
3. It opens the new smart large object in read/write access mode (MI_LO_RDWR).

   The **mi_lo_from_file( )** function does not accept an open-mode flag for the smart large object as an argument. The *open_mode* argument specifies the open mode for the operating-system file.
4. It copies the contents of the operating-system file whose name is in the *fname_spec* buffer into the new smart large object that LO*hdl_dptr* references.

   The **mi_lo_from_file( )** function opens the operating-system file in the mode that the *open_mode* argument indicates. In the *fname_spec* operating-system file, the **mi_lo_from_file( )** function begins the read operation at the file offset that *offset* indicates and reads the number of bytes that *amount* specifies. The function writes the file data to the sbspace of the new smart large object.
5. It returns an LO file descriptor that identifies the new smart large object.

   When the **mi_lo_from_file( )** function is successful, it returns a valid LO file descriptor. When it completes, **mi_lo_from_file( )** leaves the LO file position at the end of the smart large object. It does *not* reset the LO file position to the beginning of the smart large object.

   You can use this file descriptor to identify which smart large object to access in subsequent function calls, such as **mi_lo_read( )** and **mi_lo_write( ).**

The **mi_lo_from_file( )** function is a constructor function for both an LO file descriptor and an LO handle.

Server Only

If the **mi_lo_from_file( )** function allocates an LO handle, it allocates this LO handle in the current memory duration. The **mi_lo_from_file( )** function does *not* allocate memory for the LO file descriptor. Your UDR can assign this LO file descriptor to user memory with a desired memory duration.

End of Server Only

You can include environment variables in the *fname_spec* path with the following syntax:

```
$ENV_VAR
```

These environment variables must be set in the database server environment; that is, they must be set *before* the database server starts.

The **mi_lo_from_file( )** function can access the operating-system files on either the server or the client computer. The file-mode values for the *open_mode* argument indicate the location of the file to copy and the access mode of the source file. Valid values include the following file-mode constants.

File-Mode Constant	Purpose
MI_O_EXCL	Open the file only if *fname_spec* does not exist.

MI_O_TRUNC	Zero out the input file before reading it.
MI_O_APPEND	Allow appending to the end of the file. (This function does not write to the source file.)
MI_O_RDWR	Open the file in read/write mode. (This function does not write to the source file.)
MI_O_RDONLY	Open the file in read-only mode.
MI_O_TEXT	Process the file as text (not binary).
MI_O_SERVER_FILE	The *fname_spec* file is on the server computer.
MI_O_CLIENT_FILE	The *fname_spec* file is on the client computer.

Important: The MI_O_TRUNC flag is valid but is not often useful in a DataBlade API module.

The default *open_mode* value is:

```
MI_O_RDONLY | MI_O_CLIENT_FILE
```

The **mi_lo_from_file( )** function allows you to copy part of a file with the *offset* and *amount* parameters.

Each **mi_lo_from_file( )** call is implicitly associated with the current session. When this session ends, the database server deallocates any smart large object that **mi_lo_from_file( )** has created if its LO handle has not been stored in a column (its reference count is zero) and no open LO file descriptors exist for the smart large object.

Server Only

The **mi_lo_from_file( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

An **MI_LO_FD** value	is the LO file descriptor of the open smart large object that LO*hdl_dptr* references. The function also initializes the LO handle that LO*hdl_dptr* references.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_lo_copy( ), mi_lo_create( ), mi_lo_from_buffer( ), mi_lo_spec_init( ), mi_lo_from_file_by_lofd( ),** and **mi_lo_to_file( ).**

For information on the use of **mi_lo_from_file( )** in an import opaque-type support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_lo_from_file_by_lofd( )

The **mi_lo_from_file_by_lofd( )** function copies the contents of an operating-system file on the server or client computer to an open smart large object.

Syntax

```
mi_integer mi_lo_from_file_by_lofd(conn, LO_fd, fname_spec, open_mode,
   offset, amount)
   MI_CONNECTION *conn;
   MI_LO_FD LO_fd;
   const char *fname_spec;
   mi_integer open_mode;
   mi_integer offset;
   mi_integer amount;
```

conn	is one of the following connection values: A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )** A NULL-valued pointer (database server only)
LO_fd	is a file descriptor for an existing smart large object
fname_spec	is the path to the operating-system file to copy into a smart large object.
open_mode	is an integer bitmask to indicate how to open the operating-system file and where this file is located. For a list of valid file-mode constants, see the table in the following "Usage" section.
offset	is the point to begin reading in the file. The *offset* value is the number of bytes from the beginning of the file, starting at 0.
amount	is the amount of data to read, starting at the offset. An *amount* value of -1 means read to the end of the file.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_lo_from_file_by_lofd( )** function enables you to write to a smart large object that is already open. This function provides more flexibility than **mi_lo_from_file( ),** in the following ways:

- You can position the LO file descriptor anywhere before the operation.
- You can copy multiple files to a single smart large object.

To include environment variables in the *fname_spec* path, use the following syntax:

```
$ENV_VAR
```

You must set these environment variables in the database server environment, before the database server starts.

The **mi_lo_from_file_by_lofd( )** function can create the target files on either the server or the client computer. The file-mode flag values for the *open_mode* argument indicate the location of the file to copy and the access mode of the source file. Valid values include the following file-mode constants.

File-Mode Constant	Purpose
MI_O_EXCL	Open the file only if *fname_spec* does not exist.
MI_O_TRUNC	Zero out the input file before reading it.
MI_O_APPEND	Allow appending to the end of the file. (This function does not write to the source file.)
MI_O_RDWR	Open the file in read/write mode. (This function does not write to the source file.)
MI_O_RDONLY	Open the file in read-only mode.
MI_O_TEXT	Process the file as text (not binary). (Binary is used if you do not specify **MI_O_TEXT**.)
MI_O_SERVER_FILE	The *fname_spec* file is on the server computer.
MI_O_CLIENT_FILE	The *fname_spec* file is on the client computer.

Important: The MI_O_TRUNC flag is valid but is not often useful in a DataBlade API module.

The default *open_mode* value is:

```
MI_O_RDONLY | MI_O_CLIENT_FILE
```

Server Only

The **mi_lo_from_file_by_lofd( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_lo_create( ), mi_lo_spec_init( ), mi_lo_from_buffer( ), mi_lo_from_file( ),** and **mi_lo_to_file( ).**

mi_lo_from_string( )

The **mi_lo_from_string( )** function converts an LO handle in its text representation to its binary representation.

Syntax

```
MI_LO_HANDLE *mi_lo_from_string(LOhdl_str)
   char *LOhdl_str;
```

LOhdl_str is the text representation of the LO handle to convert.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_from_string( )** function converts the text version of an LO handle, which *LOhdl_str* references, to the binary representation for the LO handle. It is a constructor function for an LO handle.

Server Only

The **mi_lo_from_string( )** function allocates a new LO handle in the current memory duration.

End of Server Only

Return Values

An **MI_LO_HANDLE** pointer	is an LO handle that is the binary representation of the text LO handle that LO*hdl_str* references.
NULL	indicates that the function was not successful.

Related Topics

See also the description of **mi_lo_to_string( ).**

mi_lo_increfcount( )

The **mi_lo_increfcount( )** function increments the reference count of a smart large object.

Syntax

```
mi_integer mi_lo_increfcount(conn, LO_hdl)
   MI_CONNECTION *conn;
   MI_LO_HANDLE *LO_hdl;
```

conn	is one of the following connection values: A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )** A NULL-valued pointer (database server only)
LO_hdl	is a pointer to the LO handle.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
No	Yes

Usage

The **mi_lo_increfcount( )** function is useful for manually tracking multiple references to the same smart large object. Typical use of this function is in the **assign( )** support function of an opaque data type that contains smart large objects. Use the **mi_lo_increfcount( )** function in the **assign( )** support function to increment the reference count of the new smart large object by one, as follows:

- If the opaque type does *not* have an **lohandles( )** support function, you must use the **mi_lo_increfcount( )** function in the **assign( )** support function of the opaque type.
- If the opaque type has an **lohandles( )** support function, do not use the **mi_lo_increfcount( )** function in the **assign( )** support function; the database server automatically increments the reference count when it executes the **lohandles( )** support function at assign time.

Server Only

The **mi_lo_increfcount( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

>=0	is the new reference count for the smart large object that LO_*hdl* references.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the description of **mi_lo_decrefcount( ).**

mi_lo_invalidate( )

The **mi_lo_invalidate( )** function marks the LO handle as invalid.

Syntax

```
mi_integer mi_lo_invalidate(conn, LO_hdl)
   MI_CONNECTION *conn;
   MI_LO_HANDLE *LO_hdl;
```

conn	is one of the following connection values:
	A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )**
	A NULL-valued pointer (database server only)
LO_hdl	is a pointer to the LO handle to invalidate.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_invalidate( )** function marks as invalid the LO handle that LO_*hdl* references. You can use **mi_lo_invalidate( )** in the support function of an opaque data type to make smart large objects invalid. It enables these support functions to invalidate an LO handle before it is stored. You might want to mark an LO handle as invalid to indicate that it is not active. The **lohandles( )** support function can unambiguously determine which **MI_LO_HANDLE** values are valid for the given instance of the opaque type.

If **mi_lo_invalidate( )** fails due to an invalid connection, callback functions are invoked. If it fails due to an invalid handle, callbacks are *not* invoked.

Server Only

The **mi_lo_invalidate( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

MI_OK	indicates that the function was successful; the LO handle was successfully invalidated.
MI_ERROR	indicates that the function was not successful; the connection was invalid or the handle was invalid.

Related Topics

See also the description of **mi_lo_validate( ).**

mi_lo_lock( )

The **mi_lo_lock( )** function obtains a byte-range lock on the specified number of bytes in a smart large object.

Syntax

```
mi_integer mi_lo_lock(conn, LO_fd, offset, whence, nbytes, lock_mode)
   MI_CONNECTION *conn;
   MI_LO_FD LO_fd;
   mi_int8 *offset;
   mi_integer whence;
   mi_int8 *nbytes;
   mi_integer lock_mode;
```

conn	is one of the following connection values: A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )** A NULL-valued pointer (database server only)
LO_fd	is the LO file descriptor of the smart large object to lock.
offset	is a pointer to the eight-byte integer (**mi_int8**) offset from the starting LO seek position that *whence* specifies.
whence	is an integer value that identifies the starting LO seek position.
nbytes	is a pointer to the eight-byte integer (**mi_int8**) that specifies the number of bytes to lock. This value cannot exceed two gigabytes.
lock_mode	is an integer constant that indicates the lock mode to use. Valid constant values follow: MI_LO_SHARED_MODE — Lock mode is shared mode. MI_LO_EXCLUSIVE_MODE — Lock mode is exclusive mode.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
No	Yes

Usage

The **mi_lo_lock( )** function requests a byte-range lock of the *lock_mode* type on the open smart large object that *LO_fd* indicates. This lock request applies to *nbytes* number of bytes beginning at the LO seek location that the *whence* and *offset* arguments specify.

By default, the database server locks the entire smart large object when you request a read or write operation. With the **mi_lo_lock( )** function, you can request a byte-range lock, which allows transactions to lock only the required ranges of bytes in a smart large object. However, the smart large object on which you request the byte-range lock must have the byte-range locking feature enabled with the LO_LOCKRANGE storage-characteristic constant.

Server Only

The **mi_lo_lock( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

MI_OK	indicates that the function was successful; the requested lock was successfully obtained.
MI_ERROR	indicates that the function was *not* successful; the requested lock could not be obtained or the smart large object has not been opened for byte-range locking.

Related Topics

See also the descriptions of **mi_lo_open( ), mi_lo_specset_def_open_flags( ),** and **mi_lo_unlock( ).**

For more information on locks for smart large objects or on how to use byte-range locks, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_lo_lolist_create( )

The **mi_lo_lolist_create( )** function converts an array of LO handles into an **MI_LO_LIST** structure.

Syntax

```
mi_integer mi_lo_lolist_create(conn, LOhdl_cnt, LO_hdls, LO_list)
   MI_CONNECTION *conn;
   mi_integer LOhdl_cnt;
   MI_LO_HANDLE **LO_hdls;
   MI_LO_LIST **LO_list ;
```

conn	is one of the following connection values: A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )** A NULL-valued pointer (database server only)
LOhdl_cnt	is the number of LO handles in the LO*_hdls* array.
LO_hdls	is a pointer to an array of LO handles that **mi_lo_lolist_create( )** converts.
LO_list	is a pointer to the **MI_LO_LIST** structure that **mi_lo_lolist_create( )** creates.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_lo_lolist_create( )** function is the constructor function for the **MI_LO_LIST** structure. It populates an **MI_LO_LIST** structure with LO handles from the specified LO*_hdls* array. The **MI_LO_LIST** structure allows user-defined routines to pass an array of LO handles to and from the database server. The function handles memory allocation for the **MI_LO_LIST** structure as follows:

- When the LO*_list* value points to NULL, this function allocates an **MI_LO_LIST** structure.
- When LO*_list* does not point to NULL, the **mi_lo_lolist_create( )** function assumes that LO*_list* points to memory that has already been allocated by a previous call to **mi_lo_lolist_create( )**.

The **mi_lo_lolist_create( )** function allocates a new **MI_LO_LIST** structure in the current memory duration. The function then initializes the **MI_LO_LIST** structure with the LO handles, which LO*_hdls* indicates. It converts the number of LO handles that the LO*hdl_cnt* argument specifies.

The **mi_lo_lolist_create( )** function is useful in an **lohandles( )** support function of an opaque data type. It converts an array of LO handles to the flat **MI_LO_LIST** structure that the **lohandles( )** support function returns. The database server automatically manages the reference count of smart large objects when it calls the **lohandles( )** support function at the following times:

- Assign time (when any **assign( )** support function would execute)
- Destroy time (when any **destroy( )** support function would execute)
- Import time (when any import support function would execute)

Server Only

The **mi_lo_lolist_create( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

>0	is the number of bytes of the **MI_LO_LIST** structure.
0	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_lo_decrefcount( )** and **mi_lo_increfcount( ).**

mi_lo_open( )

The **mi_lo_open( )** function opens an existing smart large object for access.

Syntax

```
MI_LO_FD mi_lo_open(conn, LO_hdl, open_mode)
   MI_CONNECTION *conn;
   MI_LO_HANDLE *LO_hdl;
   mi_integer open_mode;
```

conn is one of the following connection values:

A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )**

A NULL-valued pointer (database server only)

LO_hdl is a pointer to the LO handle that identifies the smart large object to open.

open_mode is an integer bit mask that specifies the open mode for the smart large object that LO*_hdl* references. The bit mask can contain the following open-mode constants:

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
Access methods	MI_LO_RANDOM	Random access
	MI_LO_SEQUENTIAL	Sequential access
Buffering modes	MI_LO_BUFFER	Buffered access (Buffered I/O)
	MI_LO_NOBUFFER	Unbuffered access (Light-weight I/O)
Locking modes	MI_LO_LOCKALL	Lock-all locks
	MI_LO_LOCKRANGE	Byte-range locks

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_open( )** function opens the existing smart large object that LO*_hdl* references. To access the data of a smart large object, a DataBlade API module must first open the smart large object. The **mi_lo_open( )** function performs the following steps:

1. It opens the smart large object that LO*_hdl* references in the open mode that the *open_mode* flag indicates.
   - If the *open_mode* flag is zero (0), **mi_lo_open( )** uses the default open mode, which you can set in the LO-specification structure with the **mi_lo_specset_def_open_flags( )** function.
   - A non-zero *open_mode* flag specifies the open-mode information with a valid combination of open-mode constants.
2. It sets the LO seek position of the smart large object to byte zero (0).
3. It obtains a lock on the smart-large-object data based on the lock mode in the *open_mode* argument.
4. It returns an LO file descriptor that identifies the smart large object.

When the **mi_lo_open( )** function is successful, it returns a valid LO file descriptor. The **mi_lo_open( )** function is a constructor function for an LO file descriptor. You can then use this file descriptor to identify which smart large object to access in subsequent function calls such as **mi_lo_read( )** and **mi_lo_write( ).** However, this LO file descriptor is only valid within the current session.

Server Only

The **mi_lo_open( )** function allocates a new LO file descriptor in the current memory duration.

End of Server Only

Important: The database server does not check access permissions on the smart large object that the LO handle identifies. Your DataBlade API module must ensure that the end user or another application is trusted.

Each **mi_lo_open( )** call is implicitly associated with the current session. When this session ends, the database server deallocates any smart large objects that are not referenced by any columns (those with a reference count of zero (0)).

Server Only

The **mi_lo_open( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

An **MI_LO_FD** value	is the LO file descriptor for the open smart large object that LO_*hdl* references.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_lo_close( ), mi_lo_copy( ), mi_lo_create( ), mi_lo_read( ), mi_lo_readwithseek( ), mi_lo_specget_def_open_flags( ), mi_lo_specset_def_open_flags( ), mi_lo_tell( ), mi_lo_write( ),** and **mi_lo_writewithseek( ).**

mi_lo_ptr_cmp( )

The **mi_lo_ptr_cmp( )** function compares two LO handles to determine if they reference the same smart large object.

Syntax

```
mi_integer mi_lo_ptr_cmp(conn, LO_hdl1, LO_hdl2)
   MI_CONNECTION *conn;
   MI_LO_HANDLE *LO_hdl1;
   MI_LO_HANDLE *LO_hdl2;
```

conn	is one of the following connection values:
	A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )**
	A NULL-valued pointer (database server only)
LO_hdl1	is a pointer to the first LO handle.
LO_hdl2	is a pointer to the second LO handle.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

A simple byte-wise comparison of the two LO handles is not sufficient to determine equivalence. For example, a given smart large object might be referenced in two tables. However, if column-specific information is part of the LO handle, the two LO handles reference the same smart large object but are not equivalent because the column-level information is different.

Server Only

The **mi_lo_ptr_cmp( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

0	indicates that the two LO handles, LO*_hdl1* and LO*_hdl2*, reference the same smart large object.
1	indicates that the two LO handles, LO*_hdl1* and LO*_hdl2*, reference different smart large objects.
MI_ERROR	indicates that the function was not successful; arguments might be invalid.

mi_lo_read( )

The **mi_lo_read( )** function reads a specified number of bytes of data from an open smart large object into a buffer.

Syntax

```
mi_integer mi_lo_read(conn, LO_fd, buf, nbytes)
   MI_CONNECTION *conn;
   MI_LO_FD LO_fd;
   char *buf;
   mi_integer nbytes;
```

conn	is one of the following connection values:
	A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )**
	A NULL-valued pointer (database server only)
LO_fd	is an LO file descriptor for the smart large object from which to read the data. It is obtained from a previous call to **mi_lo_open( ).**
buf	is a pointer to a user-allocated character buffer that contains the data that **mi_lo_read( )** reads from the smart large object.
nbytes	is the maximum number of bytes to read into the *buf* character buffer. This value cannot exceed two gigabytes.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_read( )** function reads *nbytes* of data from the open smart large object that the LO*_fd* file descriptor identifies. The read begins at the current LO seek position for LO*_fd*. You can use the **mi_lo_tell( )** function to obtain the current LO seek position.

The function reads this data into the user-allocated buffer that *buf* references. The *buf* buffer must be less than two gigabytes in size. To read smart large objects that are larger than two gigabytes, read them in two-gigabyte chunks.

To perform a seek operation before a read operation, use the function **mi_lo_readwithseek( ).**

Server Only

The **mi_lo_read( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

>=0	is the actual number of bytes that the function has read from the open smart large object to the *buf* buffer.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_lo_open( ), mi_lo_readwithseek( ), mi_lo_seek( ), mi_lo_tell( ),** and **mi_lo_write( ).**

mi_lo_readwithseek( )

The **mi_lo_readwithseek( )** function performs a seek operation and then reads a specified number of bytes of data from an open smart large object.

Syntax

```
mi_integer mi_lo_readwithseek(conn, LO_fd, buf, nbytes, offset, whence)
   MI_CONNECTION *conn;
   MI_LO_FD LO_fd;
   char *buf;
   mi_integer nbytes;
   mi_int8 *offset;
   mi_integer whence;
```

conn	is one of the following connection values: A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )** A NULL-valued pointer (database server only)
LO_fd	is an LO file descriptor for the smart large object from which to read.
buf	is a pointer to a user-allocated character buffer that contains the data that **mi_lo_readwithseek( )** reads from the smart large object.
nbytes	is the maximum number of bytes to read from the *buf* character buffer. This value cannot exceed two gigabytes.
offset	is a pointer to the eight-byte integer (**mi_int8**) offset from the starting LO seek position.
whence	is an integer value that identifies the starting LO seek position.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_readwithseek( )** function reads *nbytes* of data from the open smart large object that the LO_*fd* file descriptor identifies. The read begins at the LO seek position of LO_*fd* that the *offset* and *whence* arguments indicate, as follows:

- The *whence* argument identifies the position from which to start the seek operation.

  Valid values include the following whence constants.

Whence Constant	Starting LO Seek Position
MI_LO_SEEK_SET	The start of the smart large object
MI_LO_SEEK_CUR	The current LO seek position in the smart large object
MI_LO_SEEK_END	The end of the smart large object

  – The *offset* argument identifies the offset, in bytes, relative to the starting seek position (which the *whence* argument specifies) at which to begin the read operation.

This *offset* value can be negative for all values of *whence*. For more information on how to access eight-bit (INT8) integers, see the *IBM Informix DataBlade API Programmer's Guide*.

The function reads this data into the user-allocated buffer that *buf* references. The *buf* buffer must be less than two gigabytes in size. To read smart large objects that are larger than two gigabytes, read them in two-gigabyte chunks.

Server Only

The **mi_lo_readwithseek( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

>=0	is the number of bytes that the function has read from the open smart large object to the *buf* character buffer.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_lo_read( ), mi_lo_seek( ),** and **mi_lo_writewithseek( ).**

mi_lo_release( )

The **mi_lo_release( )** function tells the database server that the resources that are associated with a temporary smart large object can be released.

Syntax

```
mi_integer mi_lo_release(conn, LO_hdl)
   MI_CONNECTION *conn;
   MI_LO_HANDLE *LO_hdl;
```

conn	is one of the following connection values:
	A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )**
	A NULL-valued pointer (database server only)
LO_hdl	is the LO handle for the temporary smart large object whose resources are to be deallocated.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_release( )** function frees the LO handle that *LO_hdl* references, thereby decrementing the reference count of the smart large object by one. This function is a destructor function for the LO handle. It is useful for telling the database server that it is safe to release resources associated with a temporary smart large object. A *temporary* smart large object is one that has one or more LO handles, but none of these handles have been inserted into a table. Temporary 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 user-defined routine 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 and sends each one to the client application:

```
SELECT filetoblob(...) FROM table1;
```

The client LIBMI application can use the **mi_lo_release( )** function to indicate to the database server when it finishes processing each of these smart large objects. Once you call this function on a temporary smart large object, the database server is free to release the resources at any time. Further use of the LO handle and any associated LO file descriptors is not guaranteed to work.

Use of this function on smart large objects that are *not* temporary does not cause any incorrect behavior; however, the call is expensive. You do not need to use this function for permanent smart large objects.

Server Only

The **mi_lo_release( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a

NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the description of **mi_lo_delete_immediate( ).**

mi_lo_seek( )

The **mi_lo_seek( )** function sets the LO seek position for the next read or write operation on an open smart large object.

Syntax

```
mi_integer mi_lo_seek(conn, LO_fd, offset, whence, seek_pos)
   MI_CONNECTION *conn;
   MI_LO_FD LO_fd;
   mi_int8 *offset;
   mi_integer whence;
   mi_int8 *seek_pos;
```

conn	is one of the following connection values:
	A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )**
	A NULL-valued pointer (database server only)
LO_fd	is an LO file descriptor for the smart large object whose seek position you want to change. A previous call to **mi_lo_open( )** obtains this descriptor.
offset	is a pointer to the eight-byte integer (**mi_int8**) offset from the specified *whence* seek position.
whence	determines how the *offset* value is interpreted.
seek_pos	is a pointer to the eight-byte integer (**mi_int8**) that identifies the new LO seek position.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_seek( )** function uses the *whence* and *offset* arguments to determine the new LO seek position of the smart large object that LO*_hdl* identifies, as follows:

- The *whence* argument identifies the position from which to start the seek operation.

  Valid values include the following whence constants.

Whence Constant	Starting Seek Position
MI_LO_SEEK_SET	The start of the smart large object
MI_LO_SEEK_CUR	The current LO seek position in the smart large object
MI_LO_SEEK_END	The end of the smart large object

- The *offset* argument identifies the offset, in bytes, from the starting seek position (which the *whence* argument specifies) at which to begin the seek.

  This *offset* value can be negative for all values of *whence*. For more information on how to access eight-bit (INT8) integers, see the *IBM Informix DataBlade API Programmer's Guide*.

The **mi_lo_seek( )** function returns the new seek position in the **mi_int8** *seek_pos* variable.

Server Only

The **mi_lo_seek( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

To obtain the current seek position, use the **mi_lo_tell( )** function.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_lo_read( ), mi_lo_readwithseek( ), mi_lo_tell( ), mi_lo_write( ),** and **mi_lo_writewithseek( ).**

mi_lo_spec_free( )

The **mi_lo_spec_free( )** function frees an LO-specification structure.

Syntax

```
mi_integer mi_lo_spec_free(conn, LO_spec)
   MI_CONNECTION *conn;
   MI_LO_SPEC *LO_spec;
```

conn	is one of the following connection values:
	A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )**
	A NULL-valued pointer (database server only)
LO_spec	is a pointer to the LO-specification structure to free.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_spec_free( )** function is the destructor for the LO-specification structure. It frees the LO-specification structure that LO_*spec* references.

Important: Do *not* use system memory-allocation calls (such as **free( )** or **mi_free( )**) to perform memory management for LO-specification structures.

When your application no longer needs an LO-specification structure, call **mi_lo_spec_free( )** to free the resources of the LO-specification structure that the **mi_lo_spec_init( )** function has allocated. Once freed, these resources can be reallocated to other structures.

Warning: Do not call the **mi_lo_spec_free( )** function for the same LO-specification structure more than once. This behavior is analogous to the behavior of the **free( )** system function for regular memory allocation.

Server Only

The **mi_lo_spec_free( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_lo_open( ), mi_lo_spec_init( ), mi_lo_specget_estbytes( ), mi_lo_specget_extsz( ), mi_lo_specget_flags( ),**

mi_lo_specget_maxbytes( ), mi_lo_specget_sbspace( ), mi_lo_specset_estbytes( ), mi_lo_specset_extsz( ), mi_lo_specset_flags( ), mi_lo_specset_maxbytes( ), and **mi_lo_specset_sbspace( ).**

mi_lo_spec_init( )

The **mi_lo_spec_init( )** function allocates and initializes the default system storage characteristics that are used to create a new smart large object.

Syntax

```
mi_integer mi_lo_spec_init(conn, LOspec_dptr)
   MI_CONNECTION *conn;
   MI_LO_SPEC **LOspec_dptr;
```

conn	is one of the following connection values: A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )** A NULL-valued pointer (database server only)
LOspec_dptr	is a doubly indirected pointer to an LO-specification structure for a smart large object.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_spec_init( )** function is a constructor for the LO-specification structure. It performs the following steps to create an LO-specification structure:

1. It handles memory allocation for an LO-specification structure.

   When the LO*spec_dptr* value (a single indirect pointer to an LO-specification structure) is NULL, this function allocates an LO- specification structure. Before you use an LO-specification structure, set LO*spec_dptr* to NULL so that **mi_lo_spec_init( )** allocates space for the LO-specification structure. When LO*spec_dptr* does not point to NULL, the **mi_lo_spec_init( )** function assumes that LO*spec_dptr* points to an LO-specification structure that has already been allocated by a previous call to **mi_lo_spec_init( )**.
2. It initializes the fields in the LO-specification structure, which LO*spec_dptr* references, to the appropriate null values (zero or a NULL-valued pointer).

   When the smart-large-object optimizer receives this initialized LO-specification structure, it obtains system-specified storage characteristics for the new smart large object.

Server Only

The **mi_lo_spec_init( )** function allocates a new LO-specification structure in the current memory duration.

The **mi_lo_spec_init( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Important: Before you use an LO-specification structure in a DataBlade API module, you must either allocate a new one with the **mi_lo_spec_init(**

) function or obtain one from an existing smart large object with the **mi_lo_stat_cspec( )** function. You can use the **mi_lo_colinfo_by_ids( )** or **mi_lo_colinfo_by_name( )** function to obtain storage characteristics that are associated with a particular column.

Do *not* use system memory-allocation calls (such as **malloc( )** or **mi_alloc( )**) to perform memory management for LO-specification structures. Use the **mi_lo_spec_init( )** function to create a new LO-specification structure and the **mi_lo_spec_free( )** function to free an LO-specification structure.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_lo_open( ), mi_lo_spec_free( ), mi_lo_specget_estbytes( ), mi_lo_specget_extsz( ), mi_lo_specget_flags( ), mi_lo_specget_maxbytes( ), mi_lo_specget_sbspace( ), mi_lo_specset_estbytes( ), mi_lo_specset_extsz( ), mi_lo_specset_flags( ), mi_lo_specset_maxbytes( ), mi_lo_specset_sbspace( ),** and **mi_lo_stat_cspec( ).**

mi_lo_specget_def_open_flags( )

The **mi_lo_specget_def_open_flags( )** function obtains from an LO-specification structure the default open-mode flag for a smart large object.

Syntax

```
mi_integer mi_lo_specget_def_open(LO_spec)
   MI_LO_SPEC *LO_spec;
```

LO_spec	is a pointer to the LO-specification structure from which to obtain the default open flags.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_specget_def_open_flags( )** function obtains the current value for the default open-mode flag from the LO-specification structure that LO_*spec* references. This default open-mode flag is the bit mask that indicates which open mode the **mi_lo_open( )** function uses when it opens the smart large object but does not specify an open-mode flag. To override the default open-mode flag, you can specify an open mode as an argument to **mi_lo_open( ).**

Important: Before you call **mi_lo_specget_def_open_flags( )**, you must put storage characteristics into an LO-specification structure.

You can use any of the following functions to initialize the LO-specification structure:

- The **mi_lo_colinfo_by_ids( )** or **mi_lo_colinfo_by_name( )** function puts storage characteristics that are associated with a particular CLOB or BLOB column in an LO-specification structure.
- The **mi_lo_stat_cspec( )** function puts storage characteristics of an existing smart large object in an LO-specification.
- The **mi_lo_specset_def_open_flags( )** function sets the default open flags in an LO-specification structure.

Return Values

MI_OK	indicates that the bit-mask value of the default open flags for the smart large object that LO_*spec* references.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_lo_copy( ), mi_lo_create( ), mi_lo_from_file( ), mi_lo_open( ), mi_lo_spec_free( ), mi_lo_spec_init( ),** and **mi_lo_specset_def_open_flags( ).**

For more information on the default open mode of a smart large object or on how to use the **mi_lo_specget_def_open_flags( )** function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_lo_specget_estbytes( )

The **mi_lo_specget_estbytes( )** function obtains from an LO-specification structure the estimated size, in bytes, of a smart large object.

Syntax

```
mi_integer mi_lo_specget_estbytes(LO_spec, estbytes)
   MI_LO_SPEC *LO_spec;
   mi_int8 *estbytes;
```

LO_spec	is a pointer to the LO-specification structure from which to obtain the estimated size.
estbytes	is a pointer to an eight-byte integer (**mi_int8**) value into which **mi_lo_specget_estbytes( )** puts the estimated number of bytes for the smart large object.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

The **mi_lo_specget_estbytes( )** function is the LO-specification accessor function that returns the estimated size from a set of storage characteristics. The *estbytes* value is the estimated final size, in bytes, of the smart large object. This estimate is an optimization hint for the smart-large-object optimizer.

Important: Before you call **mi_lo_specget_estbytes( )**, you must put storage characteristics into an LO-specification structure.

You can use any of the following functions to initialize the LO-specification structure:

- The **mi_lo_colinfo_by_ids( )** or **mi_lo_colinfo_by_name( )** function puts storage characteristics that are associated with a particular CLOB or BLOB column in an LO-specification structure.
- The **mi_lo_stat_cspec( )** function puts storage characteristics of an existing smart large object in an LO-specification.
- The **mi_lo_specset_estbytes( )** function sets the estimated size in an LO-specification structure.

The **mi_lo_specget_estbytes( )** function obtains the current value for the estimated size from the LO-specification structure that LO_*spec* references.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful; the LO-specification structure might be invalid.

Related Topics

See also the descriptions of **mi_lo_colinfo_by_ids( ), mi_lo_colinfo_by_name( ), mi_lo_spec_free( ), mi_lo_spec_init( ), mi_lo_specset_estbytes( ),** and **mi_lo_stat_cspec( ).**

For more information on the estimated size of a smart large object or on how to use the **mi_lo_specget_estbytes( )** function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_lo_specget_extsz( )

The **mi_lo_specget_extsz( )** function obtains from an LO-specification structure the allocation extent size, in kilobytes, of a smart large object.

Syntax

```
mi_integer mi_lo_specget_extsz(LO_spec)
   MI_LO_SPEC *LO_spec;
```

LO_spec	is a pointer to the LO-specification structure from which to obtain the extent size.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_specget_extsz( )** function is the LO-specification accessor function that returns the allocation extent size from a set of storage characteristics. The *extsz* value specifies the size, in kilobytes, of the allocation extents to be allocated for the smart large object when the smart-large-object optimizer writes beyond the end of the current extent.

Important: Before you call **mi_lo_specget_extsz( )**, you must put storage characteristics into an LO-specification structure.

You can use any of the following functions to initialize the LO-specification structure:

- The **mi_lo_colinfo_by_ids( )** or **mi_lo_colinfo_by_name( )** function puts storage characteristics that are associated with a particular CLOB or BLOB column in an LO-specification structure.
- The **mi_lo_stat_cspec( )** function puts storage characteristics of an existing smart large object in an LO-specification.
- The **mi_lo_specset_extsz( )** function sets the extent size in an LO-specification structure.

The **mi_lo_specget_extsz( )** function obtains the current value for the extent size from the LO-specification structure that LO_*spec* references.

Return Values

>=0	is the number of kilobytes in the extent size from the LO-specification structure that LO_*spec* references.
MI_ERROR	indicates that the function was not successful; the LO-specification structure might be invalid.

Related Topics

See also the descriptions of **mi_lo_colinfo_by_ids( ), mi_lo_colinfo_by_name( ), mi_lo_spec_free( ), mi_lo_spec_init( ), mi_lo_specset_extsz( ),** and **mi_lo_stat_cspec( ).**

For more information on the allocation extent size of a smart large object or on how to use the **mi_lo_specget_extsz( )** function, see the *IBM Informix DataBlade API*

Programmer's Guide.

mi_lo_specget_flags( )

The **mi_lo_specget_flags( )** function obtains from an LO-specification structure the attributes flag for a smart large object.

Syntax

```
mi_integer mi_lo_specget_flags(LO_spec)
   MI_LO_SPEC *LO_spec;
```

LO_spec	is a pointer to the LO-specification structure from which to obtain the flag value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_specget_flags( )** function is the LO-specification accessor function that returns the attributes flag from a set of storage characteristics. The attributes flag provides the following information about a smart large object:

- Whether to use logging on the smart large object
- Whether to store the time of last access for the smart large object
- Which data integrity to use for pages of the sbspace

Constants for these attributes are masked together into the single attributes-flag value. Therefore, to obtain a particular attribute, you must use the bitwise AND operator (&) to mask the attributes flag, as the following code fragment shows:

```
create_flags = mi_lo_specget_flags(LO_spec)
if ( create_flags & MI_LO_ATTR_LOG )
   /* logging is on */
```

Important: Before you call **mi_lo_specget_flags( )**, you must put storage characteristics into an LO-specification structure.

You can use any of the following functions to initialize the LO-specification structure:

- The **mi_lo_colinfo_by_ids( )** or **mi_lo_colinfo_by_name( )** function puts storage characteristics that are associated with a particular CLOB or BLOB column in an LO-specification structure.
- The **mi_lo_stat_cspec( )** function puts storage characteristics of an existing smart large object in an LO-specification.
- The **mi_lo_specset_flags( )** function sets the attributes flag in an LO-specification structure.

The **mi_lo_specget_flags( )** function obtains the current value for the attributes flag from the LO-specification structure that LO_*spec* references.

Return Values

>=0	is the bit mask for the attributes flags from the LO-specification structure that LO_*spec* references.
MI_ERROR	indicates that the function was not successful; the LO-specification structure might be invalid.

Related Topics

See also the descriptions of **mi_lo_colinfo_by_ids( ), mi_lo_colinfo_by_name( ), mi_lo_spec_free( ), mi_lo_spec_init( ), mi_lo_specset_flags( ),** and **mi_lo_stat_cspec( ).**

For more information on the attributes flag of a smart large object or on how to use the **mi_lo_specget_flags( )** function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_lo_specget_maxbytes( )

The **mi_lo_specget_maxbytes( )** accessor function obtains the maximum size of a smart large object (in bytes) from an LO-specification structure.

Syntax

```
mi_integer mi_lo_specget_maxbytes(LO_spec, maxbytes)
   MI_LO_SPEC *LO_spec;
   mi_int8 *maxbytes;
```

LO_spec	is a pointer to the LO-specification structure from which to obtain the maximum size.
maxbytes	is a pointer to an eight-byte integer (**mi_int8**) value into which **mi_lo_specget_maxbytes( )** puts the maximum size, in bytes, of the smart large object.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_specget_maxbytes( )** function is the LO-specification accessor function that returns the maximum size from a set of storage characteristics. The smart-large-object optimizer does not allow the size of a smart large object to exceed the *maxbytes* value.

Important: Before you call **mi_lo_specget_maxbytes( )**, you must put storage characteristics into an LO-specification structure.

You can use any of the following functions to initialize the LO-specification structure:

- The **mi_lo_colinfo_by_ids( )** or **mi_lo_colinfo_by_name( )** function puts storage characteristics that are associated with a particular CLOB or BLOB column in an LO-specification structure.
- The **mi_lo_stat_cspec( )** function puts storage characteristics of an existing smart large object in an LO-specification.
- The **mi_lo_specset_maxbytes( )** function sets the maximum size in an LO-specification structure.

The **mi_lo_specget_maxbytes( )** function obtains the current value for the maximum size from the LO-specification structure that LO_*spec* references.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful; the LO-specification structure might be invalid.

Related Topics

See also the descriptions of **mi_lo_colinfo_by_ids( ), mi_lo_colinfo_by_name( ), mi_lo_spec_free( ), mi_lo_spec_init( ), mi_lo_specset_maxbytes( ),** and **mi_lo_stat_cspec( ).**

For more information on the maximum size of a smart large object or on how to use the **mi_lo_specget_maxbytes( )** function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_lo_specget_sbspace( )

The **mi_lo_specget_sbspace( )** function obtains from an LO-specification structure the name of an sbspace where a smart large object is stored.

Syntax

```
mi_integer mi_lo_specget_sbspace(LO_spec, sbspace_name, length)
   MI_LO_SPEC *LO_spec;
   char *sbspace_name;
   mi_integer length;
```

LO_spec	is a pointer to the LO-specification structure from which to obtain the sbspace name.
sbspace_name	is a character buffer into which **mi_lo_specget_sbspace( )** puts the name of the sbspace for the smart large object.
length	is an integer value that specifies the size, in bytes, of the *sbspace_name* buffer.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

The **mi_lo_specget_sbspace( )** function is the LO-specification accessor function that returns the sbspace name from a set of storage characteristics. The function copies up to (*length*-1) bytes into the *sbspace_name* buffer and ensures that it is null terminated. An sbspace name can be up to 18 characters long. However, you might want to allocate at least 129 bytes for the *sbspace_name* buffer to accommodate future increases in the length of an sbspace name.

Important: Before you call **mi_lo_specget_sbspace( )**, you must put storage characteristics into an LO-specification structure.

You can use any of the following functions to initialize the LO-specification structure:

- The **mi_lo_colinfo_by_ids( )** or **mi_lo_colinfo_by_name( )** function puts storage characteristics that are associated with a particular CLOB or BLOB column in an LO-specification structure.
- The **mi_lo_stat_cspec( )** function puts storage characteristics of an existing smart large object in an LO-specification.
- The **mi_lo_specset_sbspace( )** function sets the sbspace name in an LO-specification structure.

The **mi_lo_specget_sbspace( )** function obtains the current value for the name of the sbspace from the LO-specification structure that LO_*spec* references.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful; the LO-specification structure might be invalid.

Related Topics

See also the descriptions of **mi_lo_colinfo_by_ids( ), mi_lo_colinfo_by_name( ), mi_lo_spec_free( ), mi_lo_spec_init( ), mi_lo_specset_sbspace( ),** and **mi_lo_stat_cspec( ).**

For more information on the sbspace name of a smart large object or on how to use the **mi_lo_specget_sbspace( )** function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_lo_specset_def_open_flags( )

The **mi_lo_specset_def_open_flags( )** function sets the default open-mode flag for a smart large object.

Syntax

```
mi_integer mi_lo_specset_def_open(LO_spec, def_open_flags)
   MI_LO_SPEC *LO_spec;
   mi_integer def_open_flags;
```

LO_spec	is a pointer to the LO-specification structure in which to save the default open flags.
def_open_flags	is an integer value that specifies the bit mask of the default open flags for the smart large object.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_specset_def_open_flags( )** function is the LO-specification accessor function that sets the default open flags for a new smart large object. The *def_open_flags* value is the bit mask that indicates which open flags to use when the **mi_lo_open( )** opens the smart large object. The default open flags indicate how the **mi_lo_open( )** function opens the smart large object when it does not include an open flag. To override the default open flags, you can specify an open mode as an argument to **mi_lo_open( ).**

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_lo_copy( ), mi_lo_create( ), mi_lo_from_file( ), mi_lo_open( ), mi_lo_spec_free( ), mi_lo_spec_init( ),** and **mi_lo_specget_def_open_flags( ).**

For more information on the default open mode of a smart large object or on how to use the **mi_lo_specset_def_open_flags( )** function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_lo_specset_estbytes( )

The **mi_lo_specset_estbytes( )** function sets the estimated size of a smart large object.

Syntax

```
mi_integer mi_lo_specset_estbytes(LO_spec, estbytes)
   MI_LO_SPEC *LO_spec;
   mi_int8 *estbytes;
```

LO_spec	is a pointer to the LO-specification structure in which to save the estimated size.
estbytes	is a pointer to an eight-byte integer (**mi_int8**) that contains the desired estimated number of bytes for the smart large object.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

The **mi_lo_specset_estbytes( )** function is the LO-specification accessor function that sets the estimated size for a new smart large object. The *estbytes* value is the estimated final size, in bytes, of the smart large object. This estimate is an optimization hint for the smart-large-object optimizer.

Important: Before you call **mi_lo_specset_estbytes( )**, you must initialize an LO-specification structure.

After you set the estimated size in an LO-specification structure, you pass this structure to a smart-large-object creation function (such as **mi_lo_create( )**) to provide the estimated size as a user-supplied storage characteristic for a new smart large object.

The smart-large-object optimizer attempts to optimize the extent size based on past operations on the smart large object and other storage characteristics (such as maximum bytes) that it obtains from the storage-characteristics hierarchy. Most applications can use the size estimate that the smart-large-object optimizer generates.

Important: Do *not* specify an estimated size unless you have enough information about the data to provide a useful estimate. If you do set the estimated size for a smart large object, do not specify a value much higher than the final size of the smart large object. Otherwise, the database server might allocate unused storage.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_lo_copy( ), mi_lo_create( ), mi_lo_from_file( ), mi_lo_spec_free( ), mi_lo_spec_init( ),** and **mi_lo_specget_estbytes( ).**

For more information on the estimated size of a smart large object or on how to use the **mi_lo_specset_estbytes( )** function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_lo_specset_extsz( )

The **mi_lo_specset_extsz( )** function sets the allocation extent size for a smart large object.

Syntax

```
mi_integer mi_lo_specget_extsz(LO_spec, extsz)
   MI_LO_SPEC *LO_spec;
   mi_integer extsz;
```

LO_spec	is a pointer to the LO-specification structure in which to save the extend size.
extsz	is an integer value for the size, in kilobytes, of the allocation extent of a smart large object.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

The **mi_lo_specset_extsz( )** function is the LO-specification accessor function that sets the allocation extent size for a new smart large object. The *extsz* value specifies the size of the allocation extents to be allocated for the smart large object when the smart-large-object optimizer writes beyond the end of the current extent.

Important: Before you call **mi_lo_specset_extsz( )**, you must initialize an LO-specification structure.

When you set the extent size with **mi_lo_specset_extsz( )**, you override any column-level or system-specified extent size in the LO-specification structure. You then pass this LO-specification structure to a smart-large-object creation function (such as **mi_lo_create( )**) to provide the extent size as a user-supplied storage characteristic for a new smart large object.

The smart-large-object optimizer attempts to optimize the extent size based on past operations on the smart large object and other storage characteristics (such as maximum bytes) that it obtains from the storage-characteristics hierarchy. Most applications can use this generated extent size.

Important: Do *not* change the system-specified extent size unless your application encounters severe storage fragmentation. For such applications, make sure that you know exactly the number of bytes by which to extend the smart large object.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_lo_copy( ), mi_lo_create( ), mi_lo_from_file( ), mi_lo_spec_free( ), mi_lo_spec_init( ),** and **mi_lo_specget_extsz( ).**

For more information on the allocation extent size of a smart large object or on how to use the **mi_lo_specset_extsz( )** function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_lo_specset_flags( )

The **mi_lo_specset_flags( )** function sets the attributes flag for a smart large object.

Syntax

```
mi_integer mi_lo_specset_flags(LO_spec, flags)
   MI_LO_SPEC *LO_spec;
   mi_integer flags;
```

LO_spec	is a pointer to the LO-specification structure in which to save the flags value.
flags	is an integer value for the bit mask of the attributes flag for the smart large object.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_specset_flags( )** function is the LO-specification accessor function that sets the attributes flag for a new smart large object. The attributes flag provides the following information about a smart large object:

- Whether to use logging on the smart large object
- Whether to store the time of last access for the smart large object

Constants for these attributes are masked together into the single attributes-flag value. Therefore, to set a particular attribute, you must use the bitwise OR operator (|) to mask the attributes flag, as the following code fragment shows:

```
create_flags = MI_LO_ATTR_LOG | MI_LO_ATTR_KEEP_LASTACCESS_TIME
mi_lo_specset_flags(LO_spec, create_flags)
```

Important: Before you call **mi_lo_specset_flags( )**, you must initialize an LO-specification structure.

When you set the attributes flag with **mi_lo_specset_flags( )**, you override any column-level or system-specified attributes flag in the LO-specification structure. You then pass this LO-specification structure to a smart-large-object creation function (such as **mi_lo_create( )**) to provide the attributes flag as a user-supplied storage characteristic for a new smart large object.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_lo_copy( ), mi_lo_create( ), mi_lo_from_file( ), mi_lo_spec_free( ), mi_lo_spec_init( ),** and **mi_lo_specget_flags( ).**

For more information on the attributes flag of a smart large object or on how to use the **mi_lo_specset_flags( )** function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_lo_specset_maxbytes( )

The **mi_lo_specset_maxbytes( )** function sets the maximum number of bytes allowed for a smart large object.

Syntax

```
mi_integer mi_lo_specset_maxbytes(LO_spec, maxbytes)
   MI_LO_SPEC *LO_spec;
   mi_int8 *maxbytes;
```

LO_spec	is a pointer to the LO-specification structure in which to save the maximum size.
maxbytes	is a pointer to an eight-byte integer (**mi_int8**) structure that contains the maximum number of bytes for the smart large object.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_specset_maxbytes( )** function is the LO-specification accessor function that sets the maximum size of a new smart large object. When you set a maximum size, the smart-large-object optimizer does not allow the size of a smart large object to exceed the *maxbytes* value. Most applications do not need to specify a maximum size.

Important: Before you call **mi_lo_specset_maxbytes( )**, you must initialize an LO-specification structure.

After you set the maximum size in an LO-specification structure, you pass this structure to a smart-large-object creation function (such as **mi_lo_create( )**) to provide the maximum size as a user-supplied storage characteristic for a new smart large object.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_lo_copy( ), mi_lo_create( ), mi_lo_from_file( ), mi_lo_spec_free( ), mi_lo_spec_init( ),** and **mi_lo_specget_maxbytes( ).**

For more information on the maximum size of a smart large object or on how to use the **mi_lo_specset_maxbytes( )** function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_lo_specset_sbspace( )

The **mi_lo_specset_sbspace( )** function sets the sbspace name of a smart large object in an LO-specification structure.

Syntax

```
mi_integer mi_lo_specset_sbspace(LO_spec, sbspace_name)
   MI_LO_SPEC *LO_spec;
   const char *sbspace_name;
```

LO_spec	is a pointer to the LO-specification structure in which to save the sbspace name.
sbspace_name	is a pointer to the sbspace name for the smart large object.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_specset_sbspace( )** function is the LO-specification accessor function that sets the sbspace name for a new smart large object. The name of the sbspace can be at most 18 characters long and must be null terminated.

Important: Before you call **mi_lo_specset_sbspace( )**, you must initialize an LO-specification structure.

When you set the sbspace name with **mi_lo_specset_sbspace( )**, you override any column-level or system-specified sbspace name in the LO-specification structure. You then pass this LO-specification structure to a smart-large-object creation function (such as **mi_lo_create( )**) to provide the sbspace name as a user-supplied storage characteristic for a new smart large object.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_lo_copy( ), mi_lo_create( ), mi_lo_from_file( ), mi_lo_spec_free( ), mi_lo_spec_init( ),** and **mi_lo_specget_sbspace( ).**

For more information on the sbspace name of a smart large object or on how to use the **mi_lo_specset_sbspace( )** function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_lo_stat( )

The **mi_lo_stat( )** function puts information about the current status of an open smart large object into an LO-status structure.

Syntax

```
mi_integer mi_lo_stat(conn, LO_fd, LOstat_dptr)
   MI_CONNECTION *conn;
   MI_LO_FD LO_fd;
   MI_LO_STAT **LOstat_dptr;
```

conn	is one of the following connection values:
	A pointer to a connection descriptor established by a previous call to **mi_lo_open( ), mi_server_connect( ),** or **mi_server_reconnect( )**
	A NULL-valued pointer (database server only)
LO_fd	is an LO file descriptor for the open smart large object whose status information you want to obtain.
LOstat_dptr	is a doubly indirected pointer to the LO-status structure for the smart large object.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_stat( )** function performs the following steps to obtain an LO-status structure:

1. It handles memory allocation if it creates a new LO-status structure.

   When the LO*stat_dptr* value (a single indirect pointer to an LO-status structure) is NULL, the **mi_lo_stat( )** function allocates an LO-status structure. Before you use an LO-status structure, set LO*stat_dptr* to NULL so that **mi_lo_stat( )** allocates space for the LO-status structure. When LO*stat_dptr* does not point to NULL, the **mi_lo_stat( )** function assumes that the LO-status structure has already been allocated by a previous call to **mi_lo_stat( )**.
2. It initializes the fields in the LO-status structure with the status information for the smart large object that the LO*_fd* file descriptor identifies.

   To access the status information, use the LO-status accessor functions. For more information on the status information and the corresponding accessor functions, see the *IBM Informix DataBlade API Programmer's Guide*.

The **mi_lo_stat( )** function is the constructor for the LO-status structure.

Server Only

The **mi_lo_stat( )** function allocates a new LO-status structure in the current memory duration.

End of Server Only

Important: You must call the **mi_lo_stat( )** function before you use an LO-status structure in a DataBlade API module.

Do *not* use system memory-allocation calls (such as **malloc( )** or **mi_alloc( )**) to perform memory management for LO-status structures. Use the **mi_lo_stat( )** function to create a new LO-specification structure and the **mi_lo_stat_free( )** function to free an LO-specification structure.

Server Only

The **mi_lo_stat( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_lo_stat_atime( ), mi_lo_stat_cspec( ), mi_lo_stat_ctime( ), mi_lo_stat_free( ), mi_lo_stat_mtime_sec( ), mi_lo_stat_mtime_usec( ), mi_lo_stat_refcnt( ),** and **mi_lo_stat_size( ).**

mi_lo_stat_atime( )

The **mi_lo_stat_atime( )** function returns from an LO-status structure the last-access time for a smart large object.

Syntax

```
mi_integer mi_lo_stat_atime(LO_stat)
   MI_LO_STAT *LO_stat;
```

LO_stat	is a pointer to an LO-status structure that **mi_lo_stat( )** allocates and fills in with status information.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

The **mi_lo_stat_atime( )** function obtains the current value for the last access time from the LO-status structure that *LO_stat* references. The resolution of the last-access time that **mi_lo_stat_atime( )** returns is the number of seconds since 00:00:00, January 1, 1970.

Important: Before you call **mi_lo_stat_atime( )**, you must initialize an LO-status structure with the **mi_lo_stat( )** function.

Return Values

>=0	is the number of seconds since January 1, 1970, for the last-access time of the smart large object whose status information is in the LO-status structure that LO_*stat* references.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_lo_stat( ), mi_lo_stat_cspec( ), mi_lo_stat_ctime( ), mi_lo_stat_free( ), mi_lo_stat_mtime_sec( ), mi_lo_stat_mtime_usec( ), mi_lo_stat_refcnt( ),** and **mi_lo_stat_size( ).**

For more information about the last-access time of a smart large object or on how to use an LO-status structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_lo_stat_cspec( )

The **mi_lo_stat_cspec( )** function returns from an LO-status structure the storage characteristics for a smart large object.

Syntax

```
MI_LO_SPEC *mi_lo_stat_cspec(LO_stat)
   MI_LO_STAT *LO_stat;
```

LO_stat	is a pointer to an LO-status structure that **mi_lo_stat( )** allocates and fills in with status information.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_stat_cspec( )** function returns a pointer to an LO-specification structure that contains the storage characteristics from the LO-status structure that LO_*stat* references. To access storage characteristics from this structure, use the LO-specification accessor functions.

Important: Before you call **mi_lo_stat_cspec( )**, you must initialize an LO-status structure with the **mi_lo_stat( )** function.

You can use the LO-specification structure that **mi_lo_stat_cspec( )** returns to create another smart large object with the same storage characteristics.

Return Values

An **MI_LO_SPEC** pointer	is a pointer to the LO-specification structure that contains the storage characteristics for the smart large object whose status information is in the LO-status structure that LO_*stat* references.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_lo_stat( ), mi_lo_stat_atime( ), mi_lo_stat_ctime( ), mi_lo_stat_free( ), mi_lo_stat_mtime_sec( ), mi_lo_stat_mtime_usec( ), mi_lo_stat_refcnt( ),** and **mi_lo_stat_size( ).**

For more information on storage characteristics of a smart large object, on how to use an LO-status structure, or on how to use the **mi_lo_stat_cspec( )** function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_lo_stat_ctime( )

The **mi_lo_stat_ctime( )** function returns from an LO-status structure the last-change time for a smart large object.

Syntax

```
mi_integer mi_lo_stat_ctime(LO_stat)
   MI_LO_STAT *LO_stat;
```

LO_stat	is a pointer to an LO-status structure that **mi_lo_stat( )** allocates and fills in with status information.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_stat_ctime( )** function obtains the last-change time from the LO-status structure that LO_*stat* references. The last-change time includes changes to metadata (modification of storage characteristics, a change in the number of references) and user data (writes to the smart large object). The resolution of the last-change time that the **mi_lo_stat_ctime( )** function returns is number of seconds since 00:00:00, January 1, 1970.

Important: Before you call **mi_lo_stat_ctime( )**, you must initialize an LO-status structure with the **mi_lo_stat( )** function.

Return Values

>=0	is the number of seconds since January 1, 1970 for the last-change time of the smart large object whose status information is in the LO-status structure that LO_*stat* references.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_lo_stat( ), mi_lo_stat_atime( ), mi_lo_stat_cspec( ), mi_lo_stat_free( ), mi_lo_stat_mtime_sec( ), mi_lo_stat_mtime_usec( ), mi_lo_stat_refcnt( ),** and **mi_lo_stat_size( ).**

For more information about the last-change time of a smart large object or about how to use an LO-status structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_lo_stat_free( )

The **mi_lo_stat_free( )** function frees an LO-status structure.

Syntax

```
mi_integer mi_lo_stat_free(conn, LO_stat)
   MI_CONNECTION *conn;
   MI_LO_STAT *LO_stat;
```

conn is one of the following connection values:

A pointer to a connection descriptor established by a previous call to **mi_lo_open( ), mi_server_connect( ),** or **mi_server_reconnect( )**

A NULL-valued pointer (database server only)

LO_stat is a pointer to an LO-status structure that **mi_lo_stat_free( )** deallocates.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_stat_free( )** function frees the LO-status structure that LO_*stat* references. This function is the destructor for the LO-status structure.

Important: Do *not* use system memory-allocation calls (such as **free( )** or **mi_free( )**) to perform memory management for LO-status structures.

When your application no longer needs status information, call **mi_lo_stat_free( )** for each LO-status structure that the **mi_lo_stat( )** function has allocated. Once freed, these resources can be reallocated to other structures.

Warning: Do not call the **mi_lo_stat_free( )** function for the same LO-status structure more than once. This behavior is analogous to the behavior of the **free( )** system function for regular memory allocation.

Server Only

The **mi_lo_stat_free( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful. One of the arguments is invalid.

Related Topics

See also the descriptions of **mi_lo_stat( ), mi_lo_stat_atime( ), mi_lo_stat_cspec( ), mi_lo_stat_ctime( ), mi_lo_stat_mtime_sec( ), mi_lo_stat_mtime_usec( ), mi_lo_stat_refcnt( ),** and **mi_lo_stat_size( ).**

mi_lo_stat_mtime_sec( )

The **mi_lo_stat_mtime_sec( )** function returns from an LO-status structure the last-modification time, in seconds, of a smart large object.

Syntax

```
mi_integer mi_lo_stat_mtime_sec(LO_stat)
   MI_LO_STAT *LO_stat;
```

LO_stat	is a pointer to an LO-status structure that **mi_lo_stat( )** allocates and fills in with status information.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_stat_mtime_sec( )** function obtains the last-modification time from the LO-status structure that LO_*stat* references. The last-modification time includes changes to user data (writes to the smart large object) only. The resolution of the last-modification time that the **mi_lo_stat_mtime_sec( )** function returns is number of seconds since 00:00:00, January 1, 1970. On some platforms, you can obtain the microsecond component of the last-modification time with the **mi_lo_stat_mtime_usec( )** function. For more information, see the description of **mi_lo_stat_mtime_usec( ).**

Important: Before you call **mi_lo_stat_mtime_sec( )**, you must initialize an LO-status structure with the **mi_lo_stat( )** function.

Return Values

>=0	is the number of seconds since 00:00:00, January 1, 1970, for the last-modification time of the smart large object whose status information is in the LO-status structure that LO_*stat* references.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_lo_stat( ), mi_lo_stat_atime( ), mi_lo_stat_cspec( ), mi_lo_stat_ctime( ), mi_lo_stat_free( ), mi_lo_stat_mtime_sec( ), mi_lo_stat_refcnt( ),** and **mi_lo_stat_size( ).**

For more information about the last-modification time of a smart large object or about how to use an LO-status structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_lo_stat_mtime_usec( )

The **mi_lo_stat_mtime_usec( )** function returns from an LO-status structure the microsecond component of the last-modification time for a smart large object.

Syntax

```
mi_integer mi_lo_stat_mtime_usec(LO_stat)
   MI_LO_STAT *LO_stat;
```

LO_stat	is a pointer to an LO-status structure that **mi_lo_stat( )** allocates and fills in with status information.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_stat_mtime_usec( )** function obtains the microsecond component of the last-modification time from the LO-status structure that LO_*stat* references. The last-modification time includes changes to user data (writes to the smart large object) only.

Important: The database server does not maintain the microsecond component of last-modification time. If your platform supports the microsecond component of system time and you choose to maintain it for smart large objects, you must explicitly set the last-modification microsecond value with **mi_lo_utimes( ).**

If the microsecond component of last-modification time is supported and maintained on your system, the **mi_lo_stat_mtime_usec( )** function can obtain it from an initialized LO-status structure. To return the seconds component of the time of last modification, use the **mi_lo_stat_mtime_sec( )** function. The database server does maintain this seconds component of the last-modification time.

Important: Before you call **mi_lo_stat_mtime_usec( )**, you must initialize an LO-status structure with the **mi_lo_stat( )** function.

Return Values

>=0	is the number of microseconds in the last-modification time for the smart large object that LO_*stat* references.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_lo_stat( ), mi_lo_stat_atime( ), mi_lo_stat_cspec( ), mi_lo_stat_ctime( ), mi_lo_stat_free( ), mi_lo_stat_mtime_sec( ), mi_lo_stat_refcnt( ), mi_lo_stat_size( ),** and **mi_lo_utimes( ).**

For more information about the last-modification time of a smart large object or about how to use an LO-status structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_lo_stat_refcnt( )

The **mi_lo_stat_refcnt( )** function returns from an LO-status structure the reference count of a smart large object.

Syntax

```
mi_integer mi_lo_stat_refcnt(LO_stat)
   MI_LO_STAT *LO_stat;
```

LO_stat	is a pointer to an LO-status structure that **mi_lo_stat( )** allocates and fills in with status information.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_stat_refcnt( )** function is the LO-status accessor function that returns the reference count from a set of status information. The reference count for a smart large object indicates the number of persistently stored LO handles that currently exist for the smart large object. The database server assumes that it can safely remove the smart large object and reuse any resources that are allocated to it when the reference count is zero (0) and any of the following conditions exists:

- The transaction in which the reference count was decremented commits.
- The connection terminates and the smart large object was created during this connection, but its reference count was never incremented.

  The database server increments a reference count when it stores the LO handle for a smart large object in a row.

Important: Before you call **mi_lo_stat_refcnt( )**, you must initialize an LO-status structure with the **mi_lo_stat( )** function.

The **mi_lo_stat_refcnt( )** function obtains the reference count from the LO-status structure that LO_*stat* references.

Return Values

>=0	is the reference count for the smart large object whose status information is in the LO-status structure that LO_*stat* references.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_lo_decrefcount( ), mi_lo_increfcount( ), mi_lo_stat( ), mi_lo_stat_atime( ), mi_lo_stat_cspec( ), mi_lo_stat_ctime( ), mi_lo_stat_free( ), mi_lo_stat_mtime_sec( ), mi_lo_stat_mtime_usec( ),** and **mi_lo_stat_size( ).**

For more information about the reference count of a smart large object or about how to use an LO-status structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_lo_stat_size( )

The **mi_lo_stat_size( )** function obtains from an LO-status structure the size, in bytes, of a smart large object.

Syntax

```
mi_integer *mi_lo_stat_size(LO_stat, size)
   MI_LO_STAT *LO_stat;
   mi_int8 *size;
```

LO_stat	is a pointer to an LO-status structure that **mi_lo_stat( )** allocates and fills in with status information.
size	is a pointer to a user-allocated **mi_int8** structure to receive the size, in bytes, of the smart large object.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_stat_size( )** function is the LO-status accessor function that returns the smart-large-object size from a set of status information. This size is the actual number of bytes that the smart-large-object data currently uses.

Important: Before you call **mi_lo_stat_size( )**, you must initialize an LO-status structure with the **mi_lo_stat( )** function.

The **mi_lo_stat_size( )** function obtains the smart-large-object size from the LO-status structure that LO_*stat* references.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_lo_stat( ), mi_lo_stat_atime( ), mi_lo_stat_cspec( ), mi_lo_stat_ctime( ), mi_lo_stat_free( ), mi_lo_stat_mtime_sec( ), mi_lo_stat_mtime_usec( ),** and **mi_lo_stat_refcnt( ).**

For more information about the size of a smart large object or about how to use an LO-status structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_lo_stat_uid( )

The **mi_lo_stat_uid( )** function returns from an LO-status structure the user identifier of the owner of a smart large object.

Syntax

```
mi_integer mi_lo_stat_uid(LO_stat)
   MI_LO_STAT *LO_stat;
```

LO_stat	is a pointer to an LO-status structure that **mi_lo_stat( )** allocates and fills in with status information.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Return Values

>=0	The user identifier (user ID) of the owner for the smart large object that LO_*stat* references.
MI_ERROR	The function was not successful.

Related Topics

See also the description of **mi_lo_stat( ).**

mi_lo_tell( )

The **mi_lo_tell( )** function returns the current LO seek position for an open smart large object, relative to the beginning of the smart large object.

Syntax

```
mi_integer mi_lo_tell(conn, LO_fd, seek_pos)
   MI_CONNECTION *conn;
   MI_LO_FD LO_fd;
   mi_int8 *seek_pos;
```

conn	is one of the following connection values: A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )** A NULL-valued pointer (database server only)
LO_fd	is an LO file descriptor for the open smart large object whose LO seek position you want to determine.
seek_pos	is a pointer to the eight-byte integer (**mi_int8**) into which **mi_lo_tell( )** copies the current LO seek position.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_tell( )** function obtains the LO seek position for the smart large object that LO*_fd* references. The LO *seek position* is the offset for the next read or write operation on the smart large object that is associated with the LO file descriptor, LO*_fd*. The **mi_lo_tell( )** function returns this seek position in the **mi_int8** *seek_pos* variable. For more information on how to access eight-bit (INT8) integers, see the *IBM Informix DataBlade API Programmer's Guide*.

Server Only

The **mi_lo_tell( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_lo_read( ), mi_lo_readwithseek( ), mi_lo_seek( ), mi_lo_write( ),** and **mi_lo_writewithseek( ).**

mi_lo_to_buffer( )

The **mi_lo_to_buffer( )** function copies a specified number of bytes from a smart large object to a user-defined buffer.

Syntax

```
mi_integer mi_lo_to_buffer(conn, LO_hdl, size, buf_ptr)
   MI_CONNECTION *conn;
   MI_LO_HANDLE *LO_hdl;
   mi_integer size;
   char **buf_ptr;
```

conn	is one of the following connection values: A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )** A NULL-valued pointer (database server only)
LO_hdl	is the LO handle for the smart large object from which you want to copy the data.
size	is an integer number of bytes to copy from the smart large object.
buf_ptr	is a doubly indirected pointer to a user-defined buffer to which you want to copy the data.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_to_buffer( )** function copies up to *size* bytes from the smart large object that the LO_*hdl* LO handle references. The read operation from the smart large object starts at a zero-byte offset. This function allows you to read data from a smart large object without opening the smart large object.

If the smart large object is smaller than the *size* value, **mi_lo_to_buffer( )** copies only the number of bytes in the smart large object. If the smart large object contains more than *size* bytes, the **mi_lo_to_buffer( )** function copies only *size* bytes to the user-defined *buffer*.

When *buf_ptr* points to NULL, **mi_lo_to_buffer( )** allocates the memory for the buffer. Otherwise, the function assumes that you have allocated memory that *buf_ptr* references.

Server Only

The **mi_lo_to_buffer( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

>=0	is the number of bytes copied from the smart large object to the user-defined buffer that *buf_ptr* references.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_lo_from_buffer( )** and **mi_lo_to_file( ).**

mi_lo_to_file( )

The **mi_lo_to_file( )** function copies a smart large object to an operating-system file on the client or server computer.

Syntax

```
const char *mi_lo_to_file(conn, LO_hdl, fname_spec, open_mode, size)
   MI_CONNECTION *conn;
   MI_LO_HANDLE *LO_hdl;
   const char *fname_spec;
   mi_integer open_mode;
   mi_integer *size;
```

conn	is one of the following connection values: A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )** A NULL-valued pointer (database server only)
LO_hdl	is a pointer to an LO handle of the smart large object to copy.
fname_spec	is a pathname template for the target file that holds the data. This pathname can include special symbols for the filename.
open_mode	is an integer bit mask to indicate how to open the operating-system file and where this file is located. For a list of valid file-mode constants, see the table in the following "Usage" section.
size	is a pointer to the size of the file after **mi_lo_to_file( )** completes the copy.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_to_file( )** function can create the target files on either the server or the client computer. The flag values for the *open_mode* argument indicate the location and access mode of the operating-system file to copy.

Valid values include the following file-mode constants.

File-Mode Constant	Purpose
MI_O_EXCL	Raise an exception if a file by that name already exists.
MI_O_TRUNC	Truncate the file, if it already exists.
MI_O_APPEND	Append to the file, if it already exists.
MI_O_RDWR	Open the file in read/write mode.
MI_O_WRONLY	Open the file in write-only mode.
MI_O_BINARY	Process the data as binary data.
MI_O_TEXT	Process the data as text (not binary). (Binary is used if you do not specify MI_O_TEXT.)
MI_O_SERVER_FILE	The *fname_spec* file is created on the server

	computer. The file mode is read/write for all users. The file owner is the client user ID.
MI_O_CLIENT_FILE	The *fname_spec* file is created on the client computer. The file owner is the client user and file permissions will be consistent with the client user's umask setting.

The default value for the *open_mode* argument is the masking of the following flag values:

```
MI_O_CLIENT_FILE | MI_O_WRONLY | MI_O_TRUNC | MI_O_BINARY
```

By default, the **mi_lo_to_file( )** function generates a filename in the following form:

```
fname_spec.hex_id
```

In this format, *fname_spec* is the filename that you specify as an argument to **mi_lo_to_file( )**, and *hex_id* is the unique hexadecimal smart-large-object identifier. The maximum number of digits for a smart-large-object identifier is 16; however, most smart large objects have an identifier with fewer significant digits.

For example, suppose you specify an *fname_spec* value as follows:

```
'/tmp/resume'
```

If the LO handle has an identifier of `00000000000203b2`, the **mi_lo_to_file( )** function creates the following file:

```
/tmp/resume.00000000000203b2
```

To change this default filename, you can specify the following wildcard symbols in the filename portion of *fname_spec*:

- One or more contiguous question mark (?) characters in the filename preserve digits of the LO-handle identifier.

  The **mi_lo_to_file( )** function replaces each question mark with a hexadecimal digit from the identifier of the LO handle. Substitution is right to left. Question marks need not be contiguous. If you specify more than 16 question marks, the **mi_lo_to_file( )** function ignores them.
- An exclamation point (!) at the end of the filename indicates that the filename does not need to be unique.

  The **mi_lo_to_file( )** function omits the exclamation point from the result and does not substitute any characters. The exclamation point overrides the question marks in the filename specification.

Tip: These wildcards are also valid in the "fname_spec" argument of the **mi_lo_filename( )** function.

The following table shows some examples of wildcard substitution when the hexadecimal identifier for the LO handle of a smart large object is `0000000000000019`.

Filename Specification	Actual Filename
x!	**x**
resume.txt!	**resume.txt**
x	**x.0000000000000019**

?resume	**9resume**
resume??.txt	resume19.txt
resume???.???	**resume000.019**
?abc???.???	**0abc000.019**
???a???.???	**000a000.019**
???a???.???b	**000a000.019b**
???a???.???b!	**???a???.???b**
???a???.???b??????????????	**???a???.?00b00000000000019**

If an exception because of file I/O problems occurs, the action that the database server takes depends on the file location set in the *flags* argument:
- If MI_O_SERVER_FILE is set, **errno** is set.
- If MI_O_CLIENT_FILE is set, the exception is raised within the database server.

Server Only

The **mi_lo_to_file( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

The **mi_lo_to_file( )** function is useful in export functions for opaque data types that contain smart large objects.

End of Server Only

Return Values

A **char** pointer	is a pointer to the filename that results from the wildcard expansion.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_lo_filename( ), mi_lo_from_file( ),** and **mi_lo_to_buffer( ).**

For more information, including information on the use of **mi_lo_to_file( )** in an export opaque-type support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_lo_to_string( )

The **mi_lo_to_string( )** function converts an LO handle in binary representation to its text representation.

Syntax

```
char *mi_lo_to_string(LO_hdl)
   MI_LO_HANDLE *LO_hdl;
```

LO_hdl	is a pointer to the LO handle to convert to text representation.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_to_string( )** function converts the LO handle that *LO_hdl* references to its text representation. The **mi_lo_to_string( )** function allocates a character string with the current memory duration. Use **mi_free( )** to free this character string when it is no longer needed.

Return Values

A **char** pointer	is a pointer to the text representation of the LO handle that LO_*hdl* references.
NULL	indicates that the function was not successful.

Related Topics

See also the description of **mi_lo_from_string( ).**

mi_lo_truncate( )

The **mi_lo_truncate( )** function truncates a smart large object at a specified byte position.

Syntax

```
mi_integer mi_lo_truncate(conn, LO_fd, offset)
   MI_CONNECTION *conn;
   MI_LO_FD LO_fd;
   mi_int8 *offset;
```

conn	is one of the following connection values:
	A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )**
	A NULL-valued pointer (database server only)
LO_fd	is an LO file descriptor for the open smart large object whose value you want to truncate.
offset	is a pointer to the eight-byte integer (**mi_int8**) that identifies the offset at which the truncation of the smart large object begins.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_truncate( )** function sets the last valid byte of a smart large object to the specified *offset* value. If this *offset* value is less than the current end of the smart large object, the database server reclaims all storage, from the position that *offset* indicates to the end of the smart large object.

Server Only

The **mi_lo_truncate( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

MI_OK	The function was successful.
MI_ERROR	The function was not successful.

Related Topics

See also the description of **mi_lo_open( ).**

mi_lo_unlock( )

The **mi_lo_unlock( )** releases a byte-range lock on a smart large object.

Syntax

```
mi_integer mi_lo_unlock(conn, LO_fd, offset, whence, nbytes)
   MI_CONNECTION *conn;
   MI_LO_FD LO_fd;
   mi_int8 *offset;
   mi_integer whence;
   mi_int8 *nbytes;
```

conn	is one of the following connection values:
	A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )**
	A NULL-valued pointer (database server only)
LO_fd	is an LO file descriptor for the open smart large object whose bytes you want to unlock.
offset	is a pointer to the eight-byte integer (**mi_int8**) that identifies the offset at which the unlock of the smart-large-object bytes begins.
whence	is an integer value that identifies the starting LO seek position.
nbytes	is a pointer to the eight-byte integer (**mi_int8**) that specifies the number of bytes to unlock. This value cannot exceed two gigabytes.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_unlock( )** function removes a byte-range lock on the open smart large object that *LO_fd* indicates. This unlock request applies to *nbytes* number of bytes beginning at the LO seek location that the *whence* and *offset* arguments specify.

When the **mi_lo_unlock( )** function requests the release of a byte-range lock, the database server attempts to unlock a locked smart large object if it has a share-mode or update-mode lock. For exclusive locks, the database server does not permit the release of the lock until the end of the transaction.

Server Only

The **mi_lo_unlock( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

MI_OK	indicates that the function was successful; the requested lock was successfully released.

MI_ERROR	indicates that the function was *not* successful; the requested lock could not be released or the smart large object has not been opened for byte-range locking.

Related Topics

See also the descriptions of **mi_lo_lock( ), mi_lo_open( ),** and **mi_lo_specset_def_open_flags( ).**

For more information on locks for smart large objects or on how to use byte-range locks, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_lo_utimes( )

The **mi_lo_utimes( )** function enables you to set the last-access and last-modification time of a smart large object.

Syntax

```
mi_integer mi_lo_utimes(conn, LO_hdl, access_sec, access_usec, mod_sec,
   mod_usec)
   MI_CONNECTION *conn;
   MI_LO_HANDLE *LO_hdl;
   mi_integer access_sec;
   mi_integer access_usec;
   mi_integer mod_sec;
   mi_integer mod_usec;
```

conn	is one of the following connection values: A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )** A NULL-valued pointer (database server only)
LO_hdl	is a pointer to an LO handle that identifies the smart large object whose time is to be updated.
access_sec	is an integer value that specifies the seconds component of the new last-access time. A value of zero (0) tells the database server to set the last-access time to the current time.
access_usec	is an integer value that specifies the microseconds component of the new last-access time. This parameter is not currently implemented and must be set to zero (0).
mod_sec	is an integer value that specifies the seconds component of the new last-modification time. A value of zero (0) for *mod_sec* and *mod_usec* tells the database server to set the seconds component of the last-modification time to the current time and the microsecond component of this time to zero (0).
mod_usec	is an integer value that specifies the microseconds component of the new last-modification time. A value of zero (0) for *mod_sec* and *mod_usec* tells the database server to set the seconds component of the last-modification time to the current time and the microsecond component of this time to zero (0). The database server does *not* maintain the microsecond component of the last-modification time. You can use **mi_lo_utimes( )** to maintain this value for your smart large object.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Warning: This advanced function can adversely affect your UDR if you use the function incorrectly. Use it only when no regular DataBlade API function can perform the task you need done.

Usage

The **mi_lo_utimes( )** function updates the last-access and last-modification time of the smart large object that the *LO_hdl* LO handle identifies. The last-access time is the time that the smart large object was last accessed (last read or write operation). The last-modification time is the time that the smart large object was last modified (last write operation on user data of a smart large object).

The **mi_lo_utimes( )** function performs an operation on a smart large object analogous to the operation that the UNIX or Linux **touch** command performs on an operating-system file. This function is valid on UNIX, Linux, and Windows.

You can use **mi_lo_utimes( )** to set the last-access and last-modification time to either the current time or a specified time, as follows.

Updated Time	Arguments for Current System Time	Arguments for Specified System Time
Last-access time	*access_sec* = 0, *access_usec* = 0	*access_sec* = specified number of seconds since 00:00:00, January 1, 1970 *access_usec* = 0 You can update the last-access time even if the last-access time is not enabled as part of its storage characteristics of the smart large object.
Last-modification time	*mod_sec* = 0, *mod_usec* = 0	*mod_sec*, = specified number of seconds since 00:00:00, January 1, 1970 *mod_usec* = specified number of microseconds (only if the platform supports the microsecond component of system time)

Server Only

The **mi_lo_utimes( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_lo_stat_atime( ), mi_lo_stat_mtime_sec( ),** and **mi_lo_stat_mtime_usec( ).**

For more information about the last-access time of a smart large object or about last-modification time of a smart large object, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_lo_validate( )

The **mi_lo_validate( )** function checks whether a given LO handle is valid.

Syntax

```
mi_integer mi_lo_validate(conn, LO_hdl)
   MI_CONNECTION *conn;
   MI_LO_HANDLE *LO_hdl;
```

conn	is one of the following connection values: A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )** A NULL-valued pointer (database server only)
LO_hdl	is a pointer to the LO handle to validate.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

The **mi_lo_validate( )** checks whether the LO handle that LO_*hdl* references is valid. An LO handle might be invalid for any of the following reasons:

- The memory address is invalid or NULL.
- The LO handle contains invalid reference data because:
  - It was never set to a valid value.
  - It was explicitly invalidated with the **mi_lo_invalidate( )** function.

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.

If **mi_lo_validate( )** fails because of an invalid connection, callbacks are invoked.

Server Only

The **mi_lo_validate( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

0	indicates that the LO handle that LO_*hdl* references is valid.
>0	indicates that the LO handle that LO_*hdl* references is invalid.
MI_ERROR	indicates that the connection was invalid.

Related Topics

See also the description of **mi_lo_invalidate( ).**

mi_lo_write( )

The **mi_lo_write( )** function writes a specified number of bytes to an open smart large object.

Syntax

```
mi_integer mi_lo_write(conn, LO_fd, buf, nbytes)
   MI_CONNECTION *conn;
   MI_LO_FD LO_fd;
   const char *buf;
   mi_integer nbytes;
```

conn	is one of the following connection values: A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )** A NULL-valued pointer (database server only)
LO_fd	is an LO file descriptor for the smart large object to which to write. It is obtained by a previous call to **mi_lo_open( ).**
buf	is a pointer to a user-allocated character buffer of at least *nbytes* bytes that contains the data to be written to the smart large object.
nbytes	is the maximum number of bytes to write to the smart large object. This value cannot exceed two gigabytes.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_write( )** function writes *nbytes* of data to the smart large object that the LO*_fd* file descriptor identifies. The write begins at the current LO seek position for LO*_fd*. You can use the **mi_lo_tell( )** function to obtain the current LO seek position.

The function obtains the data from the user-allocated buffer that *buf* references. The *buf* buffer must be less than two gigabytes in size.

If the database server writes less than *nbytes* of data to the smart large object, the **mi_lo_write( )** function returns the number of bytes that it wrote and raises an exception. This condition can occur when the sbspace runs out of space.

To perform a seek operation before a write operation, use the function **mi_lo_writewithseek( ).**

Server Only

The **mi_lo_write( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

>=0	is the actual number of bytes that the function has written from the *buf* character buffer, the open smart large object.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_lo_open( ), mi_lo_read( ), mi_lo_seek( ), mi_lo_tell( ),** and **mi_lo_writewithseek( ).**

mi_lo_writewithseek( )

The **mi_lo_writewithseek( )** function performs a seek operation and then writes a specified number of bytes of data to an open smart large object.

Syntax

```
mi_integer mi_lo_writewithseek(conn, LO_fd, buf, nbytes, offset, whence)
   MI_CONNECTION *conn;
   MI_LO_FD LO_fd;
   const char *buf;
   mi_integer nbytes;
   mi_int8 *offset;
   mi_integer whence;
```

conn	is one of the following connection values: A pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( )** A NULL-valued pointer (database server only)
LO_fd	is an LO file descriptor for the smart large object to which to write.
buf	is a pointer to a user-allocated character buffer of at least *nbytes* bytes that contains the data to be written to the smart large object.
nbytes	is the number of bytes to write to the smart large object. This value cannot exceed two gigabytes.
offset	is a pointer to the eight-byte integer (**mi_int8**) offset from the starting LO seek position.
whence	is an integer value that identifies the starting LO seek position.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lo_writewithseek( )** function writes *nbytes* of data to the smart large object that the LO_*fd* file descriptor identifies. The function obtains the data to write from the user-allocated buffer that *buf* references. The buffer must be less than two gigabytes in size.

The write begins at the LO seek position of LO_*fd* that the *offset* and *whence* arguments indicate, as follows:

- The *whence* argument identifies the position from which to start the seek operation.

  Valid values include the following whence constants.

Whence Constant	Starting LO Seek Position
MI_LO_SEEK_SET	The start of the smart large object
MI_LO_SEEK_CUR	The current LO seek position in the smart large object
MI_LO_SEEK_END	The end of the smart large object

  – The *offset* argument identifies the offset, in bytes, relative to the starting seek position (which the *whence* argument specifies) at which to begin the write operation.

This *offset* value can be negative for all values of *whence*. For more information on how to access eight-bit (INT8) integers, see the *IBM Informix DataBlade API Programmer's Guide*.

Tip: The **mi_lo_writewithseek( )** function is useful in client LIBMI applications because it reduces the number of round trips between the client application and the database server.

If the database server writes less than *nbytes* of data to the smart large object, **mi_lo_writewithseek( )** returns the number of bytes that it wrote and raises an exception. This condition can occur when the sbspace runs out of space.

Server Only

The **mi_lo_writewithseek( )** function does not need a connection descriptor to execute. If your UDR does not need a valid connection for other operations, you can specify a NULL-valued pointer for the *conn* parameter to establish a NULL connection. For information on the advantages of a NULL connection, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

>=0	is the number of bytes that the function wrote from the *buf* character buffer, the open smart large object.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_lo_readwithseek( ), mi_lo_seek( ),** and **mi_lo_write( ).**

mi_lock_memory( )

The mi_lock_memory( ) function locks and waits for a named-memory mutex specified by name and duration.

Syntax

```
mi_integer mi_lock_memory(mem_name, duration)
   mi_string *mem_name;
   MI_MEMORY_DURATION duration;
```

mem_name is the null-terminated name of the named-memory block to lock.

duration is a value that specifies the memory duration of the named-memory block to lock. Valid values for *duration* are:

PER_ROUTINE	For the duration of one iteration of the UDR
PER_COMMAND	For the duration of the execution of the current subquery
PER_STATEMENT *(Deprecated)*	For the duration of the current SQL statement
PER_STMT_EXEC	For the duration of the *execution* of the current SQL statement
PER_STMT_PREP	For the duration of the current prepared SQL statement
PER_TRANSACTION	For the duration of one transaction
PER_SESSION	For the duration of the current client session
PER_SYSTEM	For the duration of the database server execution

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Warning: This advanced function can adversely affect your UDR if you use the function incorrectly. Use it only when no regular DataBlade API function can perform the task you need done.

Usage

The **mi_lock_memory( )** function requests a lock on the named-memory block based on its memory duration of *duration* and its name, which *mem_name* references. The function waits until this lock has been obtained before it returns control to its calling function.

Important: After you obtain a lock on a named-memory block, release it as soon as possible. You must explicitly release a named-memory lock with the **mi_unlock_memory( )** function.

Return Values

MI_OK	indicates that the function successfully locked the specified named-memory block.
MI_NO_SUCH_NAME	indicates that the requested named-memory block does not exist for the specified duration.
MI_POTENTIAL_DEADLOCK	indicates that acquisition of a lock on the specified named-memory block failed because it could result in deadlock.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_named_alloc( ), mi_named_zalloc( ), mi_try_lock_memory( ),** and **mi_unlock_memory( ).**

mi_lvarchar_to_string( )

The **mi_lvarchar_to_string( )** function returns the data in a varying-length structure as a null-terminated string.

Syntax

```
mi_string *mi_lvarchar_to_string(lv_ptr)
   mi_lvarchar *lv_ptr;
```

lv_ptr	is a pointer to the varying-length structure whose data is to be converted to a null-terminated string.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_lvarchar_to_string( )** function returns a null-terminated string for the data in the varying-length structure that *lv_ptr* references. This function allocates memory for the string it returns. Because **mi_lvarchar_to_string( )** uses **mi_alloc( ),** this string memory has the current memory duration. You must use the **mi_free( )** function to free this string memory when it is no longer needed.

Server Only

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

End of Server Only

Return Values

An **mi_string** pointer	is a pointer to the newly allocated buffer that contains the string equivalent of the varying-length data.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_alloc( ), mi_free( ), mi_new_var( ), mi_string_to_lvarchar( ), mi_var_copy( ), mi_var_free( ),** and **mi_var_to_buffer( ).**

mi_module_lock( )

The **mi_module_lock( )** function prevents the shared-object file of the current UDR from being unloaded.

Syntax

```
mi_integer mi_module_lock(lock_flag)
   mi_integer lock_flag;
```

lock_flag is the value to set the module-lock flag for the shared-object file of the current UDR. This flag can have the following values:

MI_TRUE	Sets the module-lock flag to prevent the routine manager from unloading the shared-object file.
MI_FALSE	Unsets the module-lock flag to indicate that the routine manager can unload the shared-object file when necessary. This action does not force an unload of the shared-object file.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Warning: This advanced function can adversely affect your UDR if you use the function incorrectly. Use it only when no regular DataBlade API function can perform the task you need done.

Usage

The **mi_module_lock( )** function sets the module-lock flag to the value that *lock_flag* specifies. The module-lock flag indicates whether to lock the shared-object file in the memory of the database server. When the module-lock flag is MI_TRUE, the routine manager *does* not allow the shared-object file to be unloaded for any reason. This function is useful for preventing arbitrary unloading of shared-object files, which necessitates reinitialization the next time any UDR on the shared-object file is used.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the description of **mi_udr_lock( ).**

mi_money_to_binary( )

The **mi_money_to_binary( )** function converts a text (string) representation of a monetary value to its binary (internal) MONEY representation.

Syntax

```
mi_money *mi_money_to_binary(money_string)
   mi_lvarchar *money_string;
```

money_string	is a pointer to the monetary string to convert to its internal MONEY format.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_money_to_binary( )** function converts the monetary string that *money_string* references to its internal MONEY value. An internal MONEY value is the format that the database server uses to store a value in a MONEY column of the database. This format represents a fixed-point decimal number.

Global Language Support

The **mi_money_to_binary( )** function accepts the monetary string in the monetary format of the current processing locale. It also performs any code-set conversion necessary between the current processing locale and the target locale.

End of Global Language Support

Important: The **mi_money_to_binary( )** function is supported only for compatibility with earlier versions of existing DataBlade API modules. The DataBlade API will eventually discontinue support for this function. Use the **mi_string_to_money( )** function in any new DataBlade API modules.

Return Values

An **mi_money** pointer	is a pointer to the internal MONEY representation that **mi_money_to_binary( )** has created.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_binary_to_money( )** and **mi_string_to_money( ).**

mi_money_to_string( )

The **mi_money_to_string( )** function creates a text (string) representation of a monetary value from the binary (internal) MONEY representation.

Syntax

```
mi_string *mi_money_to_string(money_data)
   mi_money *money_data;
```

money_data	is a pointer to the internal MONEY representation of the monetary value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_money_to_string( )** function converts the internal MONEY value that *money_data* contains into a monetary string. An internal MONEY value is the format that the database server uses to store a value in a MONEY column of the database.

Important: The **mi_money_to_string( )** function replaces the **mi_binary_to_money( )** function for internal MONEY-to-string conversion in DataBlade API modules.

Global Language Support

The **mi_money_to_string( )** function formats the monetary string in the monetary format of the current processing locale. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Return Values

An **mi_string** pointer	is a pointer to the monetary string equivalent of *money_data*.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_date_to_string( ), mi_datetime_to_string( ), mi_decimal_to_string( ), mi_interval_to_string( ),** and **mi_string_to_decimal( ).**

For more information on how to convert internal MONEY format to monetary strings, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_named_alloc( )

The mi_named_alloc( ) function allocates a named block of memory of the specified size in the specified memory duration.

Syntax

```
mi_integer mi_named_alloc(size, mem_name, duration, mem_ptr)
   mi_integer size;
   mi_string *mem_name;
   MI_MEMORY_DURATION duration;
   void **mem_ptr;
```

size	is the number of bytes to allocate to the named-memory block.
mem_name	is the null-terminated name to assign the named-memory block.
duration	is a value that specifies the memory duration of the named-memory block to allocate. Valid values for *duration* follow:

PER_ROUTINE	For the duration of one iteration of the UDR
PER_COMMAND	For the duration of the execution of the current subquery
PER_STATEMENT *(Deprecated)*	For the duration of the current SQL statement
PER_STMT_EXEC	For the duration of the *execution* of the current SQL statement
PER_STMT_PREP	For the duration of the current prepared SQL statement
PER_TRANSACTION	For the duration of one transaction
PER_SESSION	For the duration of the current client session
PER_SYSTEM	For the duration of the database server execution

mem_ptr	is the pointer to the allocated named-memory block.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
No	Yes

Warning: This advanced function can adversely affect your UDR if you use the function incorrectly. Use it only when no regular DataBlade API function can perform the task you need done.

Usage

The **mi_named_alloc( )** function allocates a named-memory block of *size* bytes of memory, with a memory duration of *duration*, and assigns the block the name that *mem_name* references. The function saves the pointer to the allocated named-memory block in the *mem_ptr* argument. The **mi_named_alloc( )** function is a constructor function for named memory.

Important: The **mi_named_alloc( )** function only allocates a block of named memory. It does not request a lock on this memory.

A DataBlade API module can use **mi_named_free( )** to free named memory when that memory is no longer needed.

Return Values

MI_OK	indicates that the function successfully allocated the specified named-memory block and a pointer to this block is stored in *mem_ptr*.
MI_NAME_ALREADY_EXISTS	indicates that a named-memory block with the *mem_name* name already exists for the specified duration.
MI_ERROR	indicates that the function was *not* successful. The *mem_ptr* pointer is set to a NULL-valued pointer.

Related Topics

See also the descriptions of **mi_alloc( ), mi_dalloc( ), mi_free( ), mi_lock_memory( ), mi_named_free( ), mi_named_get( ), mi_named_zalloc( ), mi_try_lock_memory( ), mi_unlock_memory( ),** and **mi_zalloc( ).**

mi_named_free( )

The mi_named_free( ) function frees a block of named memory.

Syntax

```
void mi_named_free(mem_name, duration)
   mi_string *mem_name;
   MI_MEMORY_DURATION duration;
```

mem_name	is the null-terminated name of an existing named-memory block.
duration	is a value that specifies the memory duration of the named-memory block to free. Valid values for *duration* are:

PER_ROUTINE	For the duration of one iteration of the UDR
PER_COMMAND	For the duration of the execution of the current subquery
PER_STATEMENT *(Deprecated)*	For the duration of the current SQL statement
PER_STMT_EXEC	For the duration of the *execution* of the current SQL statement
PER_STMT_PREP	For the duration of the current prepared SQL statement
PER_TRANSACTION	For the duration of one transaction
PER_SESSION	For the duration of the current client session
PER_SYSTEM	For the duration of the database server execution

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Warning: This advanced function can adversely affect your UDR if you use the function incorrectly. Use it only when no regular DataBlade API function can perform the task you need done.

Usage

The **mi_named_free( )** function frees a named-memory block based on its memory duration of *duration* and its name, which *mem_name* references. 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. If you do not explicitly free named memory, the database server frees it when its memory duration expires.

If *mem_name* references a named-memory block that was already freed or does not reference a named-memory block, the **mi_named_free( )** function does not return a value. The function returns silently in all cases, even when it cannot find *mem_name*.

The **mi_named_free( )** function frees named memory previously allocated with **mi_named_alloc( )** or **mi_named_zalloc( ).** It does *not* free memory allocated with **malloc( )**, **mi_alloc( ), mi_dalloc( ),** or **mi_zalloc( ).**

Warning: Do not attempt to free a named-memory block if it is currently locked. Always unlock a named-memory block with **mi_unlock_memory( )** before you attempt to free the memory. Failure to do so can severely impact the operation of the database server.

Return Values

None.

Related Topics

See also the descriptions of **mi_alloc( ), mi_dalloc( ), mi_free( ), mi_lock_memory( ), mi_named_alloc( ), mi_named_get( ), mi_named_zalloc( ),** and **mi_zalloc( ).**

mi_named_get( )

The mi_named_get( ) function retrieves the address of a named-memory block.

Syntax

```
mi_integer mi_named_get (mem_name, duration, mem_ptr)
   mi_string *mem_name;
   MI_MEMORY_DURATION duration;
   void **mem_ptr;
```

mem_name	is the null-terminated name of the named-memory block whose address the function retrieves.
duration	is a value that specifies the memory duration of the named-memory block to retrieve. Valid values for *duration* are:

PER_ROUTINE	For the duration of one iteration of the UDR
PER_COMMAND	For the duration of the execution of the current subquery
PER_STATEMENT *(Deprecated)*	For the duration of the current SQL statement
PER_STMT_EXEC	For the duration of the *execution* of the current SQL statement
PER_STMT_PREP	For the duration of the current prepared SQL statement
PER_TRANSACTION	For the duration of one transaction
PER_SESSION	For the duration of the current client session
PER_SYSTEM	For the duration of the database server execution

mem_ptr	is the pointer to the retrieved named-memory block.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Warning: This advanced function can adversely affect your UDR if you use the function incorrectly. Use it only when no regular DataBlade API function can perform the task you need done.

Usage

The **mi_named_get( )** function obtains the address of a named-memory block based on its memory duration of *duration* and its name, which *mem_name* references. This function is useful in a UDR that needs access to a particular block of named memory. The UDR can specify the name and memory duration of the desired named-memory block to **mi_named_get( )** and receive the address of this memory.

Important: The **mi_named_get( )** function only retrieves the address of a block of named memory. It does not request a lock on this memory.

Return Values

MI_OK	indicates that the function successfully retrieved the specified named-memory block and a pointer to this block is stored in *mem_ptr*.
MI_NO_SUCH_NAME	indicates that the requested named-memory block does not exist.
MI_ERROR	indicates that the function was *not* successful. The *mem_ptr* pointer is set to a NULL-valued pointer.

Related Topics

See also the descriptions of **mi_alloc( ), mi_dalloc( ), mi_free( ), mi_lock_memory( ), mi_named_alloc( ), mi_named_free( ), mi_named_zalloc( ),** and **mi_zalloc( ).**

mi_named_zalloc( )

The mi_named_zalloc( ) function allocates and initializes a named-memory block in the specified memory duration.

Syntax

```
mi_integer *mi_named_zalloc(size, mem_name, duration, mem_ptr)
   mi_integer size;
   mi_string mem_name;
   MI_MEMORY_DURATION duration;
   char **mem_ptr;
```

size	is the number of bytes to allocate to the named-memory block.
mem_name	is the null-terminated name to assign the named-memory block.
duration	is a value that specifies the memory duration of the named-memory block to allocate. Valid values for *duration* are:

PER_ROUTINE	For the duration of one iteration of the UDR
PER_COMMAND	For the duration of the execution of the current subquery
PER_STATEMENT *(Deprecated)*	For the duration of the current SQL statement
PER_STMT_EXEC	For the duration of the *execution* of the current SQL statement
PER_STMT_PREP	For the duration of the current prepared SQL statement
PER_TRANSACTION	For the duration of one transaction
PER_SESSION	For the duration of the current client session
PER_SYSTEM	For the duration of the database server execution

mem_ptr	is the pointer to the zero-filled allocated named-memory block.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Warning: This advanced function can adversely affect your UDR if you use the function incorrectly. Use it only when no regular DataBlade API function can perform the task you need done.

Usage

The **mi_named_zalloc( )** function allocates a named-memory block of *size* bytes of memory, with a memory duration of *duration*, and assigns it the name that *mem_name* references. It then initializes this named-memory block with zeros. The function saves the pointer to the allocated named-memory block in the *mem_ptr* argument. The **mi_named_zalloc( )** function is a constructor function for named memory.

Important: The **mi_named_zalloc( )** function only allocates a block of named memory. It does not request a lock on this memory.

A DataBlade API module can use **mi_named_free( )** to free named memory when that memory is no longer needed.

Return Values

MI_OK	indicates that the function was successful.
MI_NAME_ALREADY_EXISTS	indicates that a named-memory block with the *mem_name* name already exist for the specified duration.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_alloc( ), mi_dalloc( ), mi_free( ), mi_lock_memory( ), mi_named_alloc( ), mi_named_free( ), mi_named_get( ),** and **mi_zalloc( ).**

mi_new_var( )

The **mi_new_var( )** function creates a new varying-length structure for text data.

Syntax

```
mi_lvarchar *mi_new_var(data_len)
   mi_integer data_len;
```

data_len	is the length of the data to store in the new varying-length structure.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_new_var( )** function is a constructor function for a varying-length structure. It allocates a varying-length structure with *data_len* bytes of data in its data portion. The function allocates memory for the varying-length structure that it returns. Therefore, you must use the **mi_var_free( )** function to free this structure when it is no longer needed.

Server Only

The **mi_new_var( )** function allocates a new varying-length structure with the current memory duration.

End of Server Only

Important: Do not use the DataBlade API memory-management functions such as **mi_alloc( )** to allocate a varying-length structure.

Return Values

An **mi_lvarchar** pointer	is a pointer to the new variable-length structure.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_vardata( ), mi_get_vardata_align( ), mi_get_varlen( ), mi_lvarchar_to_string( ), mi_set_vardata( ), mi_set_vardata_align( ), mi_set_varlen( ), mi_string_to_lvarchar( ), mi_var_copy( )**, and **mi_var_free( ).**

mi_next_row( )

The **mi_next_row( )** function retrieves the next row from the cursor of a query.

Syntax

```
MI_ROW *mi_next_row(conn, error)
   MI_CONNECTION *conn;
   mi_integer *error;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
error	is a pointer to the return code that **mi_next_row( )** generates.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_next_row( )** function returns a pointer to a row structure that contains the row just fetched (the *current row*) from the cursor associated with the current statement. The current statement is associated with the connection that *conn* references. Only one cursor per connection is current, and within this cursor, only one row at a time is current.

The contents of this row structure are valid until the next call to **mi_next_row( )**. The row structure itself is valid until one of the following conditions occurs:

- The query finishes.
- The **mi_close( )** function is called on the connection.

Important: Do not use the **mi_row_free( )** function to free the row structure that **mi_next_row( )** uses. Use **mi_row_free( )** only for row structures that you allocate with **mi_row_create( ).**

Use the **mi_next_row( )** function when **mi_get_result( )** returns MI_ROWS to indicate that the cursor is ready to be accessed. Upon successful completion, the **mi_next_row( )** function advances the cursor pointer to the next row to be fetched from the cursor. After **mi_next_row( )** successfully fetches a row, you can extract column values with the **mi_value( )** or **mi_value_by_name( )** function.

The **mi_next_row( )** *function* is typically executed in a loop that terminates when no more rows remain to be fetched from the cursor. To indicate the "no more rows" condition, the function takes the following steps:

- Returns the NULL-valued pointer
- Sets the *error* argument to MI_NO_MORE_RESULTS

An *error* value of MI_NO_MORE_RESULTS indicates that you are at the end of the cursor or that the cursor is empty.

Return Values

An **MI_ROW** pointer	is a pointer to the current row.

NULL	indicates that the function was not successful, no more rows remain to be retrieved from the cursor, or the cursor is empty.

Upon failure, **mi_next_row( )** returns NULL and sets *error* to MI_ERROR.

Related Topics

See also the descriptions of **mi_close( ), mi_get_result( ), mi_get_row_desc_without_row( ), mi_value( ),** and **mi_value_by_name( ).**

mi_open( )

The **mi_open( )** function establishes a connection to a database server.

Syntax

```
MI_CONNECTION *mi_open(db_name, user_name, user_passwd)
   const char *db_name;
   const char *user_name;
   const char *user_passwd;
```

db_name	is the name of the database to open.
user_name	is the name of the user account on the server computer.
user_passwd	is the account password.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_open( )** function obtains a connection descriptor for a connection. This function is a constructor function for a connection descriptor. The DataBlade API module can pass the new connection descriptor to subsequent routines that require a connection. The **mi_open( )** function also initializes the DataBlade API when this function is the first DataBlade API function in a client LIBMI application or a C UDR.

Server Only

For a C UDR, the **mi_open( )** function establishes a UDR connection, which gives the UDR access to the session in which the SQL statement that initiated the UDR executed. To open a UDR connection, pass *all* three arguments to **mi_open( )** as the NULL-valued pointers. The **mi_open( )** function allocates a new connection descriptor in the PER_STMT_EXEC memory duration.

Important: In a C UDR, do not specify the name of a database or user account to the **mi_open( )** function. If you do not provide the three arguments as NULL-valued pointers, **mi_open( )** fails but does not generate a message.

End of Server Only

Client Only

For a client LIBMI application, the **mi_open( )** function establishes a client connection to the default database server and opens the *db_name* database. You can also specify the user account with the *user_name* and *user_passwd* arguments. The establishment of a client connection begins a session. The **mi_open( )** function allocates a new connection descriptor that is valid until the end of the session.

If a client LIBMI application sets default database parameters with **mi_set_default_database_info( ), mi_open( )** uses these defaults when it receives NULL-valued pointers as arguments. Otherwise, **mi_open( )** uses the following default values.

mi_open( ) Argument	Default Value When Argument Is NULL

Database name	None
User name	The name of the user login account that is running the executable file
Password	The password of the user account that is running the executable file

If the client LIBMI application uses a shared-memory communication, it can establish only one connection per application.

The **mi_open( )** function uses the default connection parameters to establish a client connection. To specify a nondefault system name for a client session, call the **mi_sysname( )** function before the call to **mi_open( )**.

Important: Do not specify the system name in the format "dbname@servername" within the "dbname" argument of the **mi_open( )** function. Instead, call the **mi_sysname( )** function before **mi_open( )**.

End of Client Only

Return Values

An **MI_CONNECTION** pointer	is a pointer to the connection descriptor for the newly established connection.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_close( ), mi_get_session_connection( ), mi_server_connect( ), mi_set_default_database_info( ),** and **mi_sysname( ).**

mi_open_prepared_statement( )

The **mi_open_prepared_statement( )** function sends a prepared statement that is a query to the database server for execution and opens a cursor for the retrieved rows of the query.

Syntax

```
mi_integer mi_open_prepared_statement(stmt_desc, control,
params_are_binary, n_params, values, lengths, nulls,
types, cursor_name,retlen, rettypes)
   MI_STATEMENT *stmt_desc;
   mi_integer control;
   mi_integer params_are_binary;
   mi_integer n_params;
   MI_DATUM values[];
   mi_integer lengths[];
   mi_integer nulls[];
   mi_string *types[];
   mi_string *cursor_name;
   mi_integer retlen;
   mi_string *rettypes[];
```

stmt_desc
: is a pointer to a statement descriptor for the prepared statement to open. The **mi_prepare( )** function generates this statement descriptor.

control
: is a bit-mask flag that controls the following characteristics:

  Whether the returned rows are returned in their binary (internal) or text (external) representation

  The type of cursor to create and open

params_are_binary
: is set to one of the following values:

  1 (MI_TRUE) The input-parameter values (in the *values* array) are passed in their internal (binary) representation.

  0 (MI_FALSE)
  The input-parameter values (in the *values* array) are passed in their external (text) representation.

n_params
: is the number of entries in the *nulls, lengths*, and *values* arrays.

values
: is an array of **MI_DATUM** structures that contain the values of the input parameters in the prepared statement.

lengths
: is an array of the lengths (in bytes) of the input-parameter values.

nulls
: is an array that indicates whether each input parameter contains an SQL NULL value. Each array element is set to one of the following values:

  1 (MI_TRUE) The value of the associated input parameter *is* an SQL NULL value.

0 (MI_FALSE)	The value of the associated input parameter is *not* an SQL NULL value.

types	is an array of pointers to the data type names for the input parameters. This array can be a NULL-valued pointer.
cursor_name	is a name of the cursor that holds the fetched rows. This name must be unique.
retlen	is the length of the *rettypes* array. Currently valid values are:
	>0 Indicates the number of columns that the query returns
	0 Indicates that no result values exist
rettypes	is an array of pointers to the data type names to which the result values are cast. This array can be a NULL-valued pointer if result values do not need to be cast.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_open_prepared_statement( )** performs the following tasks:

- Binds any input-parameter values to the input-parameter placeholders in the prepared SQL statement that the *stmt_desc* statement descriptor references

  For any input parameters specified in the statement text of the SQL statement, you must initialize the *values*, *lengths*, and *nulls* arrays. 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. You can provide the input-parameter values in either of the representations that correspond with the *params_are_binary* flag.

  For information on how to bind input-parameter values, see the *IBM Informix DataBlade API Programmer's Guide*.
- Sends the prepared statement to the database server for execution
- Opens an explicit cursor whose characteristics are specified by a bit mask in the *control* argument

Cursor Type	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 and MI_SEND_SCROLL
Update sequential cursor with hold	MI_SEND_HOLD
Read-only sequential cursor with hold	MI_SEND_READ and MI_SEND_HOLD
Update scroll cursor with hold	MI_SEND_SCROLL and MI_SEND_HOLD
Read-only scroll cursor with hold	MI_SEND_READ and MI_SEND_SCROLL and MI_SEND_HOLD

The cursor is stored as part of the statement descriptor. Only one cursor per statement descriptor is current.

The *control* argument also determines the representation of any returned values.

Data Representation	Control-Flag Value
External (text) representation	*None (default)*
Internal (binary) representation	MI_BINARY

The **mi_open_prepared_statement( )** function allocates type descriptors for each of the data types of the input parameters in the *types* array. 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. 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.

You can set the data types of the selected columns by setting a pointer to type name for each returned column in the *rettypes* array. If the pointer is NULL, the type is not modified. It will either be the return type of the column or the type set by a previous **mi_open_prepared_statement( )** call. You cannot set the return types of subcolumns of columns of a complex type.

You can use the *cursor_name* argument to specify the name of the cursor that holds the fetched rows. This name must be unique within the client session.

Server Only

When you specify a non-NULL value as the *cursor_name* argument for **mi_open_prepared_statement( )**, make sure that you specify a NULL-valued pointer as the *name* argument for the **mi_prepare( )** function. If you specify a non-NULL cursor name for **mi_prepare( ),** use a NULL-valued pointer as the *cursor_name* value for **mi_open_prepared_statement( )**. 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.

End of Server Only

To use an internally-generated unique name for the cursor, specify a NULL-valued pointer.

Client Only

To use an internally generated unique name for the cursor, specify a NULL-valued pointer for the *cursor_name* argument of **mi_open_prepared_statement( )**.

End of Client Only

Once opened, you can set up rows in the cursor for fetching with the **mi_fetch_statement( )** function.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

A successful return indicates only that the connection is valid and a cursor was successfully opened. It does *not* indicate the success of the SQL statement. Use **mi_get_result( )** to determine the success of the SQL statement.

Related Topics

See also the descriptions of **mi_close_statement( ), mi_exec_prepared_statement( ), mi_fetch_statement( ), mi_get_cursor_table( ),** and **mi_prepare( ).**

mi_parameter_count( )

The **mi_parameter_count( )** function returns the number of input parameters in a prepared statement.

Syntax

```
mi_integer mi_parameter_count(stmt_desc)
   MI_STATEMENT *stmt_desc;
```

stmt_desc	is a pointer to the statement descriptor for a prepared statement.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_parameter_count( )** function obtains the number of input parameters in the statement descriptor that *stmt_desc* references. A statement descriptor describes a prepared statement. You can use the **mi_parameter_count( )** function with the DataBlade API functions that obtain information about each input parameter of the prepared statement (such as **mi_parameter_type_id( )** and **mi_parameter_nullable( )**).

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 parameter information for other SQL statements, **mi_parameter_count( )** raises an exception.

Return Values

>=0	is the number of input parameters contained in the text of the prepared statement.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_close_statement( ), mi_exec_prepared_statement( ), mi_fetch_statement( ), mi_parameter_nullable( ), mi_parameter_precision( ), mi_parameter_scale( ), mi_parameter_type_id( ), mi_parameter_type_name( ),** and **mi_prepare( ).**

mi_parameter_nullable( )

The **mi_parameter_nullable( )** function indicates whether the column associated with a specified input parameter in a prepared statement can contain an SQL NULL value.

Syntax

```
mi_integer mi_parameter_nullable(stmt_desc, param_id)
   MI_STATEMENT *stmt_desc;
   mi_integer param_id;
```

stmt_desc	is a pointer to the statement descriptor for the prepared statement that contains the input parameter.
param_id	is the integer parameter identifier of the input parameter, which specifies the position of the input parameter in the prepared statement. Input-parameter numbering follows C programming conventions: the first parameter in the statement is at position zero (0).

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_parameter_nullable( )** function checks whether the column that is associated with the input parameter at position *param_id* is nullable. A column is nullable when it was declared with the NOT NULL column-level constraint to specify that it is not able to hold SQL NULL values. For more information on column-level constraints, see the description of CREATE TABLE in the *IBM Informix Guide to SQL: Syntax*.

The **mi_parameter_nullable( )** function is an accessor function for a statement descriptor, which describes a prepared statement. The statement descriptor stores information about whether input-parameter columns are nullable in the zero-based parameter-nullable array. To obtain information about the *n*th input parameter, use a *param_id* value of *n*-1.

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 parameter information for other SQL statements, **mi_parameter_nullable( )** raises an exception.

Return Values

0	indicates that the column associated with the specified input parameter is defined with the NOT NULL constraint.
1	indicates that the column associated with the specified input parameter is defined to accept NULL values; that is, it has *not* been defined with the NOT NULL constraint.

MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_close_statement( ), mi_exec_prepared_statement( ), mi_fetch_statement( ), mi_get_statement_row_desc( ), mi_parameter_count( ), mi_parameter_precision( ), mi_parameter_scale( ), mi_parameter_type_id( ), mi_parameter_type_name( ), mi_prepare( ),** and **mi_statement_command_name( ).**

mi_parameter_precision( )

The **mi_parameter_precision( )** function obtains the precision of the column associated with the specified input parameter in a prepared statement.

Syntax

```
mi_integer mi_parameter_precision(stmt_desc, param_id)
   MI_STATEMENT *stmt_desc;
   mi_integer param_id;
```

stmt_desc	is a pointer to the statement descriptor for the prepared statement that contains the input parameter.
param_id	indicates the parameter identifier of the column, which specifies the position of the input parameter in the specified statement descriptor. Input-parameter numbering follows C programming conventions: the first parameter in the statement is at position zero (0).

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_parameter_precision( )** function obtains the precision of the column that is associated with the *param_id* input parameter from the statement descriptor that *stmt_desc* references. This function is an accessor function for a statement descriptor, which describes a prepared statement. The statement descriptor stores information about the precisions of input-parameter columns in the zero-based parameter-precision array. To obtain information about the *n*th input parameter, use a *param_id* value of *n*-1.

The precision is an attribute of the column data type that represents the total number of digits the column associated with an input parameter can hold, as follows.

Data Type	Meaning
DECIMAL, MONEY	Number of significant digits in the fixed-point or floating-point (DECIMAL) column
DATETIME, INTERVAL	Number of digits that are stored in the date and/or time column with the specified qualifier
Character, Varying-character	Maximum number of characters in the column

If you call **mi_parameter_precision( )** on an input parameter whose column is some other data type, the function returns zero (0).

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 parameter information for other SQL statements, **mi_parameter_precision( )** raises an exception.

Return Values

0	indicates that no precision was set for the column of the specified input parameter.
>=0	is the precision of the column that is associated with the specified input parameter.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_close_statement( ), mi_exec_prepared_statement( ), mi_fetch_statement( ), mi_get_statement_row_desc( ), mi_parameter_count( ), mi_parameter_nullable( ), mi_parameter_scale( ), mi_parameter_type_id( ), mi_parameter_type_name( ), mi_prepare( ),** and **mi_statement_command_name( ).**

For more information about input parameters or about the precision of a fixed-point data type, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_parameter_scale( )

The **mi_parameter_scale( )** function returns the scale of the column that is associated with a specified input parameter in a prepared statement.

Syntax

```
mi_integer mi_parameter_scale(stmt_desc, param_id)
   MI_STATEMENT *stmt_desc;
   mi_integer param_id;
```

stmt_desc	is a pointer to the statement descriptor for the prepared statement that contains the input parameter.
param_id	indicates the parameter identifier of the column, which specifies the position of the input parameter in the specified statement descriptor. Input-parameter numbering follows C programming conventions: the first parameter in the statement is at position zero (0).

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_parameter_scale( )** function obtains the column scale of the column that is associated with *param_id* from the statement descriptor that *stmt_desc* references. This function is an accessor function for a statement descriptor, which describes a prepared statement. The statement descriptor stores information about the scales of input-parameter columns in the zero-based parameter-scale array. To obtain information about the *n*th input parameter, use a *param_id* value of *n*-1.

The scale is an attribute of the data type. The meaning of the scale depends on the associated data type, as the following table shows.

Data Type	Meaning of Scale
DECIMAL (fixed-point), MONEY	The number of digits to the right of the decimal point
DECIMAL (floating-point)	The value 255
DATETIME, INTERVAL	The encoded integer value for the end qualifier of the data type; *end_qual* in the qualifier: *start_qual* TO *end_qual*

If you call **mi_parameter_scale( )** on some other data type, the function returns zero (0).

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 parameter information for other SQL statements, **mi_parameter_scale( )** raises an exception.

Return Values

0	indicates that the data type of the column associated with the specified input parameter is something other than DECIMAL or MONEY.
>0	is the scale of the DECIMAL or MONEY column that is associated with the specified input parameter.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_close_statement( ), mi_exec_prepared_statement( ), mi_fetch_statement( ), mi_get_statement_row_desc( ), mi_parameter_count( ), mi_parameter_nullable( ), mi_parameter_precision( ), mi_parameter_type_id( ), mi_parameter_type_name( ), mi_prepare( ),** and **mi_statement_command_name( ).**

For more information about input parameters or about the scale of a fixed-point data type, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_parameter_type_id( )

The **mi_parameter_type_id( )** function returns the type identifier of the column that is associated with the specified input parameter in a prepared statement.

Syntax

```
MI_TYPEID *mi_parameter_type_id(stmt_desc, param_id)
   MI_STATEMENT *stmt_desc;
   mi_integer param_id;
```

stmt_desc	is a pointer to the statement descriptor for the prepared statement that contains the input parameter.
param_id	indicates the parameter identifier of the column, which specifies the position of the input parameter in the specified statement descriptor. Input-parameter numbering follows C programming conventions: the first parameter in the statement is at position zero (0).

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_parameter_type_id( )** function obtains the type identifier for the column that is associated with *param_id* column from the statement descriptor that *stmt_desc* references. This function is an accessor function for a statement descriptor, which describes a prepared statement. The statement descriptor stores information about the type identifiers of input-parameter columns in the zero-based parameter-type id array. To obtain information about the *n*th input parameter, use a *param_id* value of *n*-1.

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 parameter information for other SQL statements, **mi_parameter_type_id( )** raises an exception.

Return Values

An **MI_TYPEID** pointer	is a pointer to the type identifier of the column that is associated with the specified input parameter.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_close_statement( ), mi_exec_prepared_statement( ), mi_fetch_statement( ), mi_get_statement_row_desc( ), mi_parameter_count( ), mi_parameter_nullable( ), mi_parameter_precision( ), mi_parameter_scale( ), mi_parameter_type_name( ), mi_prepare( ),** and **mi_statement_command_name( ).**

For more information about input parameters or about type identifiers, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_parameter_type_name( )

The **mi_parameter_type_name( )** function returns the data type name of the column that is associated with a specified input parameter in a prepared statement.

Syntax

```
mi_string *mi_parameter_type_name(stmt_desc, param_id)
   MI_STATEMENT *stmt_desc;
   mi_integer param_id;
```

stmt_desc	is a pointer to the statement descriptor for the prepared statement that contains the input parameter.
param_id	indicates the parameter identifier of the column, which specifies the position of the input parameter in the specified statement descriptor. Input-parameter numbering follows C programming conventions: the first parameter in the statement is at position zero (0).

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_parameter_type_name( )** function obtains the data type name of the column that is associated with *param_id* from the statement descriptor that *stmt_desc* references. This function is an accessor function for a statement descriptor, which describes a prepared statement. The statement descriptor stores information about the type names of input-parameter columns in the zero-based parameter-type name array. To obtain information about the *n*th input parameter, use a *param_id* value of *n*-1.

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 parameter information for other SQL statements, **mi_parameter_type_name( )** raises an exception.

Return Values

An **mi_string** pointer	is the name of the data type of the column that is associated with the specified input parameter.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_close_statement( ), mi_exec_prepared_statement( ), mi_fetch_statement( ), mi_get_statement_row_desc( ), mi_parameter_count( ), mi_parameter_nullable( ), mi_parameter_precision( ), mi_parameter_scale( ), mi_parameter_type_id( ), mi_prepare( ),** and **mi_statement_command_name( ).**

mi_prepare( )

The **mi_prepare( )** function sends an SQL statement to the database server to be prepared and returns a statement descriptor for the prepared statement.

Syntax

```
MI_STATEMENT *mi_prepare(conn, stmt_strng, name)
   MI_CONNECTION *conn;
   mi_string *stmt_strng;
   mi_string *name;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
stmt_strng	is a pointer to the statement string, which contains the text of the SQL statement to prepare.
name	is an optional name to assign to the cursor associated with the prepared statement on the server computer or to the prepared statement on the client computer.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_prepare( )** function sends the SQL statement in the *stmt_strng* statement string to the database server. The database server parses and optimizes the statement string and builds the necessary internal structures for a prepared statement. This function is the constructor for the statement descriptor (MI_STATEMENT).

The *stmt_strng* statement string can contain input parameters to indicate where column or expression values will be provided at runtime. Specify input parameters within the statement string with the question-mark symbol (?).

Server Only

The **mi_prepare( )** function does not allocate a new statement descriptor from memory-duration pools. For information about the scope of a statement descriptor, see the *IBM Informix DataBlade API Programmer's Guide*.

If your prepared statement will fetch rows, you can use the *name* argument to specify the name of the cursor that holds the fetched rows. This name must be unique within the client session. When you specify a non-NULL cursor name to **mi_prepare( )**, make sure that you specify a NULL-valued pointer as the *cursor_name* argument of the **mi_open_prepared_statement( )** function. If you want to postpone specification of the cursor name to the **mi_open_prepared_statement( )** call, use a NULL-valued pointer as the *name* value 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. For more information, see the *IBM Informix DataBlade API Programmer's Guide*.

Client Only

The *name* argument specifies the statement name for the prepared statement. This name must be unique within the client session. To use an internally generated unique name for the statement, specify a NULL-valued pointer for the *name* argument of **mi_prepare( )**. The *cursor_name* argument of **mi_open_prepared_statement( )** specifies the cursor name for the prepared statement.

End of Client Only

After you have a statement descriptor, you can obtain the following information about the prepared statement.

Prepared-Statement Information	DataBlade API Functions
Input parameters	Input-parameter accessor functions (which begin with "**mi_parameter_**")
Columns (from the row descriptor)	Column accessor functions (which begin with "**mi_column_**")

For more information about information in a statement descriptor, see the *IBM Informix DataBlade API Programmer's Guide*.

Return Values

An **MI_STATEMENT** pointer	is a pointer to a statement descriptor for the SQL statement in *stmt_strng*.
NULL	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_column_count( ), mi_column_id( ), mi_column_name( ), mi_column_nullable( ), mi_column_precision( ), mi_column_scale( ), mi_column_type_id( ), mi_column_typedesc( ), mi_exec_prepared_statement( ), mi_fetch_statement( ), mi_get_cursor_table( ), mi_get_statement_row_desc( ), mi_open_prepared_statement( ), mi_parameter_count( ), mi_parameter_nullable( ), mi_parameter_precision( ), mi_parameter_scale( ), mi_parameter_type_id( ), mi_parameter_type_name( ),** and **mi_statement_command_name( ).**

mi_process_exec( )

The **mi_process_exec( )** function forks and executes a new process, returning before this new process completes.

Syntax

```
mi_integer mi_process_exec(argv)
   char *argv[];
```

argv	is a pointer to the command array, which provides the commands to execute in the newly forked process.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Warning: This advanced function can adversely affect your UDR if you use the function incorrectly. Use it only when no regular DataBlade API function can perform the task you need done.

Usage

The **mi_process_exec( )** function forks a process and executes the commands in the *argv* array. The rules for the contents of the *argv* array are as follows:

- The first element must be an executable program or script.

  You must provide the *full* pathname of this program or script. A single environment variable is allowed as a prefix using the same rules as the **mi_file_open( )** function.

  To include environment variables in the command pathname, use the following syntax:

  ```
  $ENV_VAR
  ```

  To use the **PATH** environment variable, you must execute the command through a shell (such as **/bin/sh**). However, make sure that the command is also executable:

  ```
  /bin/sh -c cmd
  ```

  If the command is a script, it must be executable and start with a shell identifier (such as "**#!/bin/sh**").
- The last element must be the keyword NULL.

For example, the following code fragment executes the UNIX **touch** command on a file called **somefile**:

```
char *argv[5];
argv[0] = "/usr/ucb/touch";
argv[1] = "somefile";
argv[2] = NULL;
mi_process_exec(argv);
```

The following code fragment executes the same command but uses the Bourne shell (**/bin/sh**) to execute it:

```
argv[0] = "/bin/sh";
argv[1] = "-c";
argv[2] = "touch";
argv[3] = "somefile";
argv[4] = NULL;
mi_process_exec(argv);
```

The new process inherits its execution environment from the parent server-initialization process (the operating-system process that runs the **oninit** utility or its equivalent). The user and group identifiers are those of the application user of the session, and the working directory is as follows:

```
$INFORMIXDIR/bin
```

This function is useful if you need to perform a UNIX or Linux **fork( )** and **exec( )** from a C UDR.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

The **mi_process_exec( )** return value is not a success indicator for the executed process. No feedback occurs between the C-function call and the actual **fork( )** and **exec( )** tasks, which takes place in the ADM VP at some time in the near future when the child-status timer goes off.

Related Topics

See also the descriptions of **mi_call( )** and **mi_call_on_vp( ).**

mi_put_bytes( )

The **mi_put_bytes( )** function copies the given number of bytes, converting any difference in alignment or byte order on the server computer to that of the client computer.

Syntax

```
mi_unsigned_char1 *mi_put_bytes (data_ptr, val_ptr, nbytes)
   mi_unsigned_char1 *data_ptr;
   char *val_ptr;
   mi_integer nbytes;
```

data_ptr	is a pointer to the buffer to which to copy bytes of data.
val_ptr	is a pointer to the buffer from which to copy bytes of data.
nbytes	is the number of bytes to copy.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_put_bytes( )** function copies *nbytes* bytes from the buffer that *val_ptr* references to the user-defined buffer that *data_ptr* references. Upon completion, **mi_put_bytes( )** returns the address of the next position to which data can be copied in the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *nbytes* bytes, ready for copying in the next value. In other words, if *n* is the length of the value that *val_ptr* identifies, the returned address is *n* bytes advanced from the original buffer address in *data_ptr*.

The **mi_put_bytes( )** function is useful in a send support function of an opaque data type that contains byte data. Use this function to send untyped data (such as **void** *) within an opaque-type internal structure to a client application (which possibly has unaligned data buffers).

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_fix_integer( ), mi_fix_smallint( ), mi_get_bytes( ), mi_put_date( ), mi_put_datetime( ), mi_put_decimal( ), mi_put_double_precision( ), mi_put_int8( ), mi_put_integer( ), mi_put_interval( ), mi_put_lo_handle( ), mi_put_money( ), mi_put_real( ), mi_put_smallint( ),** and **mi_put_string( ).**

For more information on the use of **mi_put_bytes( )** in a send support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_put_date( )

The **mi_put_date( )** function copies an **mi_date** (DATE) value, converting any difference in alignment or byte order on the server computer to that of the client computer.

Syntax

```
mi_unsigned_char1 *mi_put_date(data_ptr, date_ptr)
   mi_unsigned_char1 *data_ptr;
   mi_date *date_ptr;
```

data_ptr	is a pointer to the buffer to which to copy the **mi_date** value.
date_ptr	is a pointer to the buffer from which to copy the **mi_date** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_put_date( )** function copies the **mi_date** value from the buffer that *date_ptr* references into the user-defined buffer that *data_ptr* references. Upon completion, **mi_put_date( )** returns the address of the next position to which data can be copied in the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *nbytes* bytes, ready for copying in the next value. In other words, if *n* is the length of the value that *date_ptr* identifies, the returned address is *n* bytes advanced from the original buffer address in *data_ptr*.

The **mi_put_date( )** function is useful in a send support function of an opaque data type that contains an **mi_date** value. Use this function to send an **mi_date** field of an opaque-type internal structure to a client application (which possibly has unaligned data buffers).

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_date( ), mi_put_bytes( ), mi_put_datetime( ), mi_put_decimal( ), mi_put_double_precision( ), mi_put_int8( ), mi_put_integer( ), mi_put_interval( ), mi_put_lo_handle( ), mi_put_money( ), mi_put_real( ), mi_put_smallint( ),** and **mi_put_string( ).**

For more information on the use of **mi_put_date( )** in a send support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_put_datetime( )

The **mi_put_datetime( )** function copies an **mi_datetime** (DATETIME) value, converting any difference in alignment or byte order on the server computer to that of the client computer.

Syntax

```
mi_unsigned_char1 *mi_put_datetime(data_ptr, dtime_ptr)
   mi_unsigned_char1 *data_ptr;
   mi_datetime *dtime_ptr;
```

data_ptr	is a pointer to the buffer to which to copy the **mi_datetime** value.
dtime_ptr	is a pointer to the buffer from which to copy the **mi_datetime** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_put_datetime( )** function copies the **mi_datetime** value from the buffer that *dtime_ptr* references into the user-defined buffer that *data_ptr* references. Upon completion, **mi_put_datetime( )** returns the address of the next position to which data can be copied in the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *nbytes* bytes, ready for copying in the next value. In other words, if *n* is the length of the value that *dtime_ptr* identifies, the returned address is *n* bytes advanced from the original buffer address in *data_ptr*.

The **mi_put_datetime( )** function is useful in a send support function of an opaque data type that contains an **mi_datetime** value. Use this function to send an **mi_datetime** field of an opaque-type internal structure to a client application (which possibly has unaligned data buffers).

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_datetime( ), mi_put_bytes( ), mi_put_date( ), mi_put_decimal( ), mi_put_double_precision( ), mi_put_int8( ), mi_put_integer( ), mi_put_interval( ), mi_put_lo_handle( ), mi_put_money( ), mi_put_real( ), mi_put_smallint( ),** and **mi_put_string( ).**

For more information on the use of **mi_put_datetime( )** in a send support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_put_decimal( )

The **mi_put_decimal( )** function copies an **mi_decimal** (DECIMAL) value, converting any difference in alignment or byte order on the server computer to that of the client computer.

Syntax

```
mi_unsigned_char1 *mi_put_decimal(data_ptr, dec_ptr)
   mi_unsigned_char1 *data_ptr;
   mi_decimal *dec_ptr;
```

data_ptr	is a pointer to the buffer to which to copy the **mi_decimal** value.
dec_ptr	is a pointer to the buffer from which to copy the **mi_decimal** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_put_decimal( )** function copies the **mi_decimal** value from the buffer that *dec_ptr* references into the user-defined buffer that *data_ptr* references. Upon completion, **mi_put_decimal( )** returns the address of the next position to which data can be copied in the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *nbytes* bytes, ready for copying in the next value. In other words, if *n* is the length of the value that *dec_ptr* identifies, the returned address is *n* bytes advanced from the original buffer address in *data_ptr*.

The **mi_put_decimal( )** function is useful in a send support function of an opaque data type that contains an **mi_decimal** value. Use this function to send an **mi_decimal** field of an opaque-type internal structure to a client application (which possibly has unaligned data buffers).

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_decimal( ), mi_put_bytes( ), mi_put_date( ), mi_put_datetime( ), mi_put_double_precision( ), mi_put_int8( ), mi_put_integer( ), mi_put_interval( ), mi_put_lo_handle( ), mi_put_money( ), mi_put_real( ), mi_put_smallint( ),** and **mi_put_string( ).**

For more information on the use of **mi_put_decimal( )** in a send support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_put_double_precision( )

The **mi_put_double_precision( )** function copies an **mi_double_precision** (FLOAT) value, converting any difference in alignment or byte order on the server computer to that of the client computer.

Syntax

```
mi_unsigned_char1 *mi_put_double_precision(data_ptr, dbl_ptr)
   mi_unsigned_char1 *data_ptr;
   mi_double_precision *dbl_ptr;
```

data_ptr	is a pointer to the buffer to which to copy the **mi_double_precision** value.
dbl_ptr	is a pointer to the buffer from which to copy the **mi_double_precision** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_put_double_precision( )** function copies the **mi_double_precision** value from the buffer that *dbl_ptr* references into the user-defined buffer that *data_ptr* references. Upon completion, **mi_put_double_precision( )** returns the address of the next position to which data can be copied in the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *nbytes* bytes, ready for copying in the next value. In other words, if *n* is the length of the value that *dbl_ptr* identifies, the returned address is *n* bytes advanced from the original buffer address in *data_ptr*.

The **mi_put_double_precision( )** function is useful in a send support function of an opaque data type that contains an **mi_double_precision** value. Use this function to send an **mi_double_precision** field of an opaque-type internal structure to a client application (which possibly has unaligned data buffers).

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_double_precision( ), mi_put_bytes( ), mi_put_date( ), mi_put_datetime( ), mi_put_decimal( ), mi_put_int8( ), mi_put_integer( ), mi_put_interval( ), mi_put_lo_handle( ), mi_put_money( ), mi_put_real( ), mi_put_smallint( ),** and **mi_put_string( ).**

For more information on the use of **mi_put_double_precision( )** in a send support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_put_int8( )

The **mi_put_int8( )** function copies an **mi_int8** (INT8) value, converting any difference in alignment or byte order on the server computer to that of the client computer.

Syntax

```
mi_unsigned_char1 *mi_put_int8(data_ptr, int8_ptr)
   mi_unsigned_char1 *data_ptr;
   mi_int8 *int8_ptr;
```

data_ptr	is a pointer to the buffer to which to copy the **mi_int8** value.
int8_ptr	is a pointer to the buffer from which to copy the **mi_int8** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_put_int8( )** function copies the **mi_int8** value from the buffer that *int8_ptr* references into the user-defined buffer that *data_ptr* references. Upon completion, **mi_put_int8( )** returns the address of the next position to which data can be copied in the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *nbytes* bytes, ready for copying in the next value. In other words, if *n* is the length of the value that *int8_ptr* identifies, the returned address is *n* bytes advanced from the original buffer address in *data_ptr*.

The **mi_put_int8( )** function is useful in a send support function of an opaque data type that contains an **mi_int8** value. Use this function to send an **mi_int8** field of an opaque-type internal structure to a client application (which possibly has unaligned data buffers).

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_int8( ), mi_put_bytes( ), mi_put_date( ), mi_put_datetime( ), mi_put_decimal( ), mi_put_double_precision( ), mi_put_integer( ), mi_put_interval( ), mi_put_lo_handle( ), mi_put_money( ), mi_put_real( ), mi_put_smallint( ),** and **mi_put_string( ).**

For more information on the use of **mi_put_int8( )** in a send support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_put_integer( )

The **mi_put_integer( )** function copies an **mi_integer** (INTEGER) value, converting any difference in alignment or byte order on the server computer to that of the client computer.

Syntax

```
mi_unsigned_char1 *mi_put_integer (data_ptr, int_val)
   mi_unsigned_char1 *data_ptr;
   mi_integer int_val;
```

data_ptr	is a pointer to the buffer to which to copy the **mi_integer** value.
int_val	is a pointer to the buffer from which to copy the **mi_integer** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_put_integer( )** function copies the **mi_integer** value from the buffer that *int_val* references into the user-defined buffer that *data_ptr* references. Upon completion, **mi_put_integer( )** returns the address of the next position to which data can be copied in the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *nbytes* bytes, ready for copying in the next value. In other words, if *n* is the length of the value that *int_val* identifies, the returned address is *n* bytes advanced from the original buffer address in *data_ptr*.

The **mi_put_integer( )** function is useful in a send support function of an opaque data type that contains an **mi_integer** value. Use this function to send an **mi_integer** field of an opaque-type internal structure to a client application (which possibly has unaligned data buffers).

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_fix_integer( ), mi_get_integer( ), mi_put_bytes( ), mi_put_date( ), mi_put_datetime( ), mi_put_decimal( ), mi_put_double_precision( ), mi_put_int8( ), mi_put_interval( ), mi_put_lo_handle( ), mi_put_money( ), mi_put_real( ), mi_put_smallint( ),** and **mi_put_string( ).**

For more information on the use of **mi_put_integer( )** in a send support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_put_interval( )

The **mi_put_interval( )** function copies an **mi_interval** (INTERVAL) value, converting any difference in alignment or byte order on the server computer to that of the client computer.

Syntax

```
mi_unsigned_char1 *mi_put_interval(data_ptr, intrvl_ptr)
   mi_unsigned_char1 *data_ptr;
   mi_interval *intrvl_ptr;
```

data_ptr	is a pointer to the buffer to which to copy the **mi_interval** value.
intrvl_ptr	is a pointer to the buffer from which to copy the **mi_interval** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_put_interval( )** function copies the **mi_interval** value from the buffer that *intrvl_ptr* references into the user-defined buffer that *data_ptr* references. Upon completion, **mi_put_interval( )** returns the address of the next position to which data can be copied in the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *nbytes* bytes, ready for copying in the next value. In other words, if *n* is the length of the value that *intrvl_ptr* identifies, the returned address is *n* bytes advanced from the original buffer address in *data_ptr*.

The **mi_put_interval( )** function is useful in a send support function of an opaque data type that contains an **mi_interval** value. Use this function to send an **mi_interval** field of an opaque-type internal structure to a client application (which possibly has unaligned data buffers).

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_interval( ), mi_put_bytes( ), mi_put_date( ), mi_put_datetime( ), mi_put_decimal( ), mi_put_double_precision( ), mi_put_int8( ), mi_put_integer( ), mi_put_lo_handle( ), mi_put_money( ), mi_put_real( ), mi_put_smallint( ),** and **mi_put_string( ).**

For more information on the use of **mi_put_interval( )** in a send support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_put_lo_handle( )

The **mi_put_lo_handle( )** function copies an LO handle, converting any difference in alignment or byte order on the server computer to that of the client computer.

Syntax

```
mi_unsigned_char1 *mi_put_lo_handle(data_ptr, LO_hdl)
   mi_unsigned_char1 *data_ptr;
   MI_LO_HANDLE *LO_hdl;
```

data_ptr	is a pointer to the buffer to which to copy the LO handle.
LO_hdl	is a pointer to the LO handle to copy to the buffer.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_put_lo_handle( )** function copies the LO handle that LO_*hdl* references into the user-defined buffer that *data_ptr* references. Upon completion, **mi_put_lo_handle( )** returns the address of the next position to which data can be copied in the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *nbytes* bytes, ready for copying in the next value. In other words, if *n* is the length of the value that LO_*hdl* identifies, the returned address is *n* bytes advanced from the original buffer address in *data_ptr*.

The **mi_put_lo_handle( )** function is useful in a send support function of an opaque data type that contains a smart large object. Use this function to send an LO-handle field of an opaque-type internal structure to a client application (which possibly has unaligned data buffers).

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates the function was not successful.

Related Topics

See also the descriptions of **mi_get_lo_handle( ), mi_put_bytes( ), mi_put_date( ), mi_put_datetime( ), mi_put_decimal( ), mi_put_double_precision( ), mi_put_int8( ), mi_put_integer( ), mi_put_interval( ), mi_put_money( ), mi_put_real( ), mi_put_smallint( ),** and **mi_put_string( ).**

For more information on the use of **mi_put_lo_handle( )** in a send support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_put_money( )

The **mi_put_money( )** function copies an **mi_money** (MONEY) value, converting any difference in alignment or byte order on the server computer to that of the client computer.

Syntax

```
mi_unsigned_char1 *mi_put_money(data_ptr, money_ptr)
   mi_unsigned_char1 *data_ptr;
   mi_money *money_ptr;
```

data_ptr	is a pointer to the buffer to which to copy the **mi_money** value.
money_ptr	is a pointer to the buffer from which to copy the **mi_money** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_put_money( )** function copies the **mi_money** value from the buffer that *money_ptr* references into the user-defined buffer that *data_ptr* references. Upon completion, **mi_put_money( )** returns the address of the next position to which data can be copied in the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *nbytes* bytes, ready for copying in the next value. In other words, if *n* is the length of the value that *money_ptr* identifies, the returned address is *n* bytes advanced from the original buffer address in *data_ptr*.

The **mi_put_money( )** function is useful in a send support function of an opaque data type that contains an **mi_money** value. Use this function to send an **mi_money** field of an opaque-type internal structure to a client application (which possibly has unaligned data buffers).

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates the function was not successful.

Related Topics

See also the descriptions of **mi_get_money( ), mi_put_bytes( ), mi_put_date( ), mi_put_datetime( ), mi_put_decimal( ), mi_put_double_precision( ), mi_put_int8( ), mi_put_integer( ), mi_put_interval( ), mi_put_lo_handle( ), mi_put_real( ), mi_put_smallint( ),** and **mi_put_string( ).**

For more information on the use of **mi_put_money( )** in a send support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_put_real( )

The **mi_put_real( )** function copies an **mi_real** (SMALLFLOAT) value, converting any difference in alignment or byte order on the server computer to that of the client computer.

Syntax

```
mi_unsigned_char1 *mi_put_real(data_ptr, real_ptr)
   mi_unsigned_char1 *data_ptr;
   mi_real *real_ptr;
```

data_ptr	is a pointer to the buffer to which to copy the **mi_real** value.
real_ptr	is a pointer to the buffer from which to copy the **mi_real** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_put_real( )** function copies the **mi_real** value from the buffer that *real_ptr* references into the user-defined buffer that *data_ptr* references. Upon completion, **mi_put_real( )** returns the address of the next position to which data can be copied in the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *nbytes* bytes, ready for copying in the next value. In other words, if *n* is the length of the value that *real_ptr* identifies, the returned address is *n* bytes advanced from the original buffer address in *data_ptr*.

The **mi_put_real( )** function is useful in a send support function of an opaque data type that contains an **mi_real** value. Use this function to send an **mi_real** field of an opaque-type internal structure to a client application (which possibly has unaligned data buffers).

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates the function was not successful.

Related Topics

See also the descriptions of **mi_get_real( ), mi_put_bytes( ), mi_put_date( ), mi_put_datetime( ), mi_put_decimal( ), mi_put_double_precision( ), mi_put_int8( ), mi_put_integer( ), mi_put_interval( ), mi_put_lo_handle( ), mi_put_money( ), mi_put_smallint( ),** and **mi_put_string( ).**

For more information on the use of **mi_put_real( )** in a send support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_put_smallint( )

The **mi_put_smallint( )** function copies an **mi_smallint** (SMALLINT) value, converting any difference in alignment or byte order on the server computer to that of the client computer.

Syntax

```
mi_unsigned_char1 *mi_put_smallint (data_ptr, smallint_val)
   mi_unsigned_char1 *data_ptr;
   mi_integer smallint_val;
```

data_ptr	is the address of the buffer to which to copy an **mi_smallint** value.
smallint_val	is the *promoted* **mi_smallint** value to copy.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_put_smallint( )** function copies the **mi_smallint** value from the buffer that *smallint_val* references into the user-defined buffer that *data_ptr* references. Upon completion, **mi_put_smallint( )** returns the address of the next position to which data can be copied in the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *nbytes* bytes, ready for copying in the next value. In other words, if *n* is the length of the value that *smallint_val* identifies, the returned address is *n* bytes advanced from the original buffer address in *data_ptr*.

For maximum portability, this function accepts a fully promoted **mi_integer** value instead of an **mi_smallint** value. This argument might therefore require casting.

The **mi_put_smallint( )** function is useful in a send support function of an opaque data type that contains an **mi_smallint** value. Use this function to send an **mi_smallint** field of an opaque-type internal structure to a client application (which possibly has unaligned data buffers).

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_ptr* data buffer.
NULL	indicates the function was not successful.

Related Topics

See also the descriptions of **mi_fix_smallint( ), mi_get_smallint( ), mi_put_bytes( ), mi_put_date( ), mi_put_datetime( ), mi_put_decimal( ), mi_put_double_precision( ), mi_put_int8( ), mi_put_integer( ), mi_put_interval( ), mi_put_lo_handle( ), mi_put_money( ), mi_put_real( ),** and **mi_put_string( ).**

For more information on the use of **mi_put_smallint( )** in a send support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_put_string( )

The **mi_put_string( )** function copies an **mi_string** (CHAR(x)) value to a buffer.

Syntax

```
mi_unsigned_char1 *mi_put_string(data_dptr, string_buf, srcbytes)
   mi_unsigned_char1 **data_dptr;
   mi_string *string_buf;
   mi_integer srcbytes;
```

data_dptr	is a doubly indirected pointer to the buffer to which to copy the **mi_string** value.
string_buf	is a pointer to the buffer from which to copy the **mi_string** value.
srcbytes	is the number of source bytes to copy.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_put_string( )** function copies the **mi_string** value from the buffer that *string_buf* references into the buffer that *data_dptr* references.

Tip: While other **mi_put** functions accept a preallocated buffer, **mi_put_string( )** allocates memory for data to be copied. This allocation is why a pointer to the address is passed.

Upon completion, **mi_put_string( )** returns the address of the next position to which data can be copied in the *data_ptr* buffer. The function returns the *data_ptr* address advanced by *nbytes* bytes, ready for copying in the next value. In other words, if *srcbytes* is the length of the value that *string_buf* identifies, the returned address is *srcbytes* bytes advanced from the original buffer address in *data_dptr*.

The **mi_put_string( )** function is useful in a send support function of an opaque data type that contains an **mi_string** value. Use this function to send an **mi_string** field of an opaque-type internal structure to a client application (which possibly has unaligned data buffers).

Global Language Support

If code-set conversion is required, the **mi_put_string( )** function converts the **mi_string** value from the code set of the server-processing locale to the code set of the client locale. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Return Values

An **mi_unsigned_char1** pointer	is the new address in the *data_dptr* data buffer.
NULL	indicates the function was not successful.

Related Topics

See also the descriptions of **mi_get_string( ), mi_put_bytes( ), mi_put_date( ), mi_put_datetime( ), mi_put_decimal( ), mi_put_double_precision( ), mi_put_int8( ), mi_put_integer( ), mi_put_interval( ), mi_put_lo_handle( ), mi_put_money( ), mi_put_real( ),** and **mi_put_smallint( ).**

For more information on the use of **mi_put_string( )** in a send support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_query_finish( )

The **mi_query_finish( )** function finishes execution of the current statement.

Syntax

```
mi_integer mi_query_finish(conn)
   MI_CONNECTION *conn;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_query_finish( )** function completes execution of the current statement on the connection that *conn* references. The current statement is the most recently executed SQL statement sent to the database server on that connection. This function performs the following steps:

- If the current statement was a query:
  - Process any pending results that were not already processed with calls to **mi_next_row( ).**
  - Close any implicit cursor that **mi_exec( )** or **mi_exec_prepared_statement( )** opened to hold the rows.
- Release the resources for the current statement:
  - Free the implicit statement descriptor that **mi_exec( )** created.
  - Release other resources associated with the SQL 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.

After **mi_query_finish( )** executes, the next iteration of the **mi_get_result( )** function returns a status of MI_NO_MORE_RESULTS.

This function is useful for ensuring that a statement that returns no meaningful results, such as BEGIN WORK, executed successfully.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful, the current statement failed, or no statement is being processed.

Related Topics

See also the descriptions of **mi_command_is_finished( ), mi_server_connect( ), mi_get_result( ),** and **mi_query_interrupt( ).**

mi_query_interrupt( )

The **mi_query_interrupt( )** function interrupts the current statement.

Syntax

```
mi_integer mi_query_interrupt(conn, block_until_acknowledged)
   MI_CONNECTION *conn;
   mi_integer block_until_acknowledged;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
block_until_acknowledged	is currently ignored.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_query_interrupt( )** function interrupts execution of the current statement on the connection that *conn* references. The current statement is the most recently executed SQL statement sent to the database server on that connection. This function releases the resources for the current statement. If the current statement was a query, **mi_query_interrupt( )** closes any implicit cursor that **mi_exec( )** or **mi_exec_prepared_statement( )** opened to hold the rows.

The **mi_query_interrupt( )** function does not affect prepared statements or calls to DataBlade API file-access functions. After **mi_query_interrupt( )** executes, the next iteration of the **mi_get_result( )** function returns a status of MI_NO_MORE_RESULTS.

Return Values

MI_OK	indicates that the function was successful; the query was either successfully interrupted or had already completed.
MI_ERROR	indicates that the function was *not* successful; an exception was encountered.

Related Topics

See also the description of **mi_query_finish( ).**

mi_realloc( )

The **mi_realloc( )** function reallocates a block of user memory to a specified size and returns a pointer to that block.

Syntax

```
void *mi_realloc(void *ptr, mi_integer size)
mi_integer size;
```

ptr	is a pointer to a memory block.
size	is the number of bytes to reallocate.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_realloc( )** function changes the size of user memory to the specified size, with the current default memory duration. If the *ptr* parameter is NULL, then **mi_realloc( )** is a constructor function for user memory that behaves the same as **mi_alloc( )**.

The **mi_realloc( )** function returns a pointer to the reallocated user memory. Cast this pointer to match the structure of the user-defined buffer or structure that you allocate. A DataBlade API module can use **mi_free( )** to free memory allocated by **mi_realloc( )** when that memory is no longer needed.

Return Values

A **void** pointer	is a pointer to the reallocated memory. Cast this pointer to match the user-defined buffer or structure for which the memory was allocated.
NULL	indicates that the function was unable to allocate the memory.

The **mi_realloc( )** function does *not* throw an MI_Exception event when it encounters a runtime error. Therefore, it does not cause callbacks to be invoked.

Related Topics

See also the descriptions of **mi_alloc( ), mi_dalloc( ), mi_free( ), mi_switch_mem_duration( ),** and **mi_zalloc( ).**

For more information, see the discussion on how to allocate user memory in the *IBM Informix DataBlade API Programmer's Guide*.

mi_register_callback( )

The **mi_register_callback( )** function registers a callback function for a single event type or for all event types.

Syntax

```
MI_CALLBACK_HANDLE *mi_register_callback(conn, event_type,
   cback_func,user_data, parent)
   MI_CONNECTION *conn;
   MI_EVENT_TYPE event_type;
   MI_CALLBACK_FUNC cback_func;
   void  *user_data;
   MI_CALLBACK_HANDLE *parent;
```

conn	is either a NULL-valued pointer or a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
event_type	is the type of event that the *cback_func* callback handles. For a list of valid event types, see the *IBM Informix DataBlade API Programmer's Guide*.
cback_func	is a pointer to the callback function.
user_data	is a pointer to a user-provided structure that is passed to the callback function when the event specified by *event_type* occurs. It can be used to pass additional information to the callback.
parent	must be a NULL-valued pointer.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_register_callback( )** function registers the callback that *cback_func* identifies for the event that *event_type* specifies.

Server Only

For a C UDR, *conn* must be a NULL-valued pointer for the following event types:

- MI_EVENT_SAVEPOINT
- MI_EVENT_COMMIT_ABORT
- MI_EVENT_POST_XACT
- MI_EVENT_END_STMT
- MI_EVENT_END_XACT
- MI_EVENT_END_SESSION

For the MI_Exception event, *conn* can be either a valid connection descriptor or a NULL-valued pointer. For more information, see the *IBM Informix DataBlade API Programmer's Guide*.

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

End of Client Only

The callback is enabled when it is registered. You can explicitly disable the callback with the **mi_disable_callback( )** function and reenable it with **mi_enable_callback( ).**

The **mi_register_callback( )** function initializes the DataBlade API when it is the first DataBlade API function in a client LIBMI application or a C UDR. For more information, see the *IBM Informix DataBlade API Programmer's Guide*.

You can use the *user_data* parameter to specify the address of a user-defined error structure, which the callback can use to hold additional error information.

Return Values

An **MI_CALLBACK_HANDLE** pointer	is a pointer to the newly registered callback function.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_disable_callback( ), mi_enable_callback( ), mi_retrieve_callback( ),** and **mi_unregister_callback( ).**

mi_result_command_name( )

The **mi_result_command_name( )** function returns the name of the SQL statement that is the current statement.

Syntax

```
char *mi_result_command_name(conn)
   MI_CONNECTION *conn;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_result_command_name( )** function returns the name of the current statement on the connection that *conn* references. The current statement is the most recently executed SQL statement sent to the database server on that connection. The function returns the name as a null-terminated string in memory that it has allocated in the current memory duration. This statement name is only the verb of the statement, not the entire statement syntax.

For example, suppose the **mi_exec( )** function sends the following SELECT statement to the database server over the *conn* connection:

```
SELECT * FROM customer WHERE state = "CA";
```

The following call to **mi_result_command_name( )** returns only the verb of this statement: `select`:

```
char *cmd_name;
...
cmd_name = mi_result_command_name(conn);
```

Important: Use the **mi_result_command_name( )** function only after the **mi_get_result( )** function returns *MI_DML* or *MI_DDL*. To obtain the name of the SQL statement that invoked a C UDR, use the **mi_current_command_name( )** function. To obtain the statement name of a prepared statement, use the **mi_statement_command_name( )** function.

Return Values

An **mi_string** pointer	is a pointer to the verb of the last statement or command.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_current_command_name( ), mi_get_result( ), mi_result_row_count( ),** and **mi_statement_command_name( ).**

mi_result_reference( )

The mi_result_reference( ) function returns the reference of the last object inserted on the connection.

Syntax

```
mi_integer mi_result_reference(conn, retbuf)
   MI_CONNECTION *conn;
   mi_ref *retbuf;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
retbuf	is the reference of the new row. The *retbuf* parameter is set to `0` in any of the following cases: • The INSERT command is not over a named row type. • The INSERT command was not WITH REFERENCE. • Multiple rows are inserted.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The mi_get_result( ) function must be called before **mi_result_reference( ).** The **mi_result_reference( )** function is valid only after an insert.

Return Values

The reference of the last object inserted on the connection.

mi_result_row_count( )

The **mi_result_row_count( )** function returns the number of rows affected by the current statement.

Syntax

```
mi_integer mi_result_row_count(conn)
   MI_CONNECTION *conn;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

The **mi_result_row_count( )** function returns the number of rows that the current statement on the connection that *conn* references has affected. The current statement is the most recently executed SQL statement sent to the database server on that connection. For example, if an UPDATE statement modifies three rows, a call to **mi_result_row_count( )** returns 3. If a SELECT statement returns 531 rows, **mi_result_row_count( )** returns 531.

Important: Use the **mi_result_row_count( )** function only after the **mi_get_result( )** function returns *MI_DML*.

Return Values

>=0	is the number of rows affected.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_get_result( )** and **mi_result_command_name( ).**

mi_retrieve_callback( )

The **mi_retrieve_callback( )** function retrieves the handle of a registered callback.

Syntax

```
mi_integer mi_retrieve_callback(conn, event_type, cback_handle,
   cback_func,user_data)
   MI_CONNECTION *conn;
   MI_EVENT_TYPE event_type;
   MI_CALLBACK_HANDLE *cback_handle;
   MI_CALLBACK_FUNC *cback_func;
   void **user_data;
```

conn	is either NULL or a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
event_type	is the type of event that the *cback_handle* callback handles. For a list of valid event types, see the *IBM Informix DataBlade API Programmer's Guide*.
cback_handle	is the callback handle from a previous call to **mi_register_callback( ).**
cback_func	is a pointer to the location at which to return a pointer to the callback function.
user_data	is the user-supplied argument to the callback function.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

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 temporarily change the callback that is registered for a particular event.

Server Only

For a C UDR, *conn* must be NULL for the following event types:
- 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

Client Only

For a client LIBMI application, you must provide a valid connection descriptor to **mi_retrieve_callback( )** for callbacks that handle the following event types:

- MI_Exception
- MI_Xact_State_Change
- MI_Client_Library_Error

End of Client Only

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the description of **mi_register_callback( ).**

mi_routine_end( )

The **mi_routine_end( )** function releases resources associated with a function descriptor.

Syntax

```
mi_integer mi_routine_end(conn, funcdesc_ptr)
   MI_CONNECTION *conn;
   MI_FUNC_DESC *funcdesc_ptr;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
	can be a pointer to a session-duration connection descriptor established by a previous call to **mi_get_session_connection( ).** Use of a session-duration connection descriptor is an *advanced* feature of the DataBlade API.
funcdesc_ptr	is a pointer to the function descriptor to deallocate.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_routine_end( )** function frees the function descriptor that *funcdesc_ptr* references. This function is the destructor function for the function descriptor. It frees the memory for the **MI_FPARAM** structure that is stored in the function descriptor and for the function descriptor itself.

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.

The **mi_routine_end( )** function is one of the functions of the Fastpath interface.

Server Only

The **mi_routine_end( )** function is also the destructor function for the session-duration function descriptor. It frees memory for the session-duration function descriptor (including its **MI_FPARAM** structure). However, you must explicitly free any PER_SESSION named memory that holds the function descriptor.

Warning: Session-duration function descriptors and named memory are advanced features of the DataBlade API. They can adversely affect your UDR if you use them incorrectly. Use them only when a regular function descriptor cannot perform the task you need done. For more information, see the *IBM Informix DataBlade API Programmer's Guide*.

End of Server Only

Return Values

MI_OK	indicates that the function was successful.

MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_cast_get( ), mi_func_desc_by_typeid( ), mi_routine_exec( ), mi_routine_get( ), mi_routine_get_by_typeid( ),** and **mi_td_cast_get( ).**

mi_routine_exec( )

The **mi_routine_exec( )** function executes the registered user-defined routine or cast function associated with a specified function descriptor.

Syntax

```
MI_DATUM mi_routine_exec(conn, funcdesc_ptr, error, argument_list)
   MI_CONNECTION *conn;
   MI_FUNC_DESC *funcdesc_ptr;
   mi_integer *error;
   argument_list;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).** can be a pointer to a session-duration connection descriptor established by a previous call to **mi_get_session_connection( ).** Use of a session-duration connection descriptor is an *advanced* feature of the DataBlade API.
funcdesc_ptr	is a pointer to the function descriptor that describes the routine to execute.
error	is set to the status of the **mi_routine_exec( )** function: • MI_OK: **mi_routine_exec( )** is successful. If **mi_routine_exec( )** returns NULL, MI_OK indicates that the return value of the executed user-defined routine is NULL. • MI_ERROR: **mi_routine_exec( )** is *not* successful; it returns NULL.
argument_list	is a list of the routine arguments passed with the appropriate passing mechanism for their data type.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_routine_exec( )** function is one of the functions of the Fastpath interface. It performs the following tasks:

1. Copies the arguments into the function descriptor that *funcdesc_ptr* identifies

   To obtain a function descriptor for the user-defined routine, use the **mi_routine_get( ), mi_routine_get_by_typeid( ), mi_cast_get( ),** or **mi_td_cast_get( )** function. The *argument_list* specifies the arguments of the user-defined routine. If you do not specify an argument value, **mi_routine_exec( )** uses the default value assigned to that argument. If the user-defined routine does not have any arguments, you can omit the *argument_list* argument from **mi_routine_exec( )**.
2. Executes the routine within the routine sequence of this function descriptor

   The values within the corresponding **MI_FPARAM** structure for the routine are consistent between function invocations within the sequence. Use the **mi_fparam_get( )** function to obtain this **MI_FPARAM** structure.

3. Returns an **MI_DATUM** value that contains the return value of the executed user-defined routine

   A NULL return value means that either the user-defined routine returned a NULL value or that the **mi_routine_exec( )** function failed. The *error* argument holds the status of the **mi_routine_exec( )** function. For more information on **MI_DATUM** values, see the *IBM Informix DataBlade API Programmer's Guide*.

The **mi_routine_exec( )** function can execute routines across databases.

The following call executes a user-defined function named **a_func( )**, which returns an integer value:

```
MI_CONNECTION *conn;
MI_FUNC_DESC *func_desc;
MI_DATUM ret_val;
mi_integer arg1, error;
...
func_desc = mi_routine_get(conn, 0, "a_func(mi_integer)");
ret_val = (mi_integer) mi_routine_exec(conn, func_desc,
   &error, arg1);
if ( ret_val == NULL ) AND ( error == MI_ERROR )
   /* generate an error */
else
   /* obtain function return value from ret_val */
```

Important: You cannot use the Fastpath interface to execute an iterator function or an SPL function that has "WITH RESUME" in its RETURN statement. For more information on iterator functions, see the *IBM Informix DataBlade API Programmer's Guide*.

Return Values

An **MI_DATUM** value	is the return value of the executed user-defined routine.
NULL	indicates that the user-defined routine returned NULL or that the **mi_routine_exec( )** function was not successful. Check the value of error to determine which of these events has occurred.

Related Topics

See also the descriptions of **mi_cast_get( ), mi_fparam_get( ), mi_routine_end( ), mi_routine_get( ), mi_routine_get_by_typeid( ),** and **mi_td_cast_get( ).**

mi_routine_get( )

The **mi_routine_get( )** function looks up a registered user-defined routine by a routine signature that is a character string and creates its function descriptor.

Syntax

```
MI_FUNC_DESC *mi_routine_get(conn, flags, rout_sig)
   MI_CONNECTION *conn;
   mi_integer flags;
   char *rout_sig;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).** can be a pointer to a session-duration connection descriptor established by a previous call to **mi_get_session_connection( ).** Use of a session-duration connection descriptor is an *advanced* feature of the DataBlade API.
flags	must be `0`.
rout_sig	is a character string that specifies the routine signature of the user-defined routine to be looked up. This signature has the following format: [*udr_type*] [*owner*.]*udr_name*([*parm1*],...,[*parmN*]) For more information on the syntax of the *rout_sig* argument, see the description in the following "Usage" section.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_routine_get( )** function obtains a function descriptor for the UDR that the *rout_sig* argument specifies. The *rout_sig* argument specifies the routine signature of the UDR in the following format:

```
[udr_type] [owner.]udr_name([parm1], ..., [parmN])
```

udr_type	is the word *function* (the default) or *procedure*.
owner	is the name of the UDR owner. When the UDR is defined in a database that is not ANSI-compliant, **mi_routine_get( )** looks for UDRs owned only by *owner*. If you specify a NULL-valued pointer for *owner*, **mi_routine_get( )** looks for UDRs owned by anyone. When the UDR is defined in an ANSI-complaint database, *owner* is part of its routine signature. You can specify a particular user name for *owner* to obtain UDRs of a particular owner. If you specify a NULL-valued pointer for *owner*, **mi_routine_get( )** uses the current user account as the *owner* name. If no UDRs exist for the current user, **mi_routine_get( )** looks for UDRs with user **informix** as the owner name.
udr_name	is the name of the user-defined routine to look up.

parm1,...parmN is an optional list of data types for the parameters of the user-defined routine.

This function is one of the functions of the Fastpath interface. It is a constructor function for the function descriptor.

To obtain a function descriptor for a UDR, the **mi_routine_get( )** function performs the following tasks:

1. Looks for a user-defined routine that matches the *rout_sig* routine signature in the **sysprocedures** system catalog table
2. Allocates a function descriptor for the UDR and saves the routine sequence in this descriptor
3. Allocates an **MI_FPARAM** structure for the routine and saves the argument and return-value information in this structure
4. Returns a pointer to the function descriptor that it allocated for the user-defined routine

Server Only

When you pass a public connection descriptor (from **mi_open( )**), the **mi_routine_get( )** function allocates the new function descriptor in the PER_COMMAND memory duration. If you pass a session-duration connection descriptor (from **mi_get_session_connection( )**), **mi_routine_get( )** allocates the new function descriptor in the PER_SESSION memory duration. This function descriptor is called a session-duration function descriptor. For more information, see the *IBM Informix DataBlade API Programmer's Guide*.

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. Use them only when a regular connection descriptor or function descriptor cannot perform the task you need done.

End of Server Only

The following call to **mi_routine_get( )** looks for the **a_udr( )** user-defined function in a database that is *not* ANSI compliant:

```
func_desc = mi_routine_get(conn, 0,
   "a_udr(integer, integer)");
```

American National Standards Institute

An ANSI-compliant database requires an *owner* name as part of a routine name. If **a_udr( )** was defined in an ANSI-compliant database, you must include the owner name in the routine signature, as follows:

```
func_desc = mi_routine_get(conn, 0,
   "dexter.a_udr(integer, integer)");
```

End of American National Standards Institute

The *udr_type* part of the routine signature is optional. As the preceding examples show, *udr_type* defaults to FUNCTION when this part is omitted from the routine signature. If a user-defined procedure and a user-defined function have the same routine name, include *udr_type* in the *rout_sig* signature. The following call to **mi_routine_get( )** specifies that **a_udr( )** is a user-defined function:

```
func_desc = mi_routine_get(conn, 0,
   "function a_udr(integer, integer)");
```

For user-defined procedures, specify the PROCEDURE keyword instead.

Return Values

An **MI_FUNC_DESC** pointer	is a pointer to the function descriptor for the routine that *rout_sig* specifies.
NULL	indicates that no matching user-defined routine was found or that the specified user-defined routine has multiple return values, which is possible with: • SPL routines that include the WITH RESUME clause in their RETURN statement • Iterator functions

Other internal errors raise an exception.

Related Topics

See also the descriptions of **mi_cast_get( ), mi_fparam_get( ), mi_func_desc_by_typeid( ), mi_routine_end( ), mi_routine_exec( ),** and **mi_routine_get_by_typeid( ).**

mi_routine_get_by_typeid( )

The **mi_routine_get_by_typeid( )** function looks up a registered user-defined routine (UDR) on the local database server by a routine signature that the function builds from a list of arguments. This function also creates a function descriptor for the UDR.

Syntax

```
MI_FUNC_DESC *mi_routine_get_by_typeid(conn, udr_type, udr_name,
   owner,arg_count, arg_types)
   MI_CONNECTION *conn;
   MI_UDR_TYPE udr_type;
   char *udr_name;
   char *owner;
   mi_integer arg_count;
   MI_TYPEID *arg_types;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).** can be a pointer to a session-duration connection descriptor established by a previous call to **mi_get_session_connection( ).** Use of a session-duration connection descriptor is an *advanced* feature of the DataBlade API.
udr_type	is a value of type **MI_UDR_TYPE** that indicates whether the user-defined routine is a function or a procedure:

MI_FUNC	The user-defined routine is a function.
MI_PROC	The user-defined routine is a procedure.

udr_name	is the name of the user-defined routine.
owner	is the owner of the routine. For more information on how to specify an owner name, see the following "Usage" section.
arg_count	is the integer number of arguments that the user-defined routine takes.
arg_types	is an array of pointers to type identifier, one type identifier for each of the *udr_name* routine arguments.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_routine_get_by_typeid( )** function obtains a function descriptor for the UDR that the routine signature specifies. This function builds the routine signature from the *udr_type*, *udr_name*, *owner*, *arg_count*, and *arg_types* arguments. The **mi_routine_get_by_typeid( )** function is one of the functions of the Fastpath interface. It is a constructor function for the function descriptor.

The **mi_routine_get_by_typeid( )** function performs the following tasks:

1. Looks for a user-defined routine that matches the routine signature in the **sysprocedures** system catalog table

2. Allocates a function descriptor for the routine and saves the routine sequence in this descriptor
3. Allocates an **MI_FPARAM** structure for the routine and saves the argument and return-value information in this structure
4. Returns a pointer to the function descriptor that it allocated for the user-defined routine

Server Only

When you pass a public connection descriptor (from **mi_open( )**), the **mi_routine_get_by_typeid( )** function allocates the new function descriptor in the PER_COMMAND memory duration. If you pass a session-duration connection descriptor (from **mi_get_session_connection( )**), **mi_routine_get_by_typeid( )** allocates the new function descriptor in the PER_SESSION memory duration. This function descriptor is called a session-duration function descriptor. For more information, see the *IBM Informix DataBlade API Programmer's Guide*.

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. Use them only when a regular connection descriptor or function descriptor cannot perform the task you need done.

End of Server Only

When the UDR is defined in a database that is not ANSI compliant, **mi_routine_get_by_typeid( )** looks for UDRs owned only by *owner*. If you specify a NULL-valued pointer as an *owner*, **mi_routine_get_by_typeid( )** looks for UDRs owned by anyone. The following call to **mi_routine_get_by_typeid( )** looks for the **a_proc( )** user-defined procedure in a database that is *not* ANSI compliant:

```
MI_TYPEID *arg_types[2];
MI_FUNC_DESC *func_desc;
MI_CONNECTION *conn;
...
arg_types[0] = mi_typestring_to_id(conn, "integer");
arg_types[1] = mi_typestring_to_id(conn, "datetime");
func_desc = mi_routine_get_by_typeid(conn, MI_PROC,
   "a_proc", NULL, 2, arg_types);
```

American National Standards Institute

When the UDR is defined in an ANSI-complaint database, *owner* is part of its routine signature. You can specify a particular *owner* value to obtain UDRs of a particular owner. If you specify a NULL-valued pointer as an *owner*, **mi_routine_get_by_typeid( )** uses the current user account as the *owner* value. If no UDRs exist for the current user, **mi_routine_get_by_typeid( )** looks for UDRs with user **informix** as the owner name.

If **a_proc( )** was defined in an ANSI-compliant database, the following call to **mi_routine_get_by_typeid( )** looks up the **a_proc( )** user-defined procedure owned by user **dexter**:

```
func_desc = mi_routine_get_by_typeid(conn, 0, MI_PROC,
   "a_udr", "dexter", 2, arg_types);
```

End of American National Standards Institute

Return Values

An **MI_FUNC_DESC** pointer	is a pointer to the function descriptor of the specified user-defined routine.
NULL	indicates that no matching user-defined routine was found or that the specified user-defined routine has multiple return values. The following routines can have multiple return values: • SPL routines that include the WITH RESUME clause in the RETURN statement • Iterator functions

Other internal errors raise an exception.

Related Topics

See also the descriptions of **mi_fparam_get( ), mi_func_desc_by_typeid( ), mi_routine_end( ), mi_routine_exec( ), mi_routine_get( ), mi_routine_get_by_typeid( ),** and **mi_td_cast_get( ).**

mi_routine_id_get( )

The **mi_routine_id_get( )** accessor function returns the routine identifier for the user-defined routine that the specified function descriptor describes.

Syntax

```
mi_integer mi_routine_id_get(conn, funcdesc_ptr)
   MI_CONNECTION *conn;
   MI_FUNC_DESC *funcdesc_ptr;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
funcdesc_ptr	is a pointer to a function descriptor for a user-defined routine.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_routine_id_get( )** function is one of the DataBlade API functions of the Fastpath interface. It returns the routine identifier for the UDR that the *funcdesc_ptr* function descriptor describes. The routine identifier uniquely identifies a user-defined routine. The database server generates this identifier and stores it in the **procid** column of the **sysprocedures** system catalog table.

Tip: The DataBlade API provides the **mi_funcid** data type for routine identifiers. The **mi_funcid** data type has the same structure as the **mi_integer** data type. For compatibility with earlier versions, **mi_routine_id_get( )** continues to return a routine identifier as an **mi_integer** value.

Return Values

>=0	is the routine identifier of the routine that *funcdesc_ptr* identifies.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_cast_get( ), mi_fparam_get( ), mi_func_handlesnulls( ), mi_func_isvariant( ), mi_func_negator( ), mi_routine_get( ), mi_routine_get_by_typeid( ),** and **mi_td_cast_get( ).**

mi_row_create( )

The **mi_row_create( )** function creates a row, based on a row descriptor and column data.

Syntax

```
MI_ROW *mi_row_create(conn, row_desc, col_values, col_nulls)
   MI_CONNECTION  *conn;
   MI_ROW_DESC *row_desc;
   MI_DATUM col_values[];
   mi_boolean col_nulls[];
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
row_desc	is a pointer to the row descriptor that describes the columns of the row.
col_values	is an array of **MI_DATUM** structures that contain the values of the columns in the row (or fields in a row type).
col_nulls	is an array that indicates whether a column holds an SQL NULL value. Each array element is set to one of the following values:

1 (MI_TRUE)	The value of the associated column is an SQL NULL value.
0 (MI_FALSE)	The value of the associated column is *not* an SQL NULL value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_row_create( )** function takes the column data that the *col_values* and *col_nulls* arrays hold and creates a row structure that holds this data, based on the column information in the row descriptor that *row_desc* references. The function returns a pointer to the new row structure (**MI_ROW**) that it creates. The **mi_row_create( )** function is the constructor function for the row structure.

The *col_values* array holds the values for each column of the row. This array sends the column values as **MI_DATUM** values. The **mi_row_create( )** function expects the *col_values* value for a CHAR(*n*) or NCHAR(*n*) column in its binary representation (in an **mi_lvarchar** variable-length structure). The **mi_row_create( )** function does perform a deep copy of the array values; that is, it copies both the pointer and its associated memory into the new row structure.

Server Only

The **mi_row_create( )** function allocates a new row structure with the PER_COMMAND memory duration. Values in the *col_values* array can be passed by reference or by value, depending on the data type of the value.

In a C UDR, the row structure and row descriptor are part of the same data structure. To create a row structure, the **mi_row_create( )** function just adds a data buffer, which holds copies of the values in the *col_values* and *col_nulls* arrays, to the row descriptor that *row_desc* references. The address of this row structure is the address of the data buffer within the row descriptor. There is a one-to-one correspondence between a row descriptor and its row structure.

When you call **mi_row_create( )** twice with the same row descriptor, the second call overwrites the column values of the first call, as follows:

- In the first call, the **mi_row_create( )** function just adds a data buffer to the specified row descriptor and copies the column values for the row into this data buffer.
- If you call **mi_row_create( )** a second time with the *same* row descriptor, this second call copies the new column values into the row structure associated with this row descriptor.

This behavior can be beneficial in that it saves a call to **mi_row_free( )** for the first data buffer. However, it does overwrite the column values from the first **mi_row_create( )** call with the new column values.

End of Server Only

Client Only

Values in the *col_values* array must be pass by reference for *all* data types.

In a client LIBMI application, the row structure and row descriptor are separate data structures. There is a one-to-many correspondence between a row descriptor and its associated row structures. When you call **mi_row_create( )** a second time on the same row descriptor, you obtain a second row structure that points to the same row descriptor.

End of Client Only

Return Values

An **MI_ROW** pointer	is a pointer to the newly created row.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_row_desc_create( )** and **mi_row_free( ).**

mi_row_desc_create( )

The **mi_row_desc_create( )** function creates a row descriptor from a specified type identifier.

Syntax

```
MI_ROW_DESC *mi_row_desc_create(typeid)
   MI_TYPEID *typeid;
```

typeid	is a pointer to the type identifier for the data type of the row descriptor to create.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_row_desc_create( )** function creates a row descriptor for the row type that the *typeid* type identifier specifies. This function is the constructor function for the row descriptor.

Server Only

In a C UDR, the row structure and row descriptor are part of the same data structure. The row structure is just a data buffer in the row descriptor that holds copies of the column values of a row. The **mi_row_desc_create( )** function allocates a row descriptor with a NULL-valued pointer for the row structure. Use the **mi_row_create( )** function to add the data buffer to the row descriptor.

The **mi_row_desc_create( )** function allocates a new row descriptor with the current memory duration.

End of Server Only

Client Only

In a client LIBMI application, the row structure and row descriptor are separate data structures.

End of Client Only

Return Values

An **MI_ROW_DESC** pointer	is a pointer to a row descriptor for the specified data type.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_row_create( )** and **mi_row_desc_free( ).**

mi_row_desc_free( )

The **mi_row_desc_free( )** routine frees a row descriptor.

Syntax

```
void mi_row_desc_free(row_desc)
   MI_ROW_DESC *row_desc;
```

row_desc	is a pointer to the row descriptor to be freed.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_row_desc_free( )** function frees the row descriptor that *row_desc* references. The **mi_row_desc_free( )** function is the destructor function for the row descriptor. However, only use **mi_row_desc_free( )** to free a row descriptor that was created with **mi_row_desc_create( ).** Do *not* use it to free a row descriptor that a DataBlade API function allocates. For example, do not use **mi_row_desc_free( )** to free the row descriptor for the current statement, which the **mi_get_row_desc_without_row( )** function allocates.

Server Only

In a C UDR, the row structure and row descriptor are part of the same data structure. The row structure is just a data buffer in the row descriptor that holds copies of the column values of a row. Therefore, the **mi_row_desc_free( )** function automatically frees both the row descriptor *and* the associated row structure. Examine the contents of a row structure *before* you deallocate the associated row descriptor with **mi_row_desc_free( )**.

End of Server Only

Client Only

In a client LIBMI application, the row structure and row descriptor are separate data structures. The **mi_row_desc_free( )** function only frees a row descriptor. It does not affect the associated row structure.

In a client LIBMI application, a row structure and a row descriptor are separate data structures. There can be a one-to-many correspondence 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

Return Values

None.

Related Topics

See also the descriptions of **mi_get_row_desc_without_row( ), mi_row_create( )** and **mi_row_desc_create( ).**

mi_row_free( )

The **mi_row_free( )** function frees a row structure.

Syntax

```
mi_integer mi_row_free(row)
   MI_ROW *row;
```

row	is a pointer to the row structure to be freed.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_row_free( )** function frees the row structure that *row* references. This function is the destructor function for a row structure. Use **mi_row_free( )**, however, only to free a row structure that **mi_row_create( )** created. Do *not* use **mi_row_free( )** to free a row structure that a DataBlade API function allocated. For example, do not use **mi_row_free( )** to free the row structure for the current statement, which the **mi_next_row( )** function accesses.

Server Only

In a C UDR, the row structure and row descriptor are part of the same data structure. The row structure is just a data buffer in the row descriptor that holds copies of the column values of a row. The **mi_row_free( )** function frees this data buffer and sets the pointer to this data buffer (within the row descriptor) to a NULL-valued pointer.

After a call to **mi_row_free( )**, the row structure is no longer accessible but the row descriptor is. However, the **mi_row_desc_free( )** function frees both the row descriptor *and* its associated row structure. Therefore, after a call to **mi_row_desc_free( ),** neither the row structure nor the row descriptor are accessible. To explicitly free a row structure you have allocated with **mi_row_create( ),** call **mi_row_free( )** *before* you free the row descriptor with **mi_row_desc_free( ).**

End of Server Only

Client Only

In a client LIBMI application, the row structure and row descriptor are separate data structures. The **mi_row_free( )** function only frees a row structure. It does not affect the associated row descriptor.

End of Client Only

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_next_row( ), mi_row_create( )** and **mi_row_desc_free( ).**

mi_save_set_count( )

The **mi_save_set_count( )** function returns the number of rows in a save set.

Syntax

```
mi_integer mi_save_set_count(save_set)
   MI_SAVE_SET *save_set;
```

save_set	is a pointer to an **MI_SAVE_SET** structure that a previous call to **mi_save_set_create( )** created.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Return Values

>=0	is the number of rows in the save set.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_save_set_create( )** and **mi_save_set_member( ).**

mi_save_set_create( )

The **mi_save_set_create( )** function creates a save set on the current connection.

Syntax

```
MI_SAVE_SET *mi_save_set_create(conn)
   MI_CONNECTION *conn;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_save_set_create( )** function is a constructor function for a save set. A save set is an area of memory used to store rows fetched from a DataBlade API function. It provides a mechanism for manipulating multiple database rows in memory. Rows stored in a save set can be traversed with the DataBlade API functions **mi_save_set_get_first( ), mi_save_set_get_next( ),** and **mi_save_set_get_previous( ).**

Server Only

The **mi_save_set_create( )** function allocates a new save-set structure in the PER_STMT_EXEC memory duration.

End of Server Only

When the DataBlade API module no longer requires the save set, free the save-set resources with **mi_save_set_destroy( ).** A save set is freed when the connection on which it was created is closed.

Return Values

An **MI_SAVE_SET** pointer	is a pointer to a new save set.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_save_set_count( ), mi_save_set_delete( ), mi_save_set_destroy( ), mi_save_set_get_first( ), mi_save_set_get_last( ), mi_save_set_get_next( ), mi_save_set_get_previous( ), mi_save_set_insert( ),** and **mi_save_set_member( ).**

mi_save_set_delete( )

The **mi_save_set_delete( )** function removes a row from a save set.

Syntax

```
mi_integer mi_save_set_delete(row)
   MI_ROW *row;
```

row	is a pointer to a row in a save set.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_save_set_delete( )** function is a destructor function for a save set.

Return Values

MI_OK	indicates that the row was deleted from the save set.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_save_set_get_first( ), mi_save_set_get_last( ), mi_save_set_get_next( ), mi_save_set_get_previous( ),** and **mi_save_set_insert( ).**

mi_save_set_destroy( )

The **mi_save_set_destroy( )** function destroys a save set and frees its resources.

Syntax

```
mi_integer mi_save_set_destroy(save_set)
   MI_SAVE_SET *save_set;
```

save_set	is a pointer to an **MI_SAVE_SET** structure.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_save_set_destroy( )** function frees resources for the save set that *save_set* references. This function is the destructor function for a save set. This save set must have been previously created with **mi_save_set_create( ).** It is an error to attempt to access rows in a save set that was destroyed.

Important: It is recommended that you explicitly deallocate save sets with **mi_save_set_destroy( )** once you no longer need them. Otherwise, these save sets remain until the associated SQL statement ends or the session closes (whichever occurs first).

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the description of **mi_save_set_create( ).**

mi_save_set_get_first( )

The **mi_save_set_get_first( )** function retrieves the first row from a save set.

Syntax

```
MI_ROW *mi_save_set_get_first(save_set, error)
   MI_SAVE_SET *save_set;
   mi_integer *error;
```

save_set	is a pointer to an **MI_SAVE_SET** structure.
error	is a pointer to a return code.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_save_set_get_first( )** function obtains the first row from the save set that *save_set* references. Because the DataBlade API maintains a save set as an FIFO (first-in, first-out) queue, the first row is the row most recently added to the save set; that is, it is the row at the top of the queue. This save set must have been previously created with **mi_save_set_create( ).**

When the **mi_save_set_get_first( )** function is successful, it returns a pointer to the row structure for the first row. It also moves the cursor position to point to the first row as the current row in the save set. If the save set is empty, the function takes the following steps:

1. Returns the NULL-valued pointer
2. Sets the *error* argument to MI_NO_MORE_RESULTS

Return Values

An **MI_ROW** pointer	is a pointer to the first row in the specified save set.
NULL	indicates that the function was not successful or that the save set is empty.

Upon failure, **mi_save_set_get_first( )** returns NULL and sets *error* to MI_ERROR.

Related Topics

See also the descriptions of **mi_save_set_get_last( ), mi_save_set_get_next( ),** and **mi_save_set_get_previous( ).**

mi_save_set_get_last( )

The **mi_save_set_get_last( )** function retrieves the last row from a save set.

Syntax

```
MI_ROW *mi_save_set_get_last (save_set, error)
   MI_SAVE_SET *save_set;
   mi_integer *error;
```

save_set	is a pointer to an **MI_SAVE_SET** structure.
error	is a pointer to a return code.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_save_set_get_last( )** function obtains the last row from the save set that *save_set* references. Because the DataBlade API maintains a save set as an FIFO (first-in, first-out) queue, the last row is the first row added to the save set; that is, it is the row at the bottom of the queue. This save set must have been previously created with **mi_save_set_create( ).**

When the **mi_save_set_get_last( )** function is successful, it returns a pointer to the row structure for the last row. It also moves the cursor position to point to the last row as the current row in the save set. If **mi_save_set_get_last( )** *cannot find the last row*, the function takes the following steps:

1. Returns the NULL-valued pointer
2. Sets the *error* argument to MI_NO_MORE_RESULTS

Return Values

An **MI_ROW** pointer	is a pointer to the final row in the specified save set.
NULL	indicates that the function was not successful or the function cannot find the last row.

Upon failure, **mi_save_set_get_last( )** returns NULL and sets *error* to MI_ERROR.

Related Topics

See also the descriptions of **mi_save_set_count( ), mi_save_set_create( ), mi_save_set_delete( ), mi_save_set_destroy( ), mi_save_set_get_first( ), mi_save_set_get_next( ), mi_save_set_get_previous( ),** and **mi_save_set_insert( ).**

mi_save_set_get_next( )

The **mi_save_set_get_next( )** function traverses a save set in the forward direction.

Syntax

```
MI_ROW *mi_save_set_get_next(save_set, error)
   MI_SAVE_SET *save_set;
   mi_integer *error;
```

save_set	is a pointer to an **MI_SAVE_SET** structure.
error	is a pointer to a return code.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_save_set_get_next( )** function obtains the next row from the save set that *save_set* references. Because the DataBlade API maintains a save set as an FIFO (first-in, first-out) queue, the next row is the row added to the save set just *before* the current row; that is, it is the next row towards the bottom of the queue. This save set must have been previously created with **mi_save_set_create( ).**

When the **mi_save_set_get_next( )** function is successful, it returns a pointer to the row structure for the next row. It also moves the cursor position to point to the next row as the current row in the save set. The **mi_save_set_get_next( )** *function* is typically executed in a loop that terminates when no more rows remain to be fetched from the save-set cursor (the cursor position is at the end of the cursor). To indicate the "no more rows" condition, the function takes the following steps:

1. Returns the NULL-valued pointer
2. Sets the *error* argument to MI_NO_MORE_RESULTS

Return Values

An **MI_ROW** pointer	is a pointer to the next row forward in the specified save set.
NULL	indicates that the function was not successful or no more rows remain to be retrieved from the save-set cursor.

Upon failure, **mi_save_set_get_next( )** returns NULL and sets *error* to MI_ERROR.

Related Topics

See also the descriptions of **mi_save_set_get_first( ), mi_save_set_get_last( ),** and **mi_save_set_get_previous( ).**

mi_save_set_get_previous( )

The **mi_save_set_get_previous( )** function traverses a save set in the backward direction.

Syntax

```
MI_ROW *mi_save_set_get_previous(save_set, error)
   MI_SAVE_SET *save_set;
   mi_integer *error;
```

save_set	is a pointer to an **MI_SAVE_SET** structure.
error	is a pointer to a return code.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_save_set_get_previous( )** function obtains the previous row from the save set that *save_set* references. Because the DataBlade API maintains a save set as an FIFO (first-in, first-out) queue, the previous row is the row added to the save set just *after* the current row; that is, it is the next row towards the top of the queue. This save set must have been previously created with **mi_save_set_create( ).**

When the **mi_save_set_get_previous( )** function is successful, it returns a pointer to the row structure for the previous row. It also moves the cursor position to point to this new current row of the save set. The **mi_save_set_get_previous( )** *function* is typically executed in a loop that terminates when no more rows remain to be fetched from the save-set cursor. To indicate the "no more rows" condition, the function takes the following steps:

- Returns the NULL-valued pointer
- Sets the *error* argument to MI_NO_MORE_RESULTS

An error value of MI_NO_MORE_RESULTS indicates that you are at the beginning or the end of the save-set cursor or that the save set is empty.

Return Values

An **MI_ROW** pointer	is a pointer to the previous row backward in the specified save set.
NULL	indicates that the function was not successful, that no more rows remain to be retrieved from the save-set cursor, or that the save set is empty.

Upon failure, **mi_save_set_get_previous( )** returns NULL and sets *error* to MI_ERROR.

Related Topics

See also the descriptions of **mi_save_set_get_first( ), mi_save_set_get_last( ),** and **mi_save_set_get_next( ).**

mi_save_set_insert( )

The **mi_save_set_insert( )** function appends a copy of a row to the end of a save set.

Syntax

```
MI_ROW *mi_save_set_insert(save_set, row)
   MI_SAVE_SET *save_set;
   MI_ROW *row;
```

save_set	is a pointer to an **MI_SAVE_SET** structure.
row	is a pointer to a row that, typically, a call to **mi_next_row( )** fetches from the database server.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_save_set_insert( )** function inserts a row into the save set that *save_set* references. Because the DataBlade API maintains a save set as an FIFO (first-in, first-out) queue, **mi_save_set_insert( )** appends the new row to the end of the save set. This save set must have been previously created with **mi_save_set_create( ).**

Return Values

An **MI_ROW** pointer	is a pointer to the copy of *row* that is being appended to the save set. The application can use this pointer to access the row.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_next_row( )** and **mi_save_set_delete( ).**

mi_save_set_member( )

The **mi_save_set_member( )** function determines whether a row is a member of any open save set.

Syntax

```
mi_integer mi_save_set_member(row)
   MI_ROW *row;
```

row	is an MI_ROW pointer that represents a row of data.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_save_set_member( )** function determines whether the row that *row* references is a member of any open save set.

Return Values

0	indicates that the row is *not* a member of a save set.
1	indicates that the row is a member of a save set.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_save_set_count( ), mi_save_set_create( ), mi_save_set_get_first( ), mi_save_set_get_last( ), mi_save_set_get_next( ), mi_save_set_get_previous( ),** and **mi_save_set_insert( ).**

mi_server_connect( )

The **mi_server_connect( )** function establishes a connection between a client LIBMI application and a database server.

Syntax

```
MI_CONNECTION *mi_server_connect(conn_info)
   MI_CONNECTION_INFO *conn_info;
```

conn_info	is a pointer to a connection-information descriptor that identifies a database server.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	No

Usage

The **mi_server_connect( )** function establishes a client connection that connects the client LIBMI application to the database server identified by the connection-information structure that *conn_info* references. The establishment of a client connection begins a session. The **mi_server_connect( )** function obtains a connection descriptor for the client connection. This function is a constructor function for a connection descriptor. The new connection descriptor is valid until the end of the session.

Use this function for client LIBMI applications that run against different database servers.

The **mi_server_connect( )** function also initializes the DataBlade API when it is the first DataBlade API function in a client LIBMI application.

Return Values

An **MI_CONNECTION** pointer	is a pointer to the connection descriptor for the newly established connection.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_close( ), mi_open( ),** and **mi_server_reconnect( ).**

For a description of the connection-information descriptor or general information about how to establish a session with **mi_server_connect( )**, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_server_library_version( )

The **mi_server_library_version( )** function returns the name of the database server with its major and minor version numbers.

Syntax

```
mi_integer mi_server_library_version(buf, buflen, major, minor)
   mi_char1 *buf;
   mi_integer buflen;
   mi_integer *major;
   mi_integer *minor;
```

buf	is the user-allocated buffer for the version string.
buflen	is the length of *buf* in bytes.
major	is the address of the integer where the major version is returned.
minor	is the address of the integer where the minor version is returned.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
No	Yes

Usage

The **mi_server_library_version( )** function copies the database server name with its major and minor version numbers into the user-defined buffer that *buf* references. For example, for "IBM Informix Dynamic Server Version 10.00.UC1," the major version is 10 and the minor version is 00.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the description of **mi_version_comparison( )**.

mi_server_reconnect( )

The **mi_server_reconnect( )** function re-establishes a dropped connection.

Syntax

```
mi_integer mi_server_reconnect(conn)
   MI_CONNECTION *conn;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( )** or **mi_server_connect( ).**

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	No

Usage

The **mi_server_reconnect( )** re-establishes a connection to the database server and database that the connection descriptor, *conn*, identifies. Use **mi_server_reconnect( )** when the client connection is dropped but the connection descriptor still remains. For example, **a callback function** that handles the MI_LIB_DROPCONN client-library event **can use the mi_server_reconnect( )** function.

All resources in the session context that failed, such as memory and save sets, are preserved and do not need to be rebuilt. However, any transactions that were in progress when the connection failed are no longer valid, so it is the responsibility of the application to purge invalid rows out of save sets.

You cannot reconnect to a database server to which you are already connected.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_open( )** and **mi_server_connect( ).**

mi_set_connection_user_data( )

The **mi_set_connection_user_data( )** function associates the address of user data with an open connection.

Syntax

```
mi_integer mi_set_connection_user_data(conn, user_data_ptr)
   MI_CONNECTION *conn;
   void *user_data_ptr;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
	can be a pointer to a session-duration connection descriptor established by a previous call to **mi_get_session_connection( ).** Use of a session-duration connection descriptor is a *advanced* feature of the DataBlade API.
user_data_ptr	is a pointer to user data to associate with the specified connection.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_set_connection_user_data( )** function assigns the user-data pointer that *user_data_ptr* references to the connection descriptor that *conn* references. The user-data pointer is the address to a user-defined buffer or structure that contains private information you want to keep with the specified connection.

The DataBlade API does not interpret or touch the associated user-data pointer, other than to store it in the connection descriptor. Cast the *user_data_ptr* pointer to **void *** before you store it as user data in a connection descriptor.

You can obtain the user-data pointer from a connection descriptor with the **mi_get_connection_user_data( )** function.

Warning: Session-duration function descriptors and named memory are advanced features of the DataBlade API. They can adversely affect your UDR if you use them incorrectly. Use them only when a regular function descriptor cannot perform the task you need done. For more information, see the *IBM Informix DataBlade API Programmer's Guide*.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the description of **mi_get_connection_user_data( ).**

mi_set_default_connection_info( )

The **mi_set_default_connection_info( )** function sets the default connection parameters with values from a connection-information descriptor that the user provides.

Syntax

```
mi_integer mi_set_default_connection_info(conn_info)
   MI_CONNECTION_INFO *conn_info;
```

conn_info	is a pointer to a user-provided connection-information descriptor, which sets the default connection parameters.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Ignored

Usage

Client Only

The **mi_set_default_connection_info( )** function sets default connection parameters from the user-defined values in the connection-information descriptor (**MI_CONNECTION_INFO** structure) that *conn_info* references. The connection parameters include the name of the database server and a GLS locale.

Tip: You must allocate this connection-information descriptor *before* you call **mi_set_default_connection_info( )**.

After this function sets the default connection parameters, you can pass the *conn_info* descriptor to **mi_server_connect( )** to specify the default connection parameters for a connection.

If you do not want to change a particular default value, initialize string fields to a NULL-valued pointer and integer fields to zero (0). To use the default database server, initialize the **server_name** field of the connection-information descriptor to a NULL-valued pointer. To specify a new default database server, specify a null-terminated string for the **server_name** field.

This function returns MI_ERROR if **mi_sysname( )** failed when it attempted to set the database server name. If the client LIBMI application has not registered a callback function to handle the MI_LIB_BADSERV error, it must check the return status of **mi_set_default_connection_info( )**.

End of Client Only

When **mi_set_default_connection_info( )** is the first DataBlade API function in a client LIBMI application or a user-defined routine, it initializes the DataBlade API.

You can obtain the current values of the default connection parameters with the **mi_get_default_connection_info( )** function.

Server Only

The **mi_set_default_connection_info( )** function is ignored when it is used as a user-defined routine.

Client Only

Global Language Support

In a client LIBMI application, the GLS locale in the default connection parameters refers to the database locale. For more information about GLS locales, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

End of Client Only

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_get_connection_info( ), mi_get_default_connection_info( ), mi_server_connect( ),** and **mi_sysname( ).**

For a description of the connection-information descriptor or more information on how to use the connection-information descriptor, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_set_default_database_info( )

The **mi_set_default_database_info( )** function sets the default database parameters with values from a database-information descriptor that the user provides.

Syntax

```
mi_integer mi_set_default_database_info(db_info)
   MI_DATABASE_INFO *db_info;
```

db_info	is a pointer to a user-provided database-information descriptor, which sets the default database parameters.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Ignored

Usage

Client Only

The **mi_set_default_database_info( )** function sets default database parameters with user-defined values from the database-information descriptor (**MI_DATABASE_INFO** structure) that *db_info* references. The database parameters include the name of the database, the user account, and the account password. The **mi_open( )** function can use these database parameters when it establishes a connection.

Tip: You must allocate this database-information descriptor *before* you call **mi_set_default_database_info( )**.

If you do not want to change a particular default value, set the string fields to a NULL-valued pointer and the integer fields to `0`.

End of Client Only

Server Only

In a user-defined routine, the **mi_set_default_database_info( )** function is ignored.

End of Server Only

The **mi_set_default_database_info( )** function initializes the DataBlade API when it is the first DataBlade API function in a client LIBMI application or a UDR.

You can obtain the current database parameters with the **mi_get_default_database_info( )** function.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_get_database_info( ), mi_get_default_database_info( ), mi_open( ),** and **mi_sysname( ).**

For more information on the database-information descriptor or more information on how to use the database-information descriptor, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_set_large( )

The **mi_set_large( )** macro sets the threshold-tracking field of a multirepresentational opaque type to indicate that the multirepresentational data is stored in a smart large object.

Syntax

```
void mi_set_large(size)
   MI_MULTIREP_SIZE size;
```

size	is the value of the threshold-tracking field in the internal representation of a multirepresentational opaque type. This function sets the field to MI_MULTIREP_LARGE.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

The **mi_set_large( )** macro assigns the MI_MULTIREP_LARGE constant to *size*, which is the threshold-tracking field in a multirepresentational opaque type. This macro is useful to indicate that multirepresentational data is stored in a smart large object. If the data is not stored in a smart large object, the threshold-tracking field should contain the MI_MULTIREP_SMALL constant.

Return Values

None.

Related Topics

See also the description of **mi_issmall_data( ).**

mi_set_parameter_info( )

The **mi_set_parameter_info( )** function sets the session parameters with values from a parameter-information descriptor that the user provides.

Syntax

```
mi_integer mi_set_parameter_info(sess_info)
   const MI_PARAMETER_INFO *sess_info;;
```

sess_info	is a pointer to a user-provided parameter-information descriptor from which to set the current session parameters.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_set_parameter_info( )** function sets the session parameters with the fields in the parameter-information descriptor (**MI_PARAMETER_INFO** structure) that *sess_info* references. The parameter-information descriptor determines whether callbacks are enabled and whether pointers are checked during the session. You must allocate this parameter-information descriptor *before* you call **mi_set_parameter_info( )**.

The **mi_set_parameter_info( )** function also initializes the DataBlade API when it is the first DataBlade API function in a client LIBMI application or a user-defined routine.

To obtain the session parameters, use the **mi_get_parameter_info( )** function.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the description of **mi_get_parameter_info( ).**

For more information on the parameter-information descriptor, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_set_vardata( )

The **mi_set_vardata( )** accessor routine stores data in the data portion of the data in a varying-length structure (such as **mi_lvarchar**).

Syntax

```
void mi_set_vardata(varlen_ptr, data_ptr)
   mi_lvarchar *varlen_ptr;
   char *data_ptr;
```

varlen_ptr	is a pointer to the varying-length structure.
data_ptr	is a pointer to the data to insert.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_set_vardata( )** function copies the data that *data_ptr* references to the data field of the varying-length structure that *varlen_ptr* references. The function determines the number of bytes to copy from the data length information stored in the *varlen_ptr* varying-length structure. Therefore, use **mi_set_varlen( )** to set the length of the data *before* you copy in the data.

Important: The varying-length structure that "*varlen_ptr*" references is an opaque structure. Do not access fields of this structure directly. Instead, use **mi_set_vardata( )** or **mi_set_vardata_align( )** to store the data in this structure.

The data in a varying-length structure is *not* null terminated. Do *not* copy the null terminator into the data portion of a varying-length structure.

Although the *varlen_ptr* argument is declared as a pointer to an **mi_lvarchar** value, you can also use the **mi_set_vardata( )** function to set data in other varying-length data types, such as **mi_sendrecv**.

Return Values

None.

Related Topics

See also the descriptions of **mi_get_vardata( ), mi_new_var( ), mi_set_vardata_align( ), mi_set_varlen( ), mi_set_varptr( ),** and **mi_var_free( ).**

mi_set_vardata_align( )

The **mi_set_vardata_align( )** accessor routine stores data in the data portion of a varying-length structure (such as **mi_lvarchar**) and adjusts for any initial padding required to align the data.

Syntax

```
void mi_set_vardata_align(varlen_ptr, data_ptr, align)
   mi_lvarchar *varlen_ptr;
   char *data_ptr;
   mi_integer align;
```

varlen_ptr	is a pointer to the varying-length structure.
data_ptr	is a pointer to the data to store in the data portion of the varying-length structure.
align	is the nearest *align*-byte boundary on which to align the data.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_set_vardata_align( )** function copies the data that *data_ptr* references to the data field of the varying-length structure that *varlen_ptr* references. The function determines the number of bytes to copy from the data length information stored in the *varlen_ptr* varying-length structure. Therefore, use **mi_set_varlen( )** to set the length of the data *before* you copy in the data.

Important: The varying-length structure that "*varlen_ptr*" references is an opaque structure. Do not access fields of this structure directly. Instead, use **mi_set_vardata_align( )** to store aligned data in this structure.

The **mi_set_vardata_align( )** function aligns the data on the nearest *align*-byte boundary. The function is useful for data types whose alignment is more stringent than the 4-byte alignment that the varying-length structure guarantees. For example, on some computer architectures, double-precision values might need to be stored on 64-bit boundaries. For opaque data types, this value must match the **align** column of the **sysxtdtypes** system catalog table.

The data in a varying-length structure is *not* null terminated. Do not copy the null terminator into the data portion of a varying-length structure.

Although the *varlen_ptr* argument is declared as a pointer to an **mi_lvarchar** value, you can also use the **mi_set_vardata_align( )** function to set aligned data in other varying-length data types, such as **mi_sendrecv**.

Return Values

None.

Related Topics

See also the descriptions of **mi_get_vardata_align( ), mi_new_var( ), mi_set_vardata( ), mi_set_varlen( ), mi_set_varptr( ),** and **mi_var_free( ).**

mi_set_varlen( )

The **mi_set_varlen( )** accessor routine sets the length of the data in a varying-length structure (such as **mi_lvarchar**).

Syntax

```
void mi_set_varlen(varlen_ptr, data_len)
   mi_lvarchar *varlen_ptr;
   mi_integer data_len;
```

varlen_ptr	is a pointer to the varying-length structure.
data_len	is the length of the data to store in the varying-length descriptor.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_set_varlen( )** function sets the data length to *data_len* in the varying-length structure that *varlen_ptr* references. Uses for **mi_set_varlen( )** include:

- Before a call to the **mi_set_vardata( )** or **mi_set_vardata_align( )** function to tell these functions how many bytes of data to store
- When you set your own data portion with **mi_set_varptr( )** to set the data length in a user-allocated data portion

Important: The varying-length structure that *"varlen_ptr"* references is an opaque structure. Do not access fields of this structure directly. Instead, use **mi_set_vardata( )** to set the data length in this structure.

The **mi_set_varlen( )** function sets the data length in the varying-length descriptor. If you specify a data length larger than the current data portion, this function does *not* reallocate the data portion. Therefore, when you change the data length with **mi_set_varlen( )**, make sure that the new value does not exceed the size of the data portion.

The data in a varying-length structure is *not* null terminated. Make sure that you use **mi_set_varlen( )** to set the length of the actual varying-length data, not including any null terminator.

Although the *varlen_ptr* argument is declared as a pointer to an **mi_lvarchar** value, you can also use the **mi_set_varlen( )** function to set data length in other varying-length data types, such as **mi_sendrecv**.

Return Values

None.

Related Topics

See also the descriptions of **mi_get_varlen( ), mi_new_var( ), mi_set_vardata( ), mi_set_vardata_align( ), mi_set_varptr( ),** and **mi_var_free( ).**

For information on when to set the length of a varying-length structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_set_varptr( )

The **mi_set_varptr( )** function sets the data pointer in a varying-length structure (such as **mi_lvarchar**).

Syntax

```
void mi_set_varptr(varlen_ptr, data_ptr)
   mi_lvarchar *varlen_ptr;
   char *data_ptr;
```

varlen_ptr	is a pointer to the varying-length structure.
data_ptr	is a pointer to the data.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_set_varptr( )** function sets the data pointer in the varying-length structure that *varlen_ptr* references to the buffer that *data_ptr* references. Use this function to set the data pointer in a varying-length structure to point to a data portion that you allocate.

Server Only

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

End of Server Only

Important: The varying-length structure that "*varlen_ptr*" references is an opaque structure. Do not access fields of this structure directly. Instead, use **mi_set_vardata( )** to set the data pointer in this structure.

Although the *varlen_ptr* argument is declared as a pointer to an **mi_lvarchar** value, you can also use the **mi_set_varptr( )** function to set the data pointer in other varying-length data types, such as **mi_sendrecv**.

Return Values

None.

Related Topics

See also the descriptions of **mi_new_var( ), mi_set_vardata( ), mi_set_vardata_align( ), mi_set_varptr( ),** and **mi_var_free( ).**

For information on how to set the data pointer of a varying-length structure, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_stack_limit( )

The **mi_stack_limit( )** function checks if the specified amount of space is available on the user stack.

Syntax

```
mi_smallint mi_stack_limit(mi_integer size);
```

size	is the stack size, in bytes.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_stack_limit( )** function checks if the space available on the user stack is greater than *size* plus the stack margin. The user stack checked belongs to the current thread executing on the CPU VP.

Return Values

0	indicates that there is sufficient amount of available space on the user stack.
-1	indicates that *size* exceeds available space on the user stack.

mi_statement_command_name( )

The **mi_statement_command_name( )** function returns the name of an SQL prepared statement.

Syntax

```
mi_string *mi_statement_command_name(stmt_desc)
   MI_STATEMENT *stmt_desc;
```

stmt_desc	is a statement descriptor that **mi_prepare( )** returned.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_statement_command_name( )** function returns the SQL statement name in the prepared statement that *stmt_desc* references. The function returns the name as a null-terminated string in memory that it allocated in the current memory duration. This statement name is only the verb of the statement, *not* the entire statement syntax.

For example, suppose the **mi_prepare( )** statement had prepared the following SELECT statement:

```
SELECT * FROM customer WHERE state = "CA";
```

The **mi_statement_command_name( )** returns only the verb of this statement: *select*.

Important: Use the **mi_statement_command_name( )** function only for prepared statements. To obtain the name of the current statement, use the **mi_result_command_name( )** function after the **mi_get_result( )** function returns *MI_DML* or *MI_DDL*. To obtain the name of the SQL statement that has invoked a C UDR, use the **mi_current_command_name( )** function.

The function returns the statement verb as a null-terminated string in a buffer that it allocates.

Return Values

An **mi_string** pointer	is a pointer to the verb of the last statement or command.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_current_command_name( ), mi_get_result( ), mi_prepare( ),** and **mi_result_command_name( ).**

mi_stream_clear_eof( )

The **mi_stream_clear_eof( )** function clears the end-of-file marker for a stream.

Syntax

```
mi_integer mi_stream_clear_eof(strm_desc)
     MI_STREAM *strm_desc;
```

strm_desc	is a pointer to a stream descriptor for an open stream.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
No	Yes

Return Values

MI_TRUE	indicates that the end-of-file marker has been cleared.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references is invalid.

Related Topics

See also the description of **mi_stream_set_eof( ).**

mi_stream_close( )

The **mi_stream_close( )** function closes a stream.

Syntax

```
mi_integer mi_stream_close(strm_desc)
   MI_STREAM *strm_desc;
```

strm_desc	is a pointer to a stream descriptor for an open stream.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_stream_close( )** function closes the stream that *strm_desc* identifies. To close the stream, **mi_stream_close( )** uses the stream-close function that the stream-operations structure contains.

When it closes a stream, **mi_stream_close( )** also deallocates the **MI_STREAM** structure, unless the **MI_STREAM** structure was passed as an argument to **mi_stream_init( ).** Calling **mi_stream_close( )** to close a stream of a predefined stream class always deallocates the **MI_STREAM** structure; **mi_stream_close( )** is the destructor function for a stream descriptor.

Important: The **mi_stream_close( )** function can free a stream descriptor only if **mi_stream_init( )** allocated it. If your user-defined stream-open function has allocated its own stream descriptor, **mi_stream_close( )** does not free this descriptor.

Return Values

MI_OK	indicates that the stream has been closed.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references is invalid.
MI_STREAM_ENIMPL	indicates that the stream class does not implement the stream-close function.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_open_fio( ), mi_stream_open_mi_lvarchar( ),** and **mi_stream_open_str( ).**

mi_stream_eof( )

The **mi_stream_eof( )** function determines whether the current stream seek position is at the end of the stream.

Syntax

```
mi_integer mi_stream_eof(strm_desc)
   MI_STREAM *strm_desc;
```

strm_desc	is a pointer to a stream descriptor for an open stream.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_stream_eof( )** function determines if the seek position of the stream that *strm_desc* references is currently at the end of the stream. You can use this function on any stream. The function does not have to be implemented for a user-defined stream.

Return Values

MI_TRUE	indicates that the end of the stream has been reached.
MI_FALSE	indicates that the end of the stream has *not* been reached.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references is invalid.

Related Topics

See also the descriptions of **mi_stream_open_fio( ), mi_stream_open_mi_lvarchar( ), mi_stream_open_str( ), mi_stream_read( ),** and **mi_stream_tell( ).**

mi_stream_get_error( )

The **mi_stream_get_error( )** function returns the last error status for an open stream and clears the error condition.

Syntax

```
mi_integer mi_stream_get_error(strm_desc)
   MI_STREAM *strm_desc;
```

strm_desc	is a pointer to a stream descriptor for an open stream.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_stream_get_error( )** function obtains the last error status of the stream that *strm_desc* references and then clears the error condition.

Return Values

A valid stream error code	is the last error that occurred on the stream. For information about stream error codes, see the *IBM Informix DataBlade API Programmer's Guide*.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references is invalid.

Related Topics

See also the descriptions of **mi_stream_eof( ), mi_stream_init( ),** and **mi_stream_set_error( ).**

mi_stream_getpos( )

The **mi_stream_getpos( )** function returns the current seek position of an open stream.

Syntax

```
mi_integer mi_stream_getpos(strm_desc, seek_pos)
   MI_STREAM *strm_desc;
   mi_int8 *seek_pos;
```

strm_desc	is a pointer to a stream descriptor for an open stream.
seek_pos	is a pointer to an eight-byte integer (**mi_int8**).

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_stream_getpos( )** function obtains the current seek position of the stream that *strm_desc* references. The stream seek position is the offset for the next read or write operation on the stream. The **mi_stream_getpos( )** function returns this seek position in the **mi_int8** variable that *seek_pos* references.

Return Values

MI_OK	indicates that the *seek_pos* variable contains the current stream seek position, measured in the number of bytes from the beginning of the stream.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the *seek_pos* value is invalid.
MI_STREAM_ENIMPL	indicates that the stream class does not implement the seek position.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_open_fio( ), mi_stream_open_mi_lvarchar( ), mi_stream_open_str( ), mi_stream_read( ), mi_stream_seek( ), mi_stream_setpos( ), mi_stream_tell( ),** and **mi_stream_write( ).**

mi_stream_init( )

The **mi_stream_init( )** function initializes a user-defined stream.

Syntax

```
MI_STREAM *mi_stream_init(strm_ops, strm_data, strm_desc)
   struct stream_operations *strm_ops;
   void *strm_data;
   MI_STREAM *strm_desc;
```

strm_ops	is the structure that holds the function pointers for the stream I/O functions.
strm_data	is a pointer to the stream data.
strm_desc	is a pointer to a stream descriptor. This **MI_STREAM** pointer can be a NULL-valued pointer if the stream has not been previously allocated.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_stream_init( )** function returns a pointer to the initialized stream, based on the stream I/O functions in the stream-operations structure that *strm_ops* references, and data, which *strm_data* references. Use this function to implement a user-defined stream class. Do not call **mi_stream_init( )** on a predefined stream.

The third 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 implementor.

To initialize a new stream, the function takes the following steps:

1. Determines whether to allocate a new stream descriptor
   - If the *strm_desc* argument is a NULL-valued pointer, **mi_stream_init( )** allocates a new stream.
   - If the *strm_desc* argument references a valid stream, **mi_stream_init( )** does *not* allocate the stream again.
2. If the *strm_desc* argument points to valid memory, uses the passed memory for the **MI_STREAM** structure descriptor

   The function does not allocate the stream structure again.
3. Returns a pointer to the new stream descriptor

Return Values

An **MI_STREAM** pointer	is a pointer to a newly allocated stream.
NULL	indicates that the function was not successful.

Related Topics

See also the description of **mi_stream_close( ).**

mi_stream_length( )

The **mi_stream_length( )** function obtains the length of an open stream.

Syntax

```
mi_integer mi_stream_length(strm_desc, len)
   MI_STREAM *strm_desc;
   mi_int8 *len;
```

strm_desc	is a pointer to a stream descriptor for an open stream.
len	is a pointer to an eight-byte integer (**mi_int8**).

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_stream_length( )** function gets the length the data in the stream that *strm_desc* references. The function returns the length of the stream data in the *len* parameter.

Return Values

MI_OK	indicates that the function was successful.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references is invalid or that the specified length is less than zero.
MI_STREAM_ENIMPL	indicates that the stream class does not implement the stream-length function.
MI_ERROR	indicates that the function was not successful.
MI_STREAM_EFAIL	indicates that the function was not successful for a file stream.

Related Topics

See also the descriptions of **mi_stream_open_fio( ), mi_stream_open_mi_lvarchar( ), mi_stream_open_str( ), mi_stream_seek( ), mi_stream_tell( ),** and **mi_stream_write( ).**

mi_stream_open_fio( )

The **mi_stream_open_fio( )** function opens a new stream on an operating-system file.

Syntax

```
MI_STREAM *mi_stream_open_fio(strm_desc, filename, open_flags, open_mode)
   MI_STREAM *strm_desc;
   char *filename;
   mi_integer open_flags;
   mi_integer open_mode;
```

strm_desc	is a pointer to a stream descriptor.
filename	is the pathname of an operating-system file.
open_flags	is an integer bit mask that can be any of the following open flags:

open flags that the operating-system open command supports: UNIX or Linux **open(2)** or Windows **_open**.

MI_O_SERVER_FILE (*default*)	indicates that the file to open is on the server computer.
MI_O_CLIENT_FILE	indicates that the file to open is on the client computer.

open_mode	is the file-permission mode in a format that the operating-system open command supports: UNIX or Linux **open(2)** or Windows **_open**.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_stream_open_fio( )** function initializes and opens a data stream on the operating-system file that *filename* specifies.

The **mi_stream_open_fio( )** function returns a stream descriptor that identifies the file stream. This function is a constructor function for a stream descriptor and allocates the new stream descriptor in the current memory duration.

Return Values

An **MI_STREAM** pointer	is a pointer to the newly opened stream on the specified file.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_close( ), mi_stream_getpos( ), mi_stream_length( ), mi_stream_read( ), mi_stream_seek( ), mi_stream_setpos( ), mi_stream_tell( ),** and **mi_stream_write( ).**

mi_stream_open_mi_lvarchar( )

The **mi_stream_open_mi_lvarchar( )** function opens a new stream on varying-length data.

Syntax

```
MI_STREAM *mi_stream_open_mi_lvarchar(strm_desc, varlen_ptr)
   MI_STREAM *strm_desc;
   mi_lvarchar *varlen_ptr;
```

strm_desc	is a pointer to a stream descriptor.
varlen_ptr	is a pointer to a structure that contains varying-length data.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_stream_open_mi_lvarchar( )** initializes and opens a data stream on the varying-length structure that *varlen_ptr* references. To open a varying-length-data stream, **mi_stream_open_mi_lvarchar( )** opens the stream on the data it is passed. Once it is opened, a varying-length-data stream has the same behavior as a string stream.

The **mi_stream_open_mi_lvarchar( )** function returns a stream descriptor that identifies the varying-length-data stream. This function is a constructor function for a stream descriptor. It allocates the new stream descriptor in the current memory duration.

Return Values

An **MI_STREAM** pointer	is a pointer to the newly opened stream on the specified varying-length data.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_close( ), mi_stream_getpos( ), mi_stream_length( ), mi_stream_open_str( ), mi_stream_read( ), mi_stream_seek( ), mi_stream_setpos( ), mi_stream_tell( ),** and **mi_stream_write( ).**

mi_stream_open_str( )

The **mi_stream_open_str( )** function opens a new stream on a character string.

Syntax

```
mi_integer mi_stream_open_str(strm_desc, str_ptr, str_len)
   MI_STREAM *strm_desc;
   char *str_ptr;
   mi_integer str_len;
```

strm_desc	is a pointer to a stream descriptor.
str_ptr	is a pointer to a character string.
str_len	is the length of the character string that *str_ptr* references.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_stream_open_str( )** function initializes and opens a data stream on the character string that *str_ptr* references. This function returns a stream descriptor that identifies the string stream. This function is a constructor function for a stream descriptor. It allocates the new stream descriptor in the current memory duration.

Tip: The stream operates on a copy of the passed string, so changes to the string are not reflected in the stream, and changes to the stream are not reflected in the string.

Return Values

An **MI_STREAM** pointer	is a pointer to the newly opened stream on the specified string data.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_close( ), mi_stream_getpos( ), mi_stream_length( ), mi_stream_open_mi_lvarchar( ), mi_stream_read( ), mi_stream_seek( ), mi_stream_setpos( ), mi_stream_tell( ),** and **mi_stream_write( ).**

mi_stream_read( )

The **mi_stream_read( )** function reads a specified number of bytes from a stream into a buffer.

Syntax

```
mi_integer mi_stream_read(strm_desc, buf, nbytes)
   MI_STREAM *strm_desc;
   void *buf;
   mi_integer nbytes;
```

strm_desc	is a pointer to a stream descriptor for an open stream.
buf	is a pointer to a user-allocated buffer.
nbytes	is the maximum number of bytes to read into the *buf* buffer.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_stream_read( )** function reads up to *nbytes* bytes from the open stream that *strm_desc* references. The function copies this data into the *buf* user-defined buffer. The read operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

To read to the end of the stream, call **mi_stream_read( )** in a loop until it returns the MI_STREAM_EEOF constant, as follows:

- For all calls except the last call, **mi_stream_read( )** reads *nbytes* or fewer bytes and returns the number of bytes it has read.
- For the last call, **mi_stream_read( )** returns the MI_STREAM_EEOF constant to indicate that it has reached the end of the stream without any errors.

Return Values

>=0	is the actual number of bytes that the function has read from the open stream to the *buf* buffer. The actual number of bytes read might be less than the *nbytes* value.
MI_STREAM_EEOF	indicates that the end of the stream has been reached without any errors.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references is invalid, the *buf* user-defined buffer is invalid, or the specified *nbytes* value is less than zero.
MI_STREAM_ENIMPL	indicates that the stream class does not implement the stream-read function.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_open_fio( ), mi_stream_open_mi_lvarchar( ), mi_stream_open_str( ), mi_stream_seek( ), mi_stream_tell( ),** and **mi_stream_write( ).**

mi_stream_seek( )

The **mi_stream_seek( )** function sets the stream seek position for the next read or write operation on an open stream.

Syntax

```
mi_integer mi_stream_seek(strm_desc, offset, whence)
   MI_STREAM *strm_desc;
   mi_int8 *offset;
   mi_integer whence;
```

strm_desc	is a pointer to a stream descriptor for an open stream.
offset	is a pointer to a positive or negative byte offset from the *whence* seek position.
whence	is a constant that specifies the position from which to start the seek operation to the *offset* position.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Important: Enterprise Replication does not support this function.

Usage

The **mi_stream_seek( )** function uses the *whence* and *offset* arguments to determine the new seek position of the stream that *strm_desc* references, as follows:

- The *whence* argument identifies the position from which to start the seek operation.

  Valid values include the following whence constants.

Whence Constant	Starting Position for Seek Operation
MI_LO_SEEK_SET	The start of the stream
MI_LO_SEEK_CUR	The current seek position of the stream
MI_LO_SEEK_END	The end of the stream

  – The *offset* argument identifies the offset, in bytes, from the starting position (which the *whence* argument specifies) at which to begin the seek operation.

    This *offset* value can be negative for all values of *whence*. For more information on how to access eight-bit (INT8) integers, see the *IBM Informix DataBlade API Programmer's Guide*.

Use the associated stream-open function to obtain the stream descriptor for one of these data streams. You can then pass this stream descriptor to **mi_stream_seek( )** to set the seek position on any of these streams. You can also implement an **mi_stream_seek( )** function for your own user-defined stream. For more information, see the *IBM Informix DataBlade API Programmer's Guide*.

On success, **mi_stream_seek( )** returns MI_OK. To obtain the new stream seek position, use **mi_stream_tell( ).**

Return Values

MI_OK	indicates that the new stream seek position has been set to the combination of *whence* and *offset*.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the *offset* value is invalid, or that the specified seek position is past the end of the data.
MI_STREAM_ENIMPL	indicates that the stream class does not implement the seek position.
MI_ERROR	indicates that the function was not successful.
MI_STREAM_EWHENCE	indicates that the *whence* value is invalid.

Related Topics

See also the descriptions of **mi_stream_open_fio( ), mi_stream_open_mi_lvarchar( ), mi_stream_open_str( ), mi_stream_read( ), mi_stream_setpos( ), mi_stream_tell( ),** and **mi_stream_write( ).**

mi_stream_set_eof( )

The **mi_stream_set_eof( )** function sets the end-of-file marker for a stream.

Syntax

```
mi_integer mi_stream_set_eof(strm_desc)
     MI_STREAM *strm_desc;
```

strm_desc	is a pointer to a stream descriptor for an open stream.

Return Values

MI_TRUE	indicates that the end-of-file marker has been set.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* reference is invalid.

Related Topics

See also the description of **mi_stream_clear_eof( ).**

mi_stream_set_error( )

The **mi_stream_set_error( )** function sets the last error status for an open stream.

Syntax

```
mi_integer mi_stream_set_error(strm_desc, errno)
   MI_STREAM *strm_desc;
   mi_integer errno;
```

strm_desc	is a pointer to a stream descriptor for an open stream.
errno	is a valid stream error code.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_stream_set_error( )** function sets the last error code of the stream that *strm_desc* references. For information about stream error codes, see the *IBM Informix DataBlade API Programmer's Guide*.

Return Values

MI_TRUE	indicates that the function was successful.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references is invalid.

Related Topics

See also the descriptions of **mi_stream_eof( ), mi_stream_get_error( ),** and **mi_stream_init( ).**

mi_stream_setpos( )

The **mi_stream_setpos( )** function sets the stream seek position for the next read or write operation on an open stream.

Syntax

```
mi_integer mi_stream_setpos(strm_desc, seek_pos)
   MI_STREAM *strm_desc;
   mi_int8 *seek_pos;
```

strm_desc	is a pointer to a stream descriptor for an open stream.
seek_pos	is a pointer to the seek position of the stream that the *strm_desc* stream descriptor references.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Important: Enterprise Replication does not support this function.

Usage

The **mi_stream_setpos( )** function uses the *seek_pos* argument to determine the new seek position of the stream that *strm_desc* references. This stream seek position is the offset for the next read or write operation on the stream.

Return Values

MI_OK	indicates that the new stream seek position has been set to *seek_pos*.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the *seek_pos* value is invalid, or that the specified seek position is past the end of the data.
MI_STREAM_ENIMPL	indicates that the stream class does not implement the seek position.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_open_fio( ), mi_stream_open_mi_lvarchar( ), mi_stream_open_str( ), mi_stream_read( ), mi_stream_seek( ), mi_stream_tell( ),** and **mi_stream_write( ).**

mi_stream_tell( )

The **mi_stream_tell( )** function returns the current seek position of an open stream, relative to the beginning of the stream.

Syntax

```
mi_int8 *mi_stream_tell(strm_desc)
   MI_STREAM *strm_desc;
```

strm_desc	is a pointer to a stream descriptor for an open stream.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_stream_tell( )** function obtains the current seek position of the stream that *strm_desc* references. The stream seek position is the offset for the next read or write operation on the stream. The **mi_stream_tell( )** function returns this seek position as a pointer to an **mi_int8** value.

Return Values

An **mi_int8** pointer	is a pointer to the stream seek position, measured in the number of bytes from the beginning of the stream.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_open_fio( ), mi_stream_open_mi_lvarchar( ), mi_stream_open_str( ), mi_stream_read( ), mi_stream_seek( ), mi_stream_setpos( ),** and **mi_stream_write( ).**

mi_stream_write( )

The **mi_stream_write( )** function writes a specified number of bytes to an open stream.

Syntax

```
mi_integer mi_stream_write(strm_desc, buf, nbytes)
   MI_STREAM *strm_desc;
   void *buf;
   mi_integer nbytes;
```

strm_desc	is a pointer to a stream descriptor for an open stream.
buf	is a pointer to a user-allocated buffer, of at least *nbytes* bytes, that contains the data to write to the stream.
nbytes	is the maximum number of bytes to write to the stream.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_stream_write( )** function writes up to *nbytes* bytes to the open stream that *strm_desc* references. This functions copies the data to write to the stream from the *buf* user-defined buffer. The write operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

Return Values

>=0	is the actual number of bytes that the function has written from the *buf* buffer to the open stream. The actual number of bytes written might be less than the *nbytes* value.
MI_STREAM_EEOF	indicates that the end of the stream has been reached.
MI_STREAM_EBADARG	indicates that the stream structure that *strm_desc* references is invalid, the user-defined buffer is invalid, or the specified number of bytes is less than zero.
MI_STREAM_ENIMPL	indicates that the stream class does not implement the stream-write function.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_open_fio( ), mi_stream_open_mi_lvarchar( ), mi_stream_open_str( ), mi_stream_read( ), mi_stream_seek( ),** and **mi_stream_tell( ).**

mi_streamread_boolean( )

The **mi_streamread_boolean( )** function reads an **mi_boolean** (BOOLEAN) value from a stream, converting any difference in the stream representation to that of the internal representation.

Syntax

```
mi_integer mi_streamread_boolean(strm_desc, bool_ptr)
   MI_STREAM *strm_desc;
   mi_boolean *bool_ptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream from which to read the **mi_boolean** value.
bool_ptr	is a pointer to the buffer into which to copy the **mi_boolean** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamread_boolean( )** function reads an **mi_boolean** value from the stream that *strm_desc* references. The function reads this value into the **mi_boolean** buffer that *bool_ptr* references. The read operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

The **mi_streamread_boolean( )** function is useful in a **streamread( )** support function of an opaque data type that contains an **mi_boolean** value.

Return Values

>=0	is the actual number of bytes that the function has read from the open stream to the value that *bool_ptr* references.
MI_STREAM_EEOF	indicates that the end of the stream has been reached without any errors.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the value that *bool_ptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamwrite_boolean( ).**

For more information on the use of **mi_streamread_boolean( )** in a **streamread( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamread_collection( )

The **mi_streamread_collection( )** function reads a collection structure (LIST, SET, MULTISET) from a stream, converting any difference in the stream representation to that of the internal representation.

Syntax

```
mi_integer mi_streamread_collection(strm_desc, coll_dptr)
   MI_STREAM *strm_desc;
   MI_COLLECTION **coll_ptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream from which to read the collection structure (**MI_COLLECTION**).
coll_dptr	is a pointer to the buffer into which to copy the address of the collection structure.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Important: Enterprise Replication does not support this function.

Usage

The **mi_streamread_collection( )** function reads a collection from the stream that *strm_desc* references. The function reads the collection structure from the stream and puts its address in the buffer that *coll_dptr* references. The read operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

The **mi_streamread_collection( )** function is a constructor function for a collection structure. It allocates memory for the collection structure in the current memory duration.

This function is useful in a **streamread( )** support function of an opaque data type that contains a collection structure (**MI_COLLECTION**).

Important: The **mi_streamread_collection( )** function requires the caller to have an open connection to the database server.

Return Values

>=0	is the actual number of bytes that the function has read from the open stream to the value that *coll_dptr* references.
MI_STREAM_EEOF	indicates that the end of the stream has been reached without any errors.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the value that *coll_dptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamwrite_collection( ).**

For more information on the use of **mi_streamread_collection( )** in a **streamread( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamread_date( )

The **mi_streamread_date( )** function reads an **mi_date** (DATE) value from a stream, converting any difference in the stream representation to that of the internal representation.

Syntax

```
mi_integer mi_streamread_date(strm_desc, date_ptr)
   MI_STREAM *strm_desc;
   mi_date *date_ptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream from which to read the **mi_date** value.
date_ptr	is a pointer to the buffer into which to copy the **mi_date** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamread_date( )** function reads an **mi_date** value from the stream that *strm_desc* references. The function reads this value into the **mi_date** buffer that *date_ptr* references. The read operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

The **mi_streamread_date( )** function is useful in a **streamread( )** support function of an opaque data type that contains an **mi_date** value.

Return Values

>=0	is the actual number of bytes that the function has read from the open stream to the value that *date_ptr* references.
MI_STREAM_EEOF	indicates that the end of the stream has been reached without any errors.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the value that *date_ptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamwrite_date( ).**

For more information on the use of **mi_streamread_date( )** in a **streamread( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamread_datetime( )

The **mi_streamread_datetime( )** function reads an **mi_datetime** (DATETIME) value from a stream, converting any difference in the stream representation to that of the internal representation.

Syntax

```
mi_integer mi_streamread_datetime(strm_desc, dtime_dptr)
   MI_STREAM *strm_desc;
   mi_datetime **dtime_dptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream from which to read the **mi_datetime** value.
dtime_dptr	is a pointer to the buffer into which to copy the address of the **mi_datetime** value.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
No	Yes

Usage

The **mi_streamread_datetime( )** function reads an **mi_datetime** value from the stream that *strm_desc* references. The function reads the **mi_datetime** value from the stream and puts the address of the value in the buffer that *dtime_dptr* references. The read operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

When **dtime_dptr* points to NULL, **mi_streamread_datetime( )** allocates the memory for the buffer in the current memory duration. Otherwise, the function assumes that you have allocated the memory that **dtime_dptr* references.

Important: Be sure that **dtime_dptr* points to NULL if the parameter does not point to valid memory.

The **mi_streamread_datetime( )** function is useful in a **streamread( )** support function of an opaque data type that contains an **mi_datetime** value.

Return Values

>=0	is the actual number of bytes that the function has read from the open stream to the value that *dtime_dptr* references.
MI_STREAM_EEOF	indicates that the end of the stream has been reached without any errors.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the value that *dtime_dptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamwrite_datetime( ).**

For more information on the use of **mi_streamread_datetime( )** in a **streamread( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamread_decimal( )

The **mi_streamread_decimal( )** function reads an **mi_decimal** (DECIMAL) value from a stream, converting any difference in the stream representation to that of the internal representation.

Syntax

```
mi_integer mi_streamread_decimal(strm_desc, dec_dptr)
   MI_STREAM *strm_desc;
   mi_decimal **dec_dptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream from which to read the **mi_decimal** value.
dec_dptr	is a pointer to the buffer into which to copy the address of the **mi_decimal** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamread_decimal( )** function reads an **mi_decimal** value from the stream that *strm_desc* references. The function reads the **mi_decimal** value from the stream, puts the address of the value in the buffer that *dec_dptr* references, and allocates memory for the value in the current memory duration. The read operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

When **dec_dptr* points to NULL, **mi_streamread_decimal( )** allocates the memory for the buffer in the current memory duration. Otherwise, the function assumes that you have allocated the memory that **dec_dptr* references.

Important: Be sure that **dec_dptr* points to NULL if the parameter does not point to valid memory.

The **mi_streamread_decimal( )** function is useful in a **streamread( )** support function of an opaque data type that contains an **mi_decimal** value.

Return Values

>=0	is the actual number of bytes that the function has read from the open stream to the value that *dec_dptr* references.
MI_STREAM_EEOF	indicates that the end of the stream has been reached without any errors.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the value that *dec_dptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamwrite_decimal( ).**

For more information on the use of **mi_streamread_decimal( )** in a **streamread( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamread_double( )

The **mi_streamread_double( )** function reads an **mi_double_precision** (FLOAT) value from a stream, converting any difference in the stream representation to that of the internal representation.

Syntax

```
mi_integer mi_streamread_double(strm_desc, dbl_dptr)
   MI_STREAM *strm_desc;
   mi_double_precision **dbl_dptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream from which to read the **mi_double_precision** value.
dbl_dptr	is a pointer to the buffer into which to copy the address of the **mi_double_precision** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamread_double( )** function reads an **mi_double_precision** value from the stream that *strm_desc* references. The function reads the **mi_double_precision** value from the stream and puts the address of the value in the buffer that **dbl_dptr* references. The read operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

When **dbl_dptr* points to NULL, **mi_streamread_double( )** allocates the memory for the buffer in the current memory duration. Otherwise, the function assumes that you have allocated the memory that **dbl_dptr* references.

Important: Be sure that **dbl_dptr* points to NULL if the parameter does not point to valid memory.

The **mi_streamread_double( )** function is useful in a **streamread( )** support function of an opaque data type that contains an **mi_double_precision** value.

Return Values

>=0	is the actual number of bytes that the function has read from the open stream to the value that *dbl_dptr* references.
MI_STREAM_EEOF	indicates that the end of the stream has been reached without any errors.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the value that *dbl_dptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamwrite_double( ).**

For more information on the use of **mi_streamread_double( )** in a **streamread( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamread_int8( )

The **mi_streamread_int8( )** function reads an **mi_int8** (INT8) value from a stream, converting any difference in the stream representation to that of the internal representation.

Syntax

```
mi_integer mi_streamread_int8(strm_desc, int8_dptr)
   MI_STREAM *strm_desc;
   mi_int8 **int8_dptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream from which to read the **mi_int8** value.
int8_dptr	is a pointer to the buffer into which to copy the address of the **mi_int8** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamread_int8( )** function reads an **mi_int8** value from the stream that *strm_desc* references. The function reads the **mi_int8** value from the stream and puts the address of the value in the buffer that **int8_dptr* references. The read operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

When **int8_dptr* points to NULL, **mi_streamread_int8( )** allocates the memory for the buffer in the current memory duration. Otherwise, the function assumes that you have allocated the memory that **int8_dptr* references.

Important: Be sure that **int8_dptr* points to NULL if the parameter does not point to valid memory.

The **mi_streamread_int8( )** function is useful in a **streamread( )** support function of an opaque data type that contains an **mi_int8** value.

Return Values

>=0	is the actual number of bytes that the function has read from the open stream to the value that *int8_dptr* references.
MI_STREAM_EEOF	indicates that the end of the stream has been reached without any errors.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the value that *int8_dptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamwrite_int8( ).**

For more information on the use of **mi_streamread_int8( )** in a **streamread( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamread_integer( )

The **mi_streamread_integer( )** function reads an **mi_integer** (INTEGER) value from a stream, converting any difference in the stream representation to that of the internal representation.

Syntax

```
mi_integer mi_streamread_integer(strm_desc, int_ptr)
   MI_STREAM *strm_desc;
   mi_integer *int_ptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream from which to read the **mi_integer** value.
int_ptr	is a pointer to the buffer into which to copy the **mi_integer** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamread_integer( )** function reads an **mi_integer** value from the stream that *strm_desc* references. The function reads this value into the **mi_integer** buffer that *int_ptr* references. The read operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

The **mi_streamread_integer( )** function is useful in a **streamread( )** support function of an opaque data type that contains an **mi_integer** value.

Return Values

>=0	is the actual number of bytes that the function has read from the open stream to the value that *int_ptr* references.
MI_STREAM_EEOF	indicates that the end of the stream has been reached without any errors.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the value that *int_ptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamwrite_integer( ).**

For more information on the use of **mi_streamread_integer( )** in a **streamread( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamread_interval( )

The **mi_streamread_interval( )** function reads an **mi_interval** (INTERVAL) value from a stream, converting any difference in the stream representation to that of the internal representation.

Syntax

```
mi_integer mi_streamread_interval(strm_desc, intrvl_dptr)
   MI_STREAM *strm_desc;
   mi_interval **intrvl_dptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream from which to read the **mi_interval** value.
intrvl_dptr	is a pointer to the buffer into which to copy the address of the **mi_interval** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamread_interval( )** function reads an **mi_interval** value from the stream that *strm_desc* references. The function reads the **mi_interval** value from the stream and puts the address of the value in the buffer that **intrvl_dptr* references. The read operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

When **intrvl_dptr* points to NULL, **mi_streamread_interval( )** allocates the memory for the buffer in the current memory duration. Otherwise, the function assumes that you have allocated the memory that **intrvl_dptr* references.

Important: Be sure that **intrvl_dptr* points to NULL if the parameter does not point to valid memory.

The **mi_streamread_interval( )** function is useful in a **streamread( )** support function of an opaque data type that contains an **mi_interval** value.

Return Values

>=0	is the actual number of bytes that the function has read from the open stream to the value that *intrvl_dptr* references.
MI_STREAM_EEOF	indicates that the end of the stream has been reached without any errors.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the value that *intrvl_dptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamwrite_interval( ).**

For more information on the use of **mi_streamread_interval( )** in a **streamread( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamread_lo( )

The **mi_streamread_lo( )** function reads smart-large-object data from a stream and copies the data to a smart large object in the default sbspace.

Syntax

```
mi_integer mi_streamread_lo(strm_desc, LO_hdl_dptr)
   MI_STREAM *strm_desc;
   MI_LO_HANDLE **LO_hdl_dptr;
```

strm_desc	is a pointer to the stream descriptor for the stream from which to read the smart-large-object data.
LO_hdl_dptr	is a pointer to the buffer to contain the address of the smart large object to which to copy the data.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamread_lo( )** function reads smart-large-object data from the stream that *strm_desc* references and copies the data to the smart large object at the address that LO*_hdl_dptr* references, which is always in the default sbspace. To control the location of the smart large object, you can use the **mi_streamread_lo_by_lofd( )** function instead.

The read begins at the current seek position of the stream. You can use the **mi_stream_tell( )** or **mi_stream_getpos( )** function to obtain the current stream seek position.

Important: The *mi_streamread_lo( )* function requires the caller to have an open connection to the database server.

The function is a constructor function for an LO handle. It allocates memory for the LO handle in the current memory duration.

The **mi_streamread_lo( )** function is useful in a **streamread( )** support function of an opaque data type that contains a smart large object.

Return Values

>=0	is the actual number of bytes that the function has read from the open stream to the smart large object that LO*_hdl_dptr* references.
MI_STREAM_EEOF	indicates that the end of the stream has been reached without any errors.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the smart large object that LO*_hdl_dptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ), mi_streamread_lo_by_lofd( ),** and **mi_streamwrite_lo( ).**

mi_streamread_lo_by_lofd( )

The **mi_streamread_lo_by_lofd( )** function reads smart-large-object data from a stream and copies the data to the open smart large object that a specified LO file descriptor identifies.

Syntax

```
mi_integer mi_streamread_lo_by_lofd(strm_desc, LO_fd)
   MI_STREAM *strm_desc;
   MI_LO_FD LO_fd;
```

strm_desc	is a pointer to the stream descriptor for the open stream from which to read the smart-large-object data.
*LO*_fd	is an LO file descriptor for an open smart large object.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamread_lo_by_lofd( )** function reads data from the open stream that *strm_desc* references and appends the data to the open smart large object that LO*_fd* references. This function does not close the smart large object or modify its reference count. The read operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

This function is useful in the **streamread( )** support function of an opaque data type that contains a smart large object.

Important: The **mi_streamread_lo_by_lofd( )** function requires the caller to have an open connection to the database server.

Return Values

>=0	is the actual number of bytes that the function has read from the open stream to the smart large object.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ), mi_streamread_lo( ),** and **mi_streamwrite_lo( ).**

For more information on the use of **mi_streamread_lo_by_lofd( )** in a **streamread( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamread_lvarchar( )

The **mi_streamread_lvarchar( )** function reads a varying-length structure (**mi_lvarchar**) value from a stream, converting any difference in the stream representation to that of the internal representation.

Syntax

```
mi_integer mi_streamread_lvarchar(strm_desc, varlen_dptr)
   MI_STREAM *strm_desc;
   mi_lvarchar **varlen_dptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream from which to read the varying-length structure.
varlen_dptr	is a pointer to the buffer into which to copy the address of the **mi_lvarchar** varying-length structure.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamread_lvarchar( )** function reads a varying-length structure from the stream that *strm_desc* references. The function reads the varying-length structure from the stream and puts the address of the value in the buffer that **varlen_dptr* references. The read operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

The **mi_streamread_lvarchar( )** function is a constructor function for a varying-length structure. It allocates memory for this varying-length structure in the current memory duration.

This function is useful in a **streamread( )** support function of an opaque data type that contains a varying-length structure (such as **mi_lvarchar)**.

Return Values

>=0	is the actual number of bytes that the function has read from the open stream to the value that *varlen_dptr* references.
MI_STREAM_EEOF	indicates that the end of the stream has been reached without any errors.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the value that *varlen_dptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamwrite_lvarchar( ).**

For more information on the use of **mi_streamread_lvarchar( )** in a **streamread( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamread_money( )

The **mi_streamread_money( )** function reads an **mi_money** (MONEY) value from a stream, converting any difference in the stream representation to that of the internal representation.

Syntax

```
mi_integer mi_streamread_money(strm_desc, money_dptr)
   MI_STREAM *strm_desc;
   mi_money **money_dptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream from which to read the **mi_money** value.
money_dptr	is a pointer to the buffer into which to copy the address of the **mi_money** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamread_money( )** function reads an **mi_money** value from the stream that *strm_desc* references. The function reads the **mi_money** value from the stream and puts the address of the value in the buffer that **money_dptr* references. The read operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

When **money_dptr* points to NULL, **mi_streamread_money( )** allocates the memory for the buffer in the current memory duration. Otherwise, the function assumes that you have allocated the memory that **money_dptr* references.

Important: Be sure that **money_dptr* points to NULL if the parameter does not point to valid memory.

The **mi_streamread_money( )** function is useful in a **streamread( )** support function of an opaque data type that contains an **mi_money** value.

Return Values

>=0	is the actual number of bytes that the function has read from the open stream to the value that *money_dptr* references.
MI_STREAM_EEOF	indicates that the end of the stream has been reached without any errors.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the value that *money_dptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamwrite_money( ).**

For more information on the use of **mi_streamread_money( )** in a **streamread( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamread_real( )

The **mi_streamread_real( )** function reads an **mi_real** (SMALLFLOAT) value from a stream, converting any difference in the stream representation to that of the internal representation.

Syntax

```
mi_integer mi_streamread_real(strm_desc, real_dptr)
   MI_STREAM *strm_desc;
   mi_real **real_dptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream from which to read the **mi_real** value.
real_dptr	is a pointer to the buffer into which to copy the address of the **mi_real** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamread_real( )** function reads an **mi_real** value from the stream that *strm_desc* references. The function reads the **mi_real** value from the stream and puts the address of the value in the buffer that **real_dptr* references. The read operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

When **real_dptr* points to NULL, **mi_streamread_real( )** allocates the memory for the buffer in the current memory duration. Otherwise, the function assumes that you have allocated the memory that **real_dptr* references.

Important: Be sure that **real_dptr* points to NULL if the parameter does not point to valid memory.

The **mi_streamread_real( )** function is useful in a **streamread( )** support function of an opaque data type that contains an **mi_real** value.

Return Values

>=0	is the actual number of bytes that the function has read from the open stream to the value that *real_dptr* references.
MI_STREAM_EEOF	indicates that the end of the stream has been reached without any errors.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the value that *real_dptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamwrite_real( ).**

For more information on the use of **mi_streamread_real( )** in a **streamread( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamread_row( )

The **mi_streamread_row( )** function reads a row structure (row type) value from a stream, converting any difference in the stream representation to that of the internal representation.

Syntax

```
mi_integer mi_streamread_row(strm_desc, rowstruc_dptr, fparam_ptr)
   MI_STREAM *strm_desc;
   MI_ROW **rowstruc_dptr;
   MI_FPARAM *fparam_ptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream from which to read the row structure (**MI_ROW**).
rowstruc_dptr	is a pointer to the buffer into which to copy the address of the row structure.
fparam_ptr	is a pointer to the **MI_FPARAM** structure for the user-defined routine that calls **mi_streamread_row( )**.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Important: Enterprise Replication does not support this function.

Usage

The **mi_streamread_row( )** function reads a row structure from the stream that *strm_desc* references. The function reads the row structure from the stream and puts its address in the buffer that **rowstruc_dptr* references. The read operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

The **mi_streamread_row( )** function is a constructor function for a row structure. It allocates memory for this row structure in the current memory duration.

This function is useful in a **streamread( )** support function of an opaque data type that contains a row structure (**MI_ROW**).

Important: The **mi_streamread_row( )** function requires the caller to have an open connection to the database server.

Return Values

>=0	is the actual number of bytes that the function has read from the open stream to the value that *rowstruc_dptr* references.
MI_STREAM_EEOF	indicates that the end of the stream has been reached without any errors.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the value that *rowstruc_dptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamwrite_row( ).**

For more information on the use of **mi_streamread_row( )** in a **streamread( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamread_smallint( )

The **mi_streamread_smallint( )** function reads an **mi_smallint** (SMALLINT) value from a stream, converting any difference in the stream representation to that of the internal representation.

Syntax

```
mi_integer mi_streamread_smallint(strm_desc, smallint_ptr)
   MI_STREAM *strm_desc;
   mi_smallint *smallint_ptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream from which to read the **mi_smallint** value.
smallint_ptr	is a pointer to the buffer into which to copy the **mi_smallint** value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamread_smallint( )** function reads an **mi_smallint** value from the stream that *strm_desc* references. The function reads this value into the **mi_smallint** buffer that *smallint_ptr* references. The read operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

The **mi_streamread_smallint( )** function is useful in a **streamread( )** support function of an opaque data type that contains an **mi_smallint** value.

Return Values

>=0	is the actual number of bytes that the function has read from the open stream to the value that *smallint_ptr* references.
MI_STREAM_EEOF	indicates that the end of the stream has been reached without any errors.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the value that *smallint_ptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamwrite_smallint( ).**

For more information on the use of **mi_streamread_smallint( )** in a **streamread( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamread_string( )

The **mi_streamread_string( )** function reads an **mi_string** (CHAR(x)) value from a stream.

Syntax

```
mi_integer mi_streamread_string(stream, str_data, fparam_ptr)
   MI_STREAM *stream;
   mi_string **str_data;
   MI_FPARAM *fparam_ptr;
```

stream	is a pointer to the stream descriptor for the open stream from which to read the character string.
str_data	is a pointer to the buffer into which to copy the address of the character string.
fparam_ptr	is a pointer to the **MI_FPARAM** structure for the user-defined routine that calls **mi_streamread_string( )**.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamread_string( )** function reads an **mi_string** value (or other character string) from the stream that *stream* references. The function reads the **mi_string** value from the stream, puts the address of the value in the buffer that **str_data* references, and allocates memory for the buffer in the current memory duration. The read operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

The **mi_streamread_string( )** function is useful in a **streamread( )** support function of an opaque data type that contains an **mi_string** value.

Global Language Support

If code-set conversion is required, the **mi_streamread_string( )** function converts the **mi_string** value from the code set of the client locale to that of the server-processing locale. For more information about code-set conversion, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Return Values

>=0	is the actual number of bytes that the function has read from the open stream to the value that *stream* references.
MI_STREAM_EEOF	indicates that the end of the stream has been reached without any errors.
MI_STREAM_EBADARG	indicates that the stream descriptor that *stream* references or the value that *str_data* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamwrite_string( ).**

For more information on the use of **mi_streamread_string( )** in a **streamread( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamwrite_boolean( )

The **mi_streamwrite_boolean( )** function writes an **mi_boolean** (BOOLEAN) value to a stream, converting any difference in the internal representation to that of the stream representation.

Syntax

```
mi_integer mi_streamwrite_boolean(strm_desc, bool_data)
   MI_STREAM *strm_desc;
   mi_boolean bool_data;
```

strm_desc	is a pointer to the stream descriptor for the open stream to which to write the **mi_boolean** value.
bool_data	is the **mi_boolean** value to write to the stream.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamwrite_boolean( )** function writes an **mi_boolean** value to the stream that *strm_desc* references. The function writes the value in *bool_data*. The write operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

The **mi_streamwrite_boolean( )** function is useful in a **streamwrite( )** support function of an opaque data type that contains an **mi_boolean** value.

Return Values

>=0	is the actual number of bytes that the function has written to the open stream.
MI_STREAM_EEOF	indicates that the end of the stream has been reached.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the *bool_data* value is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamread_boolean( ).**

For more information on the use of **mi_streamwrite_boolean( )** in a **streamwrite( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamwrite_collection( )

The **mi_streamwrite_collection( )** function writes a collection structure (LIST, SET, MULTISET) value to a stream, converting any difference in the internal representation to that of the stream representation.

Syntax

```
mi_integer mi_streamwrite_collection(strm_desc, coll_ptr)
   MI_STREAM *strm_desc;
   MI_COLLECTION *coll_ptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream to which to write the collection structure (**MI_COLLECTION**).
coll_ptr	is a pointer to the collection structure to write to the stream.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Important: Enterprise Replication does not support this function.

Usage

The **mi_streamwrite_collection( )** function writes a collection structure to the stream that *strm_desc* stream references. The function writes the value that *coll_ptr* references. The write operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

The **mi_streamwrite_collection( )** function is useful in a **streamwrite( )** support function of an opaque data type that contains a collection structure (**MI_COLLECTION**).

Important: The **mi_streamwrite_collection( )** function requires the caller to have an open connection to the database server.

Return Values

>=0 is the actual number of bytes that the function has written to the open stream.

MI_STREAM_EEOF
indicates that the end of the stream has been reached.

MI_STREAM_EBADARG
indicates that the stream descriptor that *strm_desc* references or the value that *coll_ptr* references is invalid.

MI_ERROR
indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamread_collection( ).**

For more information on the use of **mi_streamwrite_collection( )** in a **streamwrite( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamwrite_date( )

The **mi_streamwrite_date( )** function writes an **mi_date** (DATE) value to a stream, converting any difference in the internal representation to that of the stream representation.

Syntax

```
mi_integer mi_streamwrite_date(strm_desc, date_data)
   MI_STREAM *strm_desc;
   mi_date date_data;
```

strm_desc	is a pointer to the stream descriptor for the open stream to which to write the **mi_date** value.
date_data	is the **mi_date** value to write to the stream.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamwrite_date( )** function writes an **mi_date** value to the stream that *strm_desc* references. The function writes the *date_data* value. The write operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

The **mi_streamwrite_date( )** function is useful in a **streamwrite( )** support function of an opaque data type that contains an **mi_date** value.

Return Values

>=0	is the actual number of bytes that the function has written to the open stream.
MI_STREAM_EEOF	indicates that the end of the stream has been reached.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the *date_data* value is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamread_date( ).**

For more information on the use of **mi_streamwrite_date( )** in a **streamwrite( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamwrite_datetime( )

The **mi_streamwrite_datetime( )** function writes an **mi_datetime** (DATETIME) value to a stream, converting any difference in the internal representation to that of the stream representation.

Syntax

```
mi_integer mi_streamwrite_datetime(strm_desc, dtime_ptr)
   MI_STREAM *strm_desc;
   mi_datetime *dtime_ptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream to which to write the **mi_datetime** value.
dtime_ptr	is a pointer to the **mi_datetime** value to write to the stream.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamwrite_datetime( )** function writes an **mi_datetime** value to the stream that *strm_desc* references. The function writes the value that *dtime_ptr* references. The write operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

The **mi_streamwrite_datetime( )** function is useful in a **streamwrite( )** support function of an opaque data type that contains an **mi_datetime** value.

Return Values

>=0	is the actual number of bytes that the function has written to the open stream.
MI_STREAM_EEOF	indicates that the end of the stream has been reached.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the value that *dtime_ptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamread_datetime( ).**

For more information on the use of **mi_streamwrite_datetime( )** in a **streamwrite( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamwrite_decimal( )

The **mi_streamwrite_decimal( )** function writes an **mi_decimal** (DECIMAL) value to a stream, converting any difference in the internal representation to that of the stream representation.

Syntax

```
mi_integer mi_streamwrite_decimal(strm_desc, dec_ptr)
   MI_STREAM *strm_desc;
   mi_decimal *dec_ptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream to which to write the **mi_decimal** value.
dec_ptr	is a pointer to the **mi_decimal** value to write to the stream.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamwrite_decimal( )** function writes an **mi_decimal** value to the stream that *strm_desc* references. The function writes the value that *dec_ptr* references. The write operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

The **mi_streamwrite_decimal( )** function is useful in a **streamwrite( )** support function of an opaque data type that contains an **mi_decimal** value.

Return Values

>=0	is the actual number of bytes that the function has written to the open stream.
MI_STREAM_EEOF	indicates that the end of the stream has been reached.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the value that *dec_ptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamread_decimal( ).**

For more information on the use of **mi_streamwrite_decimal( )** in a **streamwrite( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamwrite_double( )

The **mi_streamwrite_double( )** function writes an **mi_double_precision** (FLOAT) value to a stream, converting any difference in the internal representation to that of the stream representation.

Syntax

```
mi_integer mi_streamwrite_double(strm_desc, dbl_ptr)
   MI_STREAM *strm_desc;
   mi_double_precision *dbl_ptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream to which to write the **mi_double_precision** value.
dbl_ptr	is a pointer to the **mi_double_precision** value to write to the stream.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamwrite_double( )** function writes an **mi_double_precision** value to the stream that *strm_desc* references. The function writes the value that *dbl_ptr* references. The write operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

The **mi_streamwrite_double( )** function is useful in a **streamwrite( )** support function of an opaque data type that contains an **mi_double_precision** value.

Return Values

>=0	is the actual number of bytes that the function has written to the open stream.
MI_STREAM_EEOF	indicates that the end of the stream has been reached.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the value that *dbl_ptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamread_double( ).**

For more information on the use of **mi_streamwrite_double( )** in a **streamwrite( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamwrite_int8( )

The **mi_streamwrite_int8( )** function writes an **mi_int8** (INT8) value to a stream, converting any difference in the internal representation to that of the stream representation.

Syntax

```
mi_integer mi_streamwrite_int8(strm_desc, int8_ptr)
   MI_STREAM *strm_desc;
   mi_int8 *int8_ptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream to which to write the **mi_int8** value.
int8_ptr	is a pointer to the **mi_int8** value to write to the stream.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamwrite_int8( )** function writes an **mi_int8** value to the stream that *strm_desc* references. The function writes the value that *int8_ptr* references. The write operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

The **mi_streamwrite_int8( )** function is useful in a **streamwrite( )** support function of an opaque data type that contains an **mi_int8** value.

Return Values

>=0	is the actual number of bytes that the function has written to the open stream.
MI_STREAM_EEOF	indicates that the end of the stream has been reached.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the value that *int8_ptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamread_int8( ).**

For more information on the use of **mi_streamwrite_int8( )** in a **streamwrite( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamwrite_integer( )

The **mi_streamwrite_integer( )** function writes an **mi_integer** (INTEGER) value to a stream, converting any difference in the internal representation to that of the stream representation.

Syntax

```
mi_integer mi_streamwrite_integer(strm_desc, int_data)
   MI_STREAM *strm_desc;
   mi_integer int_data;
```

strm_desc	is a pointer to the stream descriptor for the open stream to which to write the **mi_integer** value.
int_data	is the **mi_integer** value to write to the stream.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamwrite_integer( )** function writes an **mi_integer** value to the stream that *strm_desc* references. The function writes the *int_data* value. The write operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

The **mi_streamwrite_integer( )** function is useful in a **streamwrite( )** support function of an opaque data type that contains an **mi_integer** value.

Return Values

>=0	is the actual number of bytes that the function has written to the open stream.
MI_STREAM_EEOF	indicates that the end of the stream has been reached.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the *int_data* value is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamread_integer( ).**

For more information on the use of **mi_streamwrite_integer( )** in a **streamwrite( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamwrite_interval( )

The **mi_streamwrite_interval( )** function writes an **mi_interval** (INTERVAL) value to a stream, converting any difference in the internal representation to that of the stream representation.

Syntax

```
mi_integer mi_streamwrite_interval(strm_desc, intrvl_ptr)
   MI_STREAM *strm_desc;
   mi_interval *intrvl_ptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream to which to write the **mi_interval** value.
intrvl_ptr	is a pointer to the **mi_interval** value to write to the stream.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamwrite_interval( )** function writes an **mi_interval** value to the stream that *strm_desc* references. The function writes the value that *intrvl_ptr* references. The write operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

The **mi_streamwrite_interval( )** function is useful in a **streamwrite( )** support function of an opaque data type that contains an **mi_interval** value.

Return Values

>=0	is the actual number of bytes that the function has written to the open stream.
MI_STREAM_EEOF	indicates that the end of the stream has been reached.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the value that *intrvl_ptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamread_interval( ).**

For more information on the use of **mi_streamwrite_interval( )** in a **streamwrite( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamwrite_lo( )

The **mi_streamwrite_lo( )** function writes a smart large object to a stream, converting any difference in the internal representation to that of the stream representation.

Syntax

```
mi_integer mi_streamwrite_lo(strm_desc, LO_hdl_ptr)
   MI_STREAM *strm_desc;
   MI_LO_HANDLE *LO_hdl_ptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream to which to write the LO handle.
LO_hdl_ptr	is a pointer to the LO handle to write to the stream.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamwrite_lo( )** function writes the smart large object that LO*_hdl_ptr* references to the stream that *strm_desc* references. The write operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

Important: The *mi_streamwrite_lo( )* function requires the caller to have an open connection to the database server.

The **mi_streamwrite_lo( )** function is useful in the **streamwrite( )** support function of an opaque data type that contains a smart large object.

Return Values

>=0	is the actual number of bytes that the function has written to the open stream.
MI_STREAM_EEOF	indicates that the end of the stream has been reached.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the value that LO*_hdl_ptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamread_lo_by_lofd( ).**

For more information on the use of **mi_streamwrite_lo( )** in a **streamwrite( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamwrite_lvarchar( )

The **mi_streamwrite_lvarchar( )** function writes a varying-length structure (**mi_lvarchar**) value to a stream, converting any difference in the internal representation to that of the stream representation.

Syntax

```
mi_integer mi_streamwrite_lvarchar(strm_desc, varlen_ptr)
   MI_STREAM *strm_desc;
   mi_lvarchar *varlen_ptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream to which to write the varying-length structure.
varlen_ptr	is a pointer to the **mi_lvarchar** varying-length structure to write to the stream.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamwrite_lvarchar( )** function writes a varying-length structure to the stream that *strm_desc* references. The function writes the varying-length structure that *varlen_ptr* references. The write operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

The **mi_streamwrite_lvarchar( )** function is useful in a **streamwrite( )** support function of an opaque data type that contains a varying-length structure (such as **mi_lvarchar**).

Return Values

>=0	is the actual number of bytes that the function has written to the open stream.
MI_STREAM_EEOF	indicates that the end of the stream has been reached.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the value that *varlen_ptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamread_lvarchar( ).**

For more information on the use of **mi_streamwrite_lvarchar( )** in a **streamwrite( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamwrite_money( )

The **mi_streamwrite_money( )** function writes an **mi_money** (MONEY) value to a stream, converting any difference in the internal representation to that of the stream representation.

Syntax

```
mi_integer mi_streamwrite_money(strm_desc, money_ptr)
   MI_STREAM *strm_desc;
   mi_money *money_ptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream to which to write the **mi_money** value.
money_ptr	is a pointer to the **mi_money** value to write to the stream.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamwrite_money( )** function writes an **mi_money** value to the stream that *strm_desc* references. The function writes the value that *money_ptr* references. The write operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

The **mi_streamwrite_money( )** function is useful in a **streamwrite( )** support function of an opaque data type that contains an **mi_money** value.

Return Values

>=0	is the actual number of bytes that the function has written to the open stream.
MI_STREAM_EEOF	indicates that the end of the stream has been reached.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the value that *money_ptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamread_money( ).**

For more information on the use of **mi_streamwrite_money( )** in a **streamwrite( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamwrite_real( )

The **mi_streamwrite_real( )** function writes an **mi_real** (SMALLFLOAT) value to a stream, converting any difference in the internal representation to that of the stream representation.

Syntax

```
mi_integer mi_streamwrite_real(strm_desc, real_ptr)
   MI_STREAM *strm_desc;
   mi_real *real_ptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream to which to write the **mi_real** value.
real_ptr	is a pointer to the **mi_real** value to write to the stream.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamwrite_real( )** function writes an **mi_real** value to the stream that *strm_desc* references. The function writes the value that *real_ptr* references. The write operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

The **mi_streamwrite_real( )** function is useful in a **streamwrite( )** support function of an opaque data type that contains an **mi_real** value.

Return Values

>=0	is the actual number of bytes that the function has written to the open stream.
MI_STREAM_EEOF	indicates that the end of the stream has been reached.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the value that *real_ptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamread_real( ).**

For more information on the use of **mi_streamwrite_real( )** in a **streamwrite( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamwrite_row( )

The **mi_streamwrite_row( )** function writes a row structure (row type) value to a stream, converting any difference in the internal representation to that of the stream representation.

Syntax

```
mi_integer mi_streamwrite_row(strm_desc, rowstruc_ptr, fparam_ptr)
   MI_STREAM *strm_desc;
   MI_ROW *rowstruc_ptr;
   MI_FPARAM *fparam_ptr;
```

strm_desc	is a pointer to the stream descriptor for the open stream to which to write the row structure (**MI_ROW**).
rowstruc_ptr	is a pointer to the row structure to write to the stream.
fparam_ptr	is a pointer to the **MI_FPARAM** structure for the user-defined routine that calls **mi_streamwrite_row( )**.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Important: Enterprise Replication does not support this function.

Usage

The **mi_streamwrite_row( )** function writes a row structure to the stream that *strm_desc* references. The function writes the value that *rowstruc_ptr* references. The write operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

The **mi_streamwrite_row( )** function is useful in a **streamwrite( )** support function of an opaque data type that contains a row structure (**MI_ROW**).

Important: The **mi_streamwrite_row( )** function requires the caller to have an open connection to the database server.

Return Values

>=0	is the actual number of bytes that the function has written to the open stream.
MI_STREAM_EEOF	indicates that the end of the stream has been reached.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references or the value that *rowstruc_ptr* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamread_row( ).**

For more information on the use of **mi_streamwrite_row( )** in a **streamwrite( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamwrite_smallint( )

The **mi_streamwrite_smallint( )** function writes an **mi_smallint** (SMALLINT) value to a stream, converting any difference in the internal representation to that of the stream representation.

Syntax

```
mi_integer mi_streamwrite_smallint(strm_desc, smallint_data)
   MI_STREAM *strm_desc;
   mi_smallint smallint_data;
```

strm_desc	is a pointer to the stream descriptor for the open stream to which to write the **mi_smallint** value.
smallint_data	is the **mi_smallint** value to write to the stream.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamwrite_smallint( )** function writes the *smallint_data* value to the stream that *strm_desc* references. The write operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

The **mi_streamwrite_smallint( )** function is useful in a **streamwrite( )** support function of an opaque data type that contains an **mi_smallint** value.

Return Values

>=0	is the actual number of bytes that the function has written to the open stream.
MI_STREAM_EEOF	indicates that the end of the stream has been reached.
MI_STREAM_EBADARG	indicates that the stream descriptor that *strm_desc* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamread_smallint( ).**

For more information on the use of **mi_streamwrite_smallint( )** in a **streamwrite( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_streamwrite_string( )

The **mi_streamwrite_string( )** function writes an **mi_string** (CHAR(x)) value to a stream.

Syntax

```
mi_integer mi_streamwrite_string(stream, str_data, fparam_ptr)
   MI_STREAM *stream;
   mi_string *str_data;
   MI_FPARAM *fparam_ptr;
```

stream	is a pointer to the stream descriptor for the open stream to which to write the character string.
str_data	is a pointer to a buffer that contains the character string.
fparam_ptr	is a pointer to the **MI_FPARAM** structure for the user-defined routine that calls **mi_streamwrite_string( )**.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_streamwrite_string( )** function writes an **mi_string** value to the stream that *stream* references. The function writes the value that *str_data* references. The write operation begins at the current stream seek position. You can use **mi_stream_tell( )** or **mi_stream_getpos( )** to obtain this seek position.

The **mi_streamwrite_string( )** function is useful in a **streamwrite( )** support function of an opaque data type that contains an **mi_string** value.

Global Language Support

If code-set conversion is required, the **mi_streamwrite_string( )** function converts the **mi_string** value from the code set of the client locale to that of the server-processing locale. For more information about code-set conversion, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Return Values

>=0	is the actual number of bytes that the function has written to the open stream.
MI_STREAM_EEOF	indicates that the end of the stream has been reached.
MI_STREAM_EBADARG	indicates that the stream descriptor that *stream* references or the value that *str_data* references is invalid.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_stream_getpos( ), mi_stream_read( ), mi_stream_tell( ),** and **mi_streamread_string( ).**

For more information on the use of **mi_streamwrite_string( )** in a **streamwrite( )** support function, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_string_to_date( )

The **mi_string_to_date( )** function converts a text (string) representation of a date value to its binary (internal) DATE representation.

Syntax

```
mi_date mi_string_to_date(date_string)
   mi_string *date_string;
```

date_string	is a pointer to the date string to convert to its internal DATE format.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_string_to_date( )** function converts the date string that *date_string* references to its internal DATE value. An internal DATE value is the format that the database server uses to store a date in a column of the database.

Important: The **mi_string_to_date( )** function replaces the **mi_date_to_binary( )** function for string-to-internal-DATE conversion in DataBlade API modules.

Global Language Support

The **mi_string_to_date( )** function accepts the date string in the date format of the current processing locale. For more information about the environment factors that determine the format of a date string, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Return Values

An **mi_date** value	is the internal DATE representation that **mi_string_to_date( )** has created.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_date_to_string( ), mi_string_to_datetime( ), mi_string_to_decimal( ), mi_string_to_interval( ),** and **mi_string_to_money( ).**

For more information on how to convert date strings to internal DATE format, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_string_to_datetime( )

The **mi_string_to_datetime( )** function converts a text (string) representation of a date, time, or date and time value to its internal (binary) DATETIME representation.

Syntax

```
mi_datetime *mi_string_to_datetime(dt_string, type_range)
   mi_string *dt_string;
   mi_string *type_range;
```

dt_string	is a pointer to the date, time, or date and time string to convert to its internal DATETIME format.
type_range	is a pointer to a string that specifies the range of DATETIME qualifiers in the date, time, or date and time string that *dt_string* references.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

The **mi_string_to_datetime( )** function converts the date and/or time string that *dt_string* references to its internal DATETIME value. An internal DATETIME value is the format that the database server uses to store a value in a DATETIME column of the database.

The date and/or time string that *dt_string* references has the following fixed ANSI SQL format:

```
"YYYY-MM-DD HH:mm:SS.FFFFF"
```

YYYY	is the 4-digit year.
MM	is the 2-digit month.
DD	is the 2-digit day.
HH	is the 2-digit hour.
mm	is the 2-digit minute.
SS	is the 2-digit second.
FFFFF	is the fraction of a second, in which the date, time, or date and time qualifier specifies the number of digits, with a maximum precision of 5 digits.

Server Only

When you call **mi_string_to_datetime( )** in a C UDR, this date and/or time string can contain only a subset of this range. In this case, the *type_range* argument must specify the range of qualifiers that the date and/or time string contains. This qualifier range begins with the keyword **datetime** and is followed by the range of qualifiers for the value in the *dt_string*. For example, the following call to **mi_string_to_datetime( )** converts the date *01/31/07* and a time of 10:30 A.M. to its internal DATETIME format:

```
mi_datetime *internal_dt;
...
internal_dt = mi_string_to_datetime(
            "2007-01-31 10:30",
            "datetime year to minute");
```

If the *type_range* argument is a NULL-valued pointer, the **mi_string_to_datetime( )** function assumes a default qualifier range of:

```
"datetime year to second"
```

End of Server Only

Client Only

When you call **mi_string_to_datetime( )** in a client LIBMI application, this date and/or time string must contain the full range of qualifiers. The *type_range* argument must also specify the full range. For example, the following call to **mi_string_to_datetime( )** converts the date *05/31/07* and a time of 8:52 a.m. to its internal DATETIME format:

```
mi_datetime *internal_dt;
...
internal_dt = mi_string_to_datetime(
            "2007-05-31 08:52:12",
            "datetime year to second");
```

End of Client Only

Important: The **mi_string_to_datetime( )** function replaces the **mi_datetime_to_binary( )** function for string-to-internal-DATETIME conversion in DataBlade API modules.

Global Language Support

The **mi_string_to_datetime( )** function does *not* accept the date and/or time string in the date and time formats of the current processing locale.

End of Global Language Support

Return Values

An **mi_datetime** pointer	is a pointer to the internal DATETIME representation that **mi_string_to_datetime( )** has created.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_datetime_to_string( ), mi_string_to_date( ), mi_string_to_decimal( ), mi_string_to_interval( ),** and **mi_string_to_money( ).**

For more information on how to convert date and/or time strings to internal DATETIME format, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_string_to_decimal( )

The **mi_string_to_decimal( )** function converts a text (string) representation of a decimal value to its binary (internal) DECIMAL representation.

Syntax

```
mi_decimal *mi_string_to_decimal(decimal_string)
   mi_string *decimal_string;
```

decimal_string	is a pointer to the decimal string to convert to its internal DECIMAL format.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_string_to_decimal( )** function converts the decimal string that *decimal_string* references to its internal DECIMAL (**mi_decimal**) value. An internal DECIMAL value is the format that the database server uses to store a value in a DECIMAL column of the database. This format can represent both fixed-point and floating-point decimal numbers.

Important: The **mi_string_to_decimal( )** function replaces the **mi_decimal_to_binary( )** function for string-to-internal-DECIMAL conversion in DataBlade API modules.

Global Language Support

The **mi_string_to_decimal( )** function accepts the decimal string in the numeric format of the current processing locale. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Return Values

An **mi_decimal** pointer	is a pointer to the internal DECIMAL representation that **mi_string_to_decimal( )** has created.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_decimal_to_string( ), mi_string_to_date( ), mi_string_to_datetime( ), mi_string_to_interval( ),** and **mi_string_to_money( ).**

For more information on how to convert decimal strings to internal DECIMAL format, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_string_to_interval( )

The **mi_string_to_interval( )** function converts a text (string) representation of an interval value to its binary (internal) INTERVAL representation.

Syntax

```
mi_interval *mi_string_to_interval(intvl_string, type_range)
   mi_string *intvl_string;
   mi_string *type_range;
```

intvl_string	is a pointer to the interval string to convert to its internal INTERVAL format.
type_range	is a pointer to a string that specifies the range of INTERVAL qualifiers in the interval string that *intvl_string* references.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_string_to_interval( )** function converts the interval string that *intvl_string* references to its internal INTERVAL value. An internal INTERVAL value is the format that the database server uses to store a value in an INTERVAL column of the database.

The interval string that *intvl_string* references has the following fixed format:

```
"YYYY-MM-DD HH:mm:SS.FFFFF"
```

YYYY	is the 4-digit year.
MM	is the 2-digit month.
DD	is the 2-digit day.
HH	is the 2-digit hour.
mm	is the 2-digit minute.
SS	is the 2-digit second.
FFFFF	is the fraction of a second, in which the date, time, or date and time qualifier specifies the number of digits, with a maximum precision of 5 digits.

However, this interval string can contain only a subset of this range. In this case, the *type_range* argument must specify the range of qualifiers that the interval string contains. This qualifier range begins with the keyword **interval** and is followed by the range of qualifiers for the interval in the *intvl_string*. For example, the following call to **mi_string_to_interval( )** converts the interval of *6 days, 5 hours, and 45 minutes* to its internal INTERVAL format:

```
internal_intvl = mi_string_to_interval(
               "06 5:45",
               "interval day to minute");
```

If the *type_range* argument is a NULL-valued pointer, the **mi_string_to_interval( )** function assumes a default qualifier range of:

```
"interval year to second"
```

Global Language Support

The **mi_string_to_interval( )** function does *not* accept the interval string in the date and time formats of the current processing locale.

End of Global Language Support

Return Values

An **mi_interval** pointer	is a pointer to the internal INTERVAL representation that **mi_string_to_interval( )** has created.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_interval_to_string( ), mi_string_to_date( ), mi_string_to_datetime( ), mi_string_to_decimal( ),** and **mi_string_to_money( ).**

For more information on how to convert interval strings to internal INTERVAL format, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_string_to_lvarchar( )

The **mi_string_to_lvarchar( )** function converts a null-terminated string to a varying-length structure.

Syntax

```
mi_lvarchar *mi_string_to_lvarchar(str)
   mi_string *str;
```

str	is the string to convert to a varying-length structure.

Valid in Client Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_string_to_lvarchar( )** function stores the null-terminated string that *str* references into the data portion of a new varying-length structure. It does not copy the null terminator. This function is a constructor function for a varying-length structure. The function allocates memory for the varying-length structure that it returns. Therefore, you must use the **mi_var_free( )** function to free this structure when it is no longer needed.

Server Only

The **mi_string_to_lvarchar( )** function allocates a new varying-length structure with the current memory duration.

End of Server Only

Return Values

An **mi_lvarchar** pointer	is a pointer to the allocated varying-length structure.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_lvarchar_to_string( ), mi_new_var( ), mi_var_copy( ), mi_var_free( ),** and **mi_var_to_buffer( ).**

mi_string_to_money( )

The **mi_string_to_money( )** function converts a text (string) representation of a monetary value to its binary (internal) MONEY representation.

Syntax

```
mi_money *mi_string_to_money(money_string)
   mi_string *money_string;
```

money_string	is a pointer to the monetary string to convert to its internal MONEY format.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_string_to_money( )** function converts the monetary string that *money_string* references to its internal MONEY value. An internal MONEY value is the format that the database server uses to store a value in a MONEY column of the database. This format represents a fixed-point decimal number.

Important: The **mi_string_to_money( )** function replaces the **mi_money_to_binary( )** function for string-to-internal-MONEY conversion in DataBlade API modules.

Global Language Support

The **mi_string_to_money( )** function accepts the monetary string in the monetary format of the current processing locale. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Return Values

An **mi_money** pointer	is a pointer to the internal MONEY representation that **mi_string_to_money( )** has created.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_money_to_string( ), mi_string_to_date( ), mi_string_to_datetime( ), mi_string_to_decimal( ),** and **mi_string_to_interval( ).**

For more information on how to convert monetary strings to internal MONEY format, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_switch_mem_duration( )

The **mi_switch_mem_duration( )** function switches the current memory duration.

Syntax

```
MI_MEMORY_DURATION mi_switch_mem_duration(duration)
   MI_MEMORY_DURATION duration;
```

duration is a value that specifies the new current memory duration. Valid values for *duration* follow:

PER_ROUTINE	For the duration of one iteration of the UDR
PER_COMMAND	For the duration of the execution of the current subquery
PER_STATEMENT *(Deprecated)*	For the duration of the current SQL statement
PER_STMT_EXEC	For the duration of the *execution* of the current SQL statement
PER_STMT_PREP	For the duration of the current prepared SQL statement
PER_TRANSACTION *(Advanced)*	For the duration of one transaction
PER_SESSION *(Advanced)*	For the duration of the current client session
PER_SYSTEM *(Advanced)*	For the duration of the database server execution

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Valid, but ignored	Yes

Usage

Server Only

The **mi_switch_mem_duration( )** function switches the current memory duration to *duration*. The current memory duration affects the duration of all subsequent memory allocations made by any DataBlade API constructor function that allocates its data structure in the current memory duration. For a list of these constructor functions, see the *IBM Informix DataBlade API Programmer's Guide*. This new current memory duration remains in effect until one of the following actions occurs:

- The user-defined routine completes.
- You change the memory duration again with another call to **mi_switch_mem_duration( )**.

For most memory allocations in a C UDR, the *duration* argument should be one of the following public memory-duration constants:

- PER_ROUTINE (or PER_FUNCTION)

- PER_COMMAND
- PER_STMT_EXEC
- PER_STMT_PREP

Important: Only use a restricted memory duration in your C UDR if a public memory duration will not safely perform the task. These restricted memory durations have long duration times and can increase the possibility of memory leakage.

The **mi_switch_mem_duration( )** function does *not* change the memory duration for the current allocation.

The **mi_switch_mem_duration( )** function returns the previous memory duration. You can save the previous memory duration to switch the memory duration back after performing some allocations. For example, if the memory duration was switched from PER_ROUTINE to PER_COMMAND for a particular purpose, you could invoke **mi_switch_mem_duration( )** to switch back to PER_ROUTINE duration once PER_COMMAND is no longer required. This action could avoid unnecessary memory consumption.

End of Server Only

Client Only

The **mi_switch_mem_duration( )** function has no effect when it is invoked in a client LIBMI application. Client LIBMI applications ignore memory duration.

End of Client Only

Return Values

An **MI_MEMORY_DURATION** constant	is a constant that indicates the previous memory duration.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_alloc( ), mi_dalloc( ), mi_free( ),** and **mi_zalloc( ).**

For more information on memory allocation and memory durations, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_sysname( )

The **mi_sysname( )** f*unction* obtains the name of the default database server.

Syntax

```
char *mi_sysname(sysname)
   char *sysname;
```

sysname	is the name of the system or a NULL-valued pointer.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

Client Only

In a client LIBMI application, the **mi_sysname( )** function obtains or sets the default database server name to *sysname*. The function returns the previous value of the name of the default database server so you can save it and set it back to its original value. The *sysname* value can be set to any of the following values:

- A string that specifies a database server name

  The value in *sysname* must be defined in the server-definition file. Therefore, if the *sysname* value does not match an entry in this file, **mi_sysname( )** returns a NULL-valued pointer.

  UNIX/Linux Only

  On UNIX or Linux, the **sqlhosts** file defines system names.

  End of UNIX/Linux Only

  Windows Only

  On Windows, the Registry defines system names.

  End of Windows Only

- A NULL-valued pointer

  The **mi_sysname( )** function returns the name of the current default database server. If a connection has not been established, there is no current default database server. The **mi_sysname( )** function returns the string `"default"` to indicate that the current system is the default database server, which is initialized to the **INFORMIXSERVER** environment variable.

End of Client Only

Server Only

In a C UDR, the **mi_sysname( )** function accepts only a NULL-valued pointer as *sysname* and returns the current default database server name; that is, the value of the **INFORMIXSERVER** environment variable. You cannot change the value of **INFORMIXSERVER** from within a C UDR.

End of Server Only

The **mi_sysname( )** function initializes the DataBlade API when it is the first DataBlade API function in a client LIBMI application or C UDR routine.

Return Values

A **char** pointer	is a pointer to the current system name.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_close( ), mi_get_next_sysname( ), mi_get_serverenv( ), mi_open( ),** and **mi_server_connect( ).**

mi_system( )

The **mi_system( )** function allows you to execute operating system commands from within a DataBlade module or C UDR.

Syntax

```
mi_integer mi_system(const mi_char *cmd);
```

cmd	is an operating system command, surrounded by quotation marks.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

Use the **mi_system( )** function from within a DataBlade module or C UDR to execute operating system commands, external user executables, or shell scripts. The **mi_system( )** function creates a child process to execute the command, waits for its completion, and then returns the command's exit status to the calling routine. The child process inherits the user and group IDs of the client running the session.

The **mi_system( )** function does not perform any validation on the command you run, therefore, you must take care to run appropriate and accurate commands. See the *IBM Informix DataBlade API Programmer's Guide* for information about good coding practices within user-defined routines.

The following code snippet shows how to use the **mi_system( )** function to execute the **rm /tmp/lo*** command:

```
ret = mi_system("rm /tmp/lo*");
check_retval(ret); /* Check exit status */
```

Windows Only

To use the **mi_system( )** function on Windows® computers, specify the name and absolute pathname of the user executable or file to use within the **mi_system( )** call. For example:

```
 mi_system("/p1/p2/myexe param1 param2")
```

or

```
mi_system("sh /p1/p2/myscript.sh")
```

The **mi_system( )** function treats the first string within the quotation marks as the command and the following strings as parameters passed to the command.

To run a command that includes symbols interpreted by the shell, put them inside a shell script and run the shell script through **mi_system( )**. For example, this command does not work:

```
mi_system("echo 'Check this out' >> /tmp/myfile")
```

because **mi_system( )** treats `echo` as the command to be run and `'Check this out'`, `>>`, and `/tmp/myfile` as its parameters. To successfully run these commands, create a shell script, such as `mysh.sh`, which contains the following lines:

```
#!/bin/sh
echo 'Check this out' >> /tmp/myfile
```

Next, run the following command:

```
mi_system("sh /p1/p2/mysh.sh")
```

End of Windows Only

Return Values

An integer	is the success, failure, or error code returned by the operating system command.
MI_ERROR	indicates that the operating system command could not be executed.

Related Topics

For more information on executing operating system commands, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_td_cast_get( )

The **mi_td_cast_get( )** function looks up a registered cast function by the type descriptors of its source and target data types and creates its function descriptor.

Syntax

```
MI_FUNC_DESC *mi_td_cast_get(conn, source_tdesc, target_tdesc, cast_status)
   MI_CONNECTION *conn;
   MI_TYPE_DESC *source_tdesc;
   MI_TYPE_DESC *target_tdesc;
   mi_char *cast_status;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
	can be a pointer to a session-duration connection descriptor established by a previous call to **mi_get_session_connection( ).** Use of a session-duration connection descriptor is a *restricted* feature of the DataBlade API.
source_tdesc	is a pointer to the type descriptor of the source data type of the cast, including its length and precision.
target_tdesc	is a pointer to the type descriptor of the target data type of the cast.
cast_status	is a pointer to the status flag to set to indicate the kind of cast function that it has located.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_td_cast_get( )** function obtains a function descriptor for a cast function whose source and target data types the *source_tdesc* and *target_tdesc* arguments reference. The **mi_td_cast_get( )** function accepts source and target data types as pointers to type descriptors. This function is one of the functions of the Fastpath interface. It is a constructor function for the function descriptor.

To obtain a function descriptor for a cast function, the **mi_td_cast_get( )** function performs the following tasks:

1. Looks in the **syscasts** system catalog table for the cast function that casts from the *source_tdesc* data type to the *target_tdesc* data type
2. Allocates a function descriptor for the cast function and saves the routine sequence in this descriptor
3. Allocates an **MI_FPARAM** structure for the cast function and saves the argument and return-value information in this structure
4. Sets the *cast_status* output parameter to provide status information about the cast function
5. Returns a pointer to the function descriptor that it allocated for the cast function

Server Only

When you pass a public connection descriptor (from **mi_open( )**), the **mi_td_cast_get( )** function allocates the new function descriptor in the PER_COMMAND memory duration. If you pass a session-duration connection descriptor (from **mi_get_session_connection( )**), **mi_td_cast_get( )** allocates the new function descriptor in the PER_SESSION memory duration. This function descriptor is called a session-duration function descriptor. For more information, see the *IBM Informix DataBlade API Programmer's Guide*.

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. Use them only when a regular connection descriptor or function descriptor will not perform the task you need done.

End of Server Only

The *cast_status* flag can have one of the following constant values.

Cast-Type Constant	**Purpose**
MI_ERROR_CAST	The **mi_td_cast_get( )** function failed.
MI_NO_CAST	A cast does not exist between the two specified types. The user must write a function to perform the cast.
MI_NOP_CAST	A cast is not needed between the two types. The types are equivalent and therefore no cast is required.
MI_SYSTEM_CAST	A built-in cast exists to cast between two data types, usually built-in data types.
MI_EXPLICIT_CAST	A user-defined cast function exists to cast between the two types. The cast is explicit.
MI_IMPLICIT_CAST	A user-defined cast function exists to cast between the two types. The cast is implicit.

The following call to **mi_td_cast_get( )** looks for a cast function that converts INTEGER data to data of an opaque data type named **percent**:

```
MI_TYPE_DESC *src_tdesc, *trgt_tdesc;
MI_CONNECTION *conn;
MI_FUNC_DESC *fdesc;
mi_char cast_stat;
...
src_tdesc = mi_typestring_to_typedesc(conn, "INTEGER");
trgt_tdesc = mi_typestring_to_typedesc(conn, "percent");
fdesc = mi_td_cast_td(conn, src_tdesc, trgt_tdesc,
   &cast_stat);
if ( fdesc == NULL )
   {
   switch ( cast_stat )
      {
      case MI_NO_CAST:
         mi_db_error_raise(NULL, MI_EXCEPTION,
            "No cast function found");
         break;
      case MI_ERROR_CAST:
         mi_db_error_raise(NULL, MI_EXCEPTION,
      "Error in mi_td_cast_get( ) function");
   break;
case MI_NOP_CAST:
   mi_db_error_raise(NULL, MI_EXCEPTION,
      "No cast function needed");
   break;
...
```

Return Values

An **MI_FUNC_DESC** pointer	is a pointer to the function descriptor of the cast function that casts from *source_tdesc* to *target_tdesc*.
NULL	indicates that the *cast_status* value is one of the following constants:

MI_ERROR_CAST	The function was not successful.
MI_NO_CAST	A cast does not exist between the two specified types. The user must write a function to perform the cast.
MI_NOP_CAST	A cast is not needed between the two types. The types are equivalent and therefore no cast is required.

Related Topics

See also the descriptions of **mi_cast_get( ), mi_fparam_get( ), mi_routine_get( ), mi_routine_end( ),** and **mi_routine_exec( ).**

For more information on how to look up a UDR or on type descriptors, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_tracefile_set( )

The **mi_tracefile_set( )** function sets the trace-output file. The database server writes trace messages to this file.

Syntax

```
mi_integer mi_tracefile_set(filename)
   const mi_string *filename;
```

filename	is the pathname of the trace-output file for writing trace messages.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_tracefile_set( )** function sets the trace-output file to the path that *filename* references. The database server writes trace messages to this file when it encounters a trace statement in a UDR and the following conditions are true:

- Tracing for the trace class specified in the trace message is turned on.
- A tracepoint for an active trace class has a threshold less than the current trace level of that class.

For information on how to turn on a trace class, see the description of the **mi_tracelevel_set( )** function.

If you do not call **mi_tracefile_set( )**, the database server writes trace output to a system-defined trace-output file with the session identifier and a **.trc** file extension. You can obtain the current session identifier with the **onstat -g ses** command.

UNIX/Linux Only

On UNIX or Linux, the system-defined trace file resides in the **/tmp** directory with the following filename:

```
session_id.trc
```

In the preceding format, *session_id* is a system-assigned session identifier.

End of UNIX/Linux Only

Windows Only

On Windows, the system-defined trace file has the following pathname:

```
$drive:\tmp\session_id.trc
```

In the pathname, $*drive* is the disk drive on which the IBM Informix product is installed and *session_id* is a system-assigned session identifier.

End of Windows Only

You can change the destination of trace messages with the **mi_tracefile_set( )** function. When you use **mi_tracefile_set( )** to specify the trace-output file, the database server first attempts to append trace messages to that file. If the file that *filename* references does not exist, the database server creates a new trace-output file with the name *filename*. You must ensure that the directory where *filename* resides provides the user **others** or **public** with read and write permission.

The *filename* can specify a full pathname or just a filename. If you specify only a filename, the DataBlade API puts the file in the working directory of the database server environment. This directory is the one in which the **oninit** (or its equivalent) was executed to start up the database server. If **mi_tracefile_set( )** receives an invalid filename, it creates a system-defined trace file.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the description of **mi_tracelevel_set( ).**

mi_tracelevel_set( )

The **mi_tracelevel_set( )** function sets the trace level for specified trace classes.

Syntax

```
mi_integer mi_tracelevel_set(set_commands)
   const mi_string *set_commands;
```

set_commands	is a list of value pairs that specify trace class names and integer trace levels. A space separates each class-name and trace-level pair from the next pair.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_tracelevel_set( )** function sets the tracing levels for the trace classes that you specify in the *set_commands* argument. This argument specifies trace classes and their associated trace levels in the following format:

```
traceclass_name traceclass_level
```

The following example sets the current trace level of trace class **funcEntry** to `50` and the level for trace class **outData** to `35`:

```
mi_integer ret;
...
ret = mi_tracelevel_set("funcEntry 50 outData 35");
```

Important: A trace-class name must be defined in the **systraceclasses** system catalog table *before* you run **mi_tracelevel_set( )**.

By default, tracing is off; that is, the current trace level is set to zero (`0`) for all trace classes. Any nonzero value for a trace level turns tracing on for the specified trace class. A trace level can be any integer from zero (`0`) to the maximum long integer value for the development platform.

If the trace level for the **funcEntry** trace class was currently `50`, the following tracepoint would execute because the value (`10`) in the argument to DPRINTF is not greater than the current level (`50`) of **funcEntry**:

```
DPRINTF("funcEntry", 10,
   ("Entering compute_output with inStr = %s", inStr));
```

The trace message from this DPRINTF call would be written to the current trace-output file. For information on how to set the trace-output file, see the description of the **mi_tracefile_set( )** function.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the description of **mi_tracefile_set( ).**

mi_transaction_state( ) (Server)

The **mi_transaction_state( )** function returns the current transaction state (none, implicit, or explicit).

Syntax

```
mi_integer mi_transaction_state( )
```

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_transaction_state( )** function enables you to base conditional actions on the transaction state. An explicit transaction is any transaction that the BEGIN WORK statement starts. An implicit transaction is any transaction that the database server initiates and the BEGIN WORK statement does not start.

Return Values

MI_NOT_IN_TRANSACTION	No transaction is currently in effect.
MI_IMPLICIT_TRANSACTION	An implicit transaction is currently in effect.
MI_EXPLICIT_TRANSACTION	An explicit transaction is currently in effect.

mi_transition_type( )

The **mi_transition_type( )** accessor function obtains the type of state transition from a transition descriptor.

Syntax

```
MI_TRANSITION_TYPE mi_transition_type(trans_desc)
   MI_TRANSITION_DESC *trans_desc;
```

trans_desc	is a pointer to the transition descriptor that was passed to the callback.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_transition_type( )** function obtains the transition type from the transition descriptor that *trans_desc* references. The DataBlade API provides the following transition-type constants.

State-Transition Type	Description
MI_BEGIN	The database server begins a new transaction.
MI_NORMAL_END	The database server just ended the current transaction by committing the transaction.
MI_ABORT_END	The database server just ended the current transaction by rolling back (aborting) the transaction.

This function is useful within a state-transition or all-events callback to determine the current transition type.

Server Only

In a C UDR routine, the following state-transition events occur at a change in transition type:

- MI_EVENT_SAVEPOINT
- MI_EVENT_COMMIT_ABORT
- MI_EVENT_POST_XACT
- MI_EVENT_END_STMT
- MI_EVENT_END_XACT
- MI_EVENT_END_SESSION

These events occur only for the MI_NORMAL_END and MI_ABORT_END state transitions.

End of Server Only

Client Only

In a client LIBMI application, the state-transition event MI_Xact_State_Change occurs at a change in transition type. This event occurs for all state transitions:

MI_BEGIN, MI_NORMAL_END, and MI_ABORT_END.

End of Client Only

Return Values

MI_BEGIN	indicates that the database server just began a new transaction.
MI_NORMAL_END	indicates that the database server just committed the transaction.
MI_ABORT_END	indicates that the database server just rolled back (aborted) the transaction.
MI_ERROR	indicates that the function was not successful.

Related Topics

For a description of the transition descriptor, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_trigger_event( )

The **mi_trigger_event( )** function returns the trigger event information for the current trigger event.

Syntax

```
mi_integer mi_trigger_event(void)
```

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_trigger_event( )** function returns the value derived from an OR operation from the following events. These values are defined in the public header file. Each bit set in the returned value indicates the type of the trigger currently executing. The returned value is combined with these values to determine the current event. This function can be called in SPL trigger functions and trigger procedures only within the triggered action list of the FOR EACH ROW clause in trigger definitions.

Return Values

MI_ERROR	indicates that the function was not successful
MI_TRIGGER_NOT_IN_EVENT	indicates that the UDR is not currently executing.
MI_TRIGGER_INSERT_EVENT	indicates an INSERT trigger event.
MI_TRIGGER_DELETE_EVENT	indicates a DELETE trigger event.
MI_TRIGGER_UPDATE_EVENT	indicates an UPDATE trigger event.
MI_TRIGGER_SELECT_EVENT	indicates a SELECT trigger event.
MI_TRIGGER_BEFORE_EVENT	indicates a BEFORE trigger event.
MI_TRIGGER_AFTER_EVENT	indicates an AFTER trigger event.
MI_TRIGGER_FOREACH_EVENT	indicates a FOREACH trigger event.
MI_TRIGGER_INSTEAD_EVENT	indicates an INSTEAD OF trigger event. (INSERT,DELETE,UPDATE operations via view)
MI_TRIGGER_REMOTE_EVENT	indicates a remote trigger event.

Related Topics

See also descriptions of **mi_trigger_get_new_row( ), mi_trigger_get_old_row( ), mi_trigger_level( ), mi_trigger_name( ),** and **mi_trigger_tabname( ).**

mi_trigger_get_new_row( )

The **mi_trigger_get_new_row( )** function returns the name of the new row value if the MI_TRIGGER_FOREACH_EVENT bit is set, which implies that neither the MI_TRIGGER_BEFORE_EVENT nor the MI_TRIGGER_AFTER_EVENT bits are set.

Syntax

```
MI_ROW *mi_trigger_get_new_row(void)
```

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

For an INSERT or UPDATE statement, the **mi_trigger_get_new_row( )** function returns the new row being inserted or the updated value of the row. It returns NULL when called in other trigger action statements. This function can be called in SPL trigger functions and trigger procedures only within the triggered action list of the FOR EACH ROW clause in trigger definitions.

Return Values

An **MI_ROW** pointer	is a pointer to the new row.
NULL	indicates the function was one of the following: • not successful. • not in the trigger action. • not in the FOR EACH row trigger. • not in one of the INSERT or UPDATE triggers.

Related Topics

See also descriptions of **mi_trigger_event( ), mi_trigger_get_old_row( ), mi_trigger_level( ), mi_trigger_name( ),** and **mi_trigger_tabname( ).**

mi_trigger_get_old_row( )

The **mi_trigger_get_old_row( )** function returns the name of the old row value if the MI_TRIGGER_FOREACH_EVENT bit is set, which implies that neither the MI_TRIGGER_BEFORE_EVENT nor the MI_TRIGGER_AFTER_EVENT bits are set. In addition, **mi_trigger_get_old_row( )** 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.

Syntax

```
MI_ROW *mi_trigger_get_old_row(void)
```

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

For a DELETE or UPDATE statement, the **mi_trigger_get_old_row( )** function returns the row that was deleted or the value of the row before it was updated. It returns NULL when called in other trigger action statements. This function can be called in SPL trigger functions and trigger procedures only within the triggered action list of the FOR EACH ROW clause in trigger definitions.

Return Values

An **MI_ROW** pointer	is a pointer to the old row.
NULL	indicates the function was one of the following: • not successful. • not in the trigger action. • not in the FOR EACH row trigger. • not in one of the DELETE or UPDATE triggers.

Related Topics

See also descriptions of **mi_trigger_event( ), mi_trigger_get_new_row( ), mi_trigger_level( ), mi_trigger_name( ),** and **mi_trigger_tabname( ).**

mi_trigger_level( )

The **mi_trigger_level( )** function returns the nesting level value of the current trigger.

Syntax

```
mi_integer mi_trigger_level(void)
```

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_trigger_level( )** function 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. This function can be called in SPL trigger functions and trigger procedures only within the triggered action list of the FOR EACH ROW clause in trigger definitions.

Return Values

0	indicates that the UDR is currently not executing as a part of trigger action.
1-61	indicates the value of the nesting level of the current trigger.

Related Topics

See also descriptions of **mi_trigger_event( ), mi_trigger_get_new_row( ), mi_trigger_get_old_row( ), mi_trigger_name( ),** and **mi_trigger_tabname( ).**

mi_trigger_name( )

The **mi_trigger_name( )** function returns the name of the currently executing trigger.

Syntax

```
mi_string *mi_trigger_name(void)
```

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_trigger_name( )** function queries the database server for the table on which the trigger is being is executed and returns the values of the owner name and the trigger name. This function can be called in SPL trigger functions and trigger procedures only within the triggered action list of the FOR EACH ROW clause in trigger definitions.

Return Values

ownername.triggername	is the name of the currently executing trigger.
NULL	indicates that the UDR is currently not executing as a part of trigger action.

Related Topics

See also descriptions of **mi_trigger_event( ), mi_trigger_get_new_row( ), mi_trigger_get_old_row( ), mi_trigger_level( ),** and **mi_trigger_tabname( ).**

mi_trigger_tabname( )

The **mi_trigger_tabname( )** function returns the triggering table or view name on which the current trigger action statement was invoked.

Syntax

```
mi_string *mi_trigger_tabname(mi_integer flags)
```

flags The *flags* parameter is a combination of the values described below:

The following two flags indicate the table information.

MI_TRIGGER_CURRENTTABLE
Return names associated with the current table.

MI_TRIGGER_TOPTABLE
Return names associated with top table.

The following five flags indicate what information about the table the API returns.

MI_TRIGGER_TABLENAME
Include the table name.

MI_TRIGGER_OWNERNAME
Include the owner name.

MI_TRIGGER_DBASENAME
Include the database name.

MI_TRIGGER_SERVERNAME
Include the server name.

MI_TRIGGER_FULLNAME
Shorthand for all four MI_TRIGGER_*NAME values combined with an OR operation.

If neither MI_TRIGGER_CURRENTTABLE nor MI_TRIGGER_TOPTABLE is defined, you receive the information for the current triggering table. If both are defined, you receive the information for the top table. You can specify any combination of table, owner, database, and server name and the result contains the appropriate punctuation.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_trigger_tabname( )** function returns the table or view name on which the current trigger action statement was invoked, By using the parameters defined above, you can control the table for which the information is returned and the content of the information returned.

The output value is not safe for embedding into an SQL statement. The components are the names as stored in the system catalog, and may need to be treated specially to be valid in an SQL statement. For example, if the name contains uppercase letters or non-alphanumeric characters (underscore counts as alphanumeric), you would have to have DELIMIDENT set in the environment and you would need to enclose the name in double quotation marks. If the name contains double quotation marks, they would have to be repeated. (Note that the

UNIX system call **open( )** provides a precedent for this type of combined flag; the MI_O_RDONLY, MI_ O_WRONLY and MI_O_RDWR flags are mutually exclusive, but one of them can be combined with various other flags such as MI_O_CREAT or MI_O_EXCL.)

In a nested trigger scenario, by default, the current trigger table name is returned. You can get the top triggering table information by specifying MI_TOPTABLE. If you specify MI_TOPTABLE in a single-level trigger, you get the current triggering table name.

This function can be called in SPL trigger functions and trigger procedures only within the triggered action list of the FOR EACH ROW clause in trigger definitions.

Return Values

database@server:owner.tabname
: is the full table name (depending on the values in the *flags* parameter).

database@server:owner.viewname
: is the full view name (depending on the values in the *flags* parameter).

Related Topics

See also descriptions of **mi_trigger_event( ), mi_trigger_get_new_row( ), mi_trigger_get_old_row( ), mi_trigger_level( ),** and **mi_trigger_name( ).**

mi_try_lock_memory( )

The **mi_try_lock_memory( )** function requests a lock on a named-memory block specified by name and memory duration.

Syntax

```
mi_integer mi_try_lock_memory(mem_name, duration)
   mi_string *mem_name;
   MI_MEMORY_DURATION duration;
```

mem_name is the null-terminated name of the named-memory block to lock.

duration is a value that specifies the memory duration of the named-memory block to lock. Valid values for *duration* follow:

PER_ROUTINE	For the duration of one iteration of the UDR
PER_COMMAND	For the duration of the execution of the current subquery
PER_STATEMENT *(Deprecated)*	For the duration of the current SQL statement
PER_STMT_EXEC	For the duration of the *execution* of the current SQL statement
PER_STMT_PREP	For the duration of the current prepared SQL statement
PER_TRANSACTION	For the duration of one transaction
PER_SESSION	For the duration of the current client session
PER_SYSTEM	For the duration of the database server execution

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Warning: This advanced function can adversely affect your UDR if you use the function incorrectly. Use it only when no regular DataBlade API function can perform the task you need done.

Usage

The **mi_try_lock_memory( )** function requests a lock on the named-memory block based on its memory duration of *duration* and its name, which *mem_name* references. The function does *not* wait until this lock has been obtained before it returns control to its calling function. If some other process currently holds a lock on the memory, **mi_try_lock_memory( )** returns a status of MI_LOCK_IS_BUSY. The calling code can call **mi_try_lock_memory( )** in a loop until the function returns the MI_OK status.

Important: After you obtain a lock on a named-memory block, release it as soon as possible. You must explicitly release a named-memory lock with the **mi_unlock_memory( )** function.

Return Values

MI_OK	indicates that the function successfully locked the specified named-memory block.
MI_NO_SUCH_NAME	indicates that the requested named-memory block does not exist.
MI_LOCK_IS_BUSY	indicates that acquisition of a lock on the specified named-memory block failed because it is already locked by another process.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_lock_memory( ), mi_named_alloc( ), mi_named_get( ), mi_named_zalloc( ),** and **mi_unlock_memory( )**..

mi_type_align( )

The **mi_type_align( )** function obtains the alignment for a data type from its type descriptor.

Syntax

```
mi_integer mi_type_align(type_desc)
   MI_TYPE_DESC *type_desc;
```

type_desc	is a pointer to the type descriptor from which to obtain the alignment of the data type.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_type_align( )** function determines the alignment for the data type that the *type_desc* type descriptor references. The default type alignment is 4 bytes. For extended data types, the alignment is the value of the **align** column of the **sysxtdtypes** system catalog table.

Return Values

>=0	is the number of bytes for type alignment of the data type in the *type_desc* type descriptor.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_type_byvalue( ), mi_type_constructor_typedesc( ), mi_type_element_typedesc( ), mi_type_full_name( ), mi_type_length( ), mi_type_maxlength( ), mi_type_owner( ), mi_type_precision( ), mi_type_qualifier( ), mi_type_scale( ), mi_type_typedesc( ), mi_type_typename( ),** and **mi_typedesc_typeid( ).**

mi_type_byvalue( )

The **mi_type_byvalue( )** function checks a type descriptor to determine whether a data type is passed by value or by reference.

Syntax

```
mi_boolean mi_type_byvalue(type_desc)
   MI_TYPE_DESC *type_desc;
```

type_desc	is a pointer to the type descriptor from which to determine the passing mechanism of a data type.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_type_byvalue( )** function determines the passing mechanism for the data type that *type_desc* references. For extended data types, the value that **mi_type_byvalue( )** returns comes from the **byvalue** column of the **sysxtdtypes** system catalog table.

Use this function to determine whether a particular **MI_DATUM** value (such as a routine argument or return value) is passed by value or by reference.

Return Values

MI_TRUE	indicates that the data type is passed by value.
MI_FALSE	indicates that the data type is *not* passed by value; it is passed by reference.

Related Topics

See also the descriptions of **mi_type_align( ), mi_type_constructor_typedesc( ), mi_type_element_typedesc( ), mi_type_full_name( ), mi_type_length( ), mi_type_maxlength( ), mi_type_owner( ), mi_type_precision( ), mi_type_qualifier( ), mi_type_scale( ), mi_type_typedesc( ), mi_type_typename( ),** and **mi_typedesc_typeid( ).**

mi_type_constructor_typedesc( )

The **mi_type_constructor_typedesc( )** function obtains a type descriptor for a constructed data type.

Syntax

```
MI_TYPE_DESC *mi_type_constructor_typedesc(type_desc)
   MI_TYPE_DESC *type_desc;
```

type_desc	is a pointer to the type descriptor from which to determine the passing mechanism of a data type.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

If the type descriptor that *type_desc* references is a constructor, **mi_type_constructor_typedesc( )** returns a pointer to the *type_desc* type descriptor. If the *type_desc* type descriptor is a constructed type, this function returns a pointer to the type descriptor for the constructed type. If the *type_desc* type descriptor is not a constructed data type, **mi_type_constructor_typedesc( )** returns a NULL-valued pointer.

Related Topics

See also the descriptions of **mi_type_align( ), mi_type_byvalue( ), mi_type_element_typedesc( ), mi_type_full_name( ), mi_type_length( ), mi_type_maxlength( ), mi_type_owner( ), mi_type_precision( ), mi_type_qualifier( ), mi_type_scale( ), mi_type_typedesc( ), mi_type_typename( ),** and **mi_typedesc_typeid( ).**

mi_type_element_typedesc( )

The **mi_type_element_typedesc( )** function obtains the type descriptor for the elements of a collection data type from its type descriptor.

Syntax

```
MI_TYPE_DESC *mi_type_element_typedesc(type_desc)
   MI_TYPE_DESC *type_desc;
```

type_desc	is a pointer to the type descriptor from which to obtain the element type of the collection data type.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

Valid collection data types are LIST, SET, and MULTISET. For example, this function could tell you that a collection type is a collection of integers.

Return Values

An **MI_TYPE_DESC** pointer	is a pointer to the type descriptor for the element type of the collection data type.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_type_align( ), mi_type_byvalue( ), mi_type_constructor_typedesc( ), mi_type_full_name( ), mi_type_length( ), mi_type_maxlength( ), mi_type_owner( ), mi_type_precision( ), mi_type_qualifier( ), mi_type_scale( ), mi_type_typedesc( ), mi_type_typename( ),** and **mi_typedesc_typeid( ).**

mi_type_full_name( )

The **mi_type_full_name( )** function obtains the full name (*owner.typename*) of a data type from its type descriptor.

Syntax

```
mi_string *mi_type_full_name(type_desc)
   MI_TYPE_DESC *type_desc;
```

type_desc	is a pointer to the type descriptor from which to obtain the full name of the data type.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_type_full_name( )** function determines the full name for the data type that *type_desc* references.

Return Values

An **mi_string** pointer	is a pointer to the full name of the data type in the *type_desc* type descriptor.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_type_align( ), mi_type_byvalue( ), mi_type_constructor_typedesc( ), mi_type_element_typedesc( ), mi_type_length( ), mi_type_maxlength( ), mi_type_owner( ), mi_type_precision( ), mi_type_qualifier( ), mi_type_scale( ), mi_type_typedesc( ), mi_type_typename( ),** and **mi_typedesc_typeid( ).**

mi_type_length( )

The **mi_type_length( )** function obtains the length of a data type from its type descriptor.

Syntax

```
mi_integer mi_type_length(type_desc)
   MI_TYPE_DESC *type_desc;
```

type_desc	is a pointer to the type descriptor from which to obtain the length.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_type_length( )** function determines the length for the data type that *type_desc* references. The length of the type can exceed the length of the data in the type. For extended data types, the length is the value of the **length** column of the **sysxtdtypes** system catalog table.

Return Values

>=0	is the length of the data type in the *type_desc* type descriptor.
MI_ERROR	indicates that the data type in *type_desc* is CHAR or NCHAR.

Related Topics

See also the descriptions of **mi_type_align( ), mi_type_byvalue( ), mi_type_constructor_typedesc( ), mi_type_element_typedesc( ), mi_type_full_name( ), mi_type_maxlength( ), mi_type_owner( ), mi_type_precision( ), mi_type_qualifier( ), mi_type_scale( ), mi_type_typedesc( ), mi_type_typename( ),** and **mi_typedesc_typeid( ).**

mi_type_maxlength( )

The **mi_type_maxlength( )** function obtains the maximum length of a data type from its type descriptor.

Syntax

```
mi_integer mi_type_maxlength(type_desc)
   MI_TYPE_DESC *type_desc;
```

type_desc	is a pointer to the type descriptor from which to obtain the maximum length.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_type_maxlength( )** function obtains the maximum length of the data type from the type descriptor that *type_desc* references. The maximum length applies to the built-in data types VARCHAR and LVARCHAR. For extended data types, the maximum length is the value of the **maxlen** column of the **sysxtdtypes** system catalog table.

Global Language Support

The NVARCHAR data type also has a maximum length.

End of Global Language Support

If you call **mi_type_maxlength( )** on some other data type, the function returns zero (0).

Return Values

>=0	is the maximum length of the data type in the *type_desc* type descriptor.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_type_align( ), mi_type_byvalue( ), mi_type_constructor_typedesc( ), mi_type_element_typedesc( ), mi_type_full_name( ), mi_type_length( ), mi_type_owner( ), mi_type_precision( ), mi_type_qualifier( ), mi_type_scale( ), mi_type_typedesc( ), mi_type_typename( ),** and **mi_typedesc_typeid( ).**

For more information, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_type_owner( )

The **mi_type_owner( )** function obtains the owner of a data type from its type descriptor.

Syntax

```
mi_string *mi_type_owner(type_desc)
   MI_TYPE_DESC *type_desc;
```

type_desc	is a pointer to the type descriptor from which to obtain the owner name.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_type_owner( )** function obtains the owner of the data type from the type descriptor that *type_desc* references. For extended data types, the owner is the value of the **owner** column of the **sysxtdtypes** system catalog table.

Return Values

An **mi_string** pointer	is a pointer to the owner name of the data type in the *type_desc* type descriptor.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_type_align( ), mi_type_byvalue( ), mi_type_constructor_typedesc( ), mi_type_element_typedesc( ), mi_type_full_name( ), mi_type_length( ), mi_type_maxlength( ), mi_type_precision( ), mi_type_qualifier( ), mi_type_scale( ), mi_type_typedesc( ), mi_type_typename( ),** and **mi_typedesc_typeid( ).**

mi_type_precision( )

The **mi_type_precision( )** function obtains the precision of a data type from its type descriptor.

Syntax

```
mi_integer mi_type_precision(type_desc)
   MI_TYPE_DESC *type_desc;
```

type_desc	is a pointer to the type descriptor from which to obtain the precision.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_type_precision( )** function obtains the data type precision from the type descriptor that *type_desc* references. The precision is an attribute of the data type that represents the total number of digits the data type can hold, as follows.

Data Type	Meaning
DECIMAL, MONEY	Number of significant digits in the fixed-point or floating-point (DECIMAL) column
DATETIME, INTERVAL	Number of digits that are stored in the date, time, or date and time column with a specified qualifier
Character, Varying-character	Maximum number of characters in the column

If you call **mi_type_precision( )** on some other data type, the function returns zero (0).

Return Values

>=0	is the precision of the data type in *type_desc*.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_type_align( ), mi_type_byvalue( ), mi_type_constructor_typedesc( ), mi_type_element_typedesc( ), mi_type_full_name( ), mi_type_length( ), mi_type_maxlength( ), mi_type_owner( ), mi_type_qualifier( ), mi_type_scale( ), mi_type_typedesc( ), mi_type_typename( ),** and **mi_typedesc_typeid( ).**

For information on type-descriptor accessor functions or on precision and scale, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_type_qualifier( )

The **mi_type_qualifier( )** function obtains the qualifier of a data type from its type descriptor.

Syntax

```
mi_integer mi_type_qualifier(type_desc)
   MI_TYPE_DESC *type_desc;
```

type_desc	is a pointer to the type descriptor from which to obtain the qualifier.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_type_qualifier( )** function obtains the data type qualifier from the type descriptor that *type_desc* references. The qualifier is the number of time increments to store for a date and/or time data type: DATETIME and INTERVAL. If you call **mi_type_qualifier( )** on some other data type, the function returns `-1`.

This function returns an encoded integer value that the database server uses internally to represent a qualifier. You can use the qualifier constants and macros (such as TU_START and TU_END) to use this encoded value.

Return Values

>=0	is the encoded integer qualifier of the data type in *type_desc*.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_type_align( ), mi_type_byvalue( ), mi_type_constructor_typedesc( ), mi_type_element_typedesc( ), mi_type_full_name( ), mi_type_length( ), mi_type_maxlength( ), mi_type_owner( ), mi_type_precision( ), mi_type_scale( ), mi_type_typedesc( ), mi_type_typename( ),** and **mi_typedesc_typeid( ).**

For information on type-descriptor accessor functions or on qualifiers, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_type_scale( )

The **mi_type_scale( )** function obtains the scale of the data type from its type descriptor.

Syntax

```
mi_integer mi_type_scale(type_desc)
   MI_TYPE_DESC *type_desc;
```

type_desc	is a pointer to the type descriptor from which to obtain the scale.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_type_scale( )** function obtains the data type scale from the type descriptor that *type_desc* references. The scale is an attribute of the data type. The meaning of the scale depends on the associated data type, as the following table shows.

Data Type	Meaning of Scale
DECIMAL (fixed-point), MONEY	The number of digits to the right of the decimal point
DECIMAL (floating-point)	The value 255
DATETIME, INTERVAL	The encoded integer value for the end qualifier of the data type; *end_qual* in the qualifier: *start_qual* TO *end_qual*

If you call **mi_type_scale( )** on some other data type, the function returns zero (0).

Return Values

>=0	is the scale of the data type in *type_desc*.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_type_align( ), mi_type_byvalue( ), mi_type_constructor_typedesc( ), mi_type_element_typedesc( ), mi_type_full_name( ), mi_type_length( ), mi_type_maxlength( ), mi_type_owner( ), mi_type_precision( ), mi_type_qualifier( ), mi_type_typedesc( ), mi_type_typename( ),** and **mi_typedesc_typeid( ).**

For information on type-descriptor accessor functions or on precision and scale, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_type_typedesc( )

The **mi_type_typedesc( )** function creates a type descriptor, given a type identifier.

Syntax

```
MI_TYPE_DESC *mi_typedesc (conn, typeid_ptr)
   MI_CONNECTION *conn;
   MI_TYPEID *typeid_ptr;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
typeid_ptr	is a pointer to the type identifier for which to generate a type descriptor.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_type_typedesc( )** function obtains the type descriptor for the type identifier that *typeid_ptr* references.

Return Values

An **MI_TYPE_DESC** pointer	is a pointer to the type descriptor for *typeid_ptr*.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_type_align( ), mi_type_byvalue( ), mi_type_constructor_typedesc( ), mi_type_element_typedesc( ), mi_type_full_name( ), mi_type_length( ), mi_type_maxlength( ), mi_type_owner( ), mi_type_precision( ), mi_type_qualifier( ), mi_type_scale( ), mi_type_typename( ),** and **mi_typedesc_typeid( ).**

mi_type_typename( )

The **mi_type_typename( )** function obtains the name of a data type from its type descriptor.

Syntax

```
mi_string *mi_type_typename(type_desc)
   MI_TYPE_DESC *type_desc;
```

type_desc	is a pointer to the type descriptor from which to obtain the name.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_type_typename( )** function obtains the data type name from the type descriptor that *type_desc* references. For extended data types, the name is the value of the **name** column of the **sysxtdtypes** system catalog table.

Return Values

An **mi_string** pointer	is the name of the data type in *type_desc*.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_type_align( ), mi_type_byvalue( ), mi_type_constructor_typedesc( ), mi_type_element_typedesc( ), mi_type_full_name( ), mi_type_length( ), mi_type_maxlength( ), mi_type_owner( ), mi_type_precision( ), mi_type_qualifier( ), mi_type_scale( ), mi_type_typedesc( ),** and **mi_typedesc_typeid( ).**

mi_typedesc_typeid( )

The **mi_typedesc_typeid( )** function obtains the type identifier of a data type from its type descriptor.

Syntax

```
MI_TYPEID *mi_typedesc_typeid(type_desc)
   MI_TYPE_DESC *type_desc;
```

type_desc	is a pointer to the type descriptor from which to obtain the type identifier.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Return Values

An **MI_TYPEID** pointer	is a pointer to the type identifier of the data type in *type_desc*.
NULL	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_type_typedesc( ), mi_typename_to_id( ),** and **mi_typestring_to_id( ).**

mi_typeid_equals( )

The **mi_typeid_equals( )** function determines whether two type identifiers represent the same data type.

Syntax

```
mi_boolean mi_typeid_equals(typeid1_ptr, typeid2_ptr)
   MI_TYPEID *typeid1_ptr;
   MI_TYPEID *typeid2_ptr;
```

typeid1_ptr	is a pointer to the type identifier of the first data type.
typeid2_ptr	is a pointer to the type identifier of the second data type.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_typeid_equals( )** function determines if the data types in the type identifiers that *typeid1_ptr* and *typeid2_ptr* reference are the same.

Important: The type identifier is an opaque structure. Do not compare two type identifiers directly. Instead, use **mi_typeid_equals( )** to determine if two type identifiers are equal.

Return Values

MI_TRUE	indicates that the two type identifiers are for the same data type.
MI_FALSE	indicates that the two type identifiers are *not* for the same data type.

Related Topics

See also the descriptions of **mi_typeid_is_builtin( ), mi_typeid_is_collection( ), mi_typeid_is_complex( ), mi_typeid_is_distinct( ), mi_typeid_is_list( ), mi_typeid_is_multiset( ), mi_typeid_is_row( ),** and **mi_typeid_is_set( ).**

mi_typeid_is_builtin( )

The **mi_typeid_is_builtin( )** function determines whether a type identifier is for a built-in data type.

Syntax

```
mi_boolean mi_typeid_is_builtin(typeid_ptr)
    MI_TYPEID *typeid_ptr;
```

typeid_ptr	is a pointer to the type identifier to check.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_typeid_is_builtin( )** function determines if the data type in the type identifier that *typeid_ptr* references is a built-in data type.

Important: The type identifier is an opaque structure. Do not access its value directly. To determine if a type identifier contains a built-in data type, always use **mi_typeid_is_builtin( )**.

Return Values

MI_TRUE	indicates that the type identifier that *typeid_ptr* references is a built-in type.
MI_FALSE	indicates that the type identifier that *typeid_ptr* references is *not* a built-in type.

Related Topics

See also the descriptions of **mi_typeid_equals( ), mi_typeid_is_collection( ), mi_typeid_is_complex( ), mi_typeid_is_distinct( ), mi_typeid_is_list( ), mi_typeid_is_multiset( ), mi_typeid_is_row( ),** and **mi_typeid_is_set( ).**

mi_typeid_is_collection( )

The **mi_typeid_is_collection( )** function determines whether a type identifier is for a collection.

Syntax

```
mi_boolean mi_typeid_is_collection(typeid_ptr)
   MI_TYPEID *typeid_ptr;
```

typeid_ptr	is a pointer to the type identifier to check.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_typeid_is_collection( )** function determines if the data type in the type identifier that *typeid_ptr* references is a collection data type. Valid collection data types are SET, MULTISET, and LIST.

Important: The type identifier is an opaque structure. Do not access its value directly. To determine if a type identifier contains a collection data type, always use **mi_typeid_is_collection( )**.

Return Values

MI_TRUE	indicates that the type identifier that *typeid_ptr* references is a collection data type.
MI_FALSE	indicates that the type identifier that *typeid_ptr* references is *not* a collection data type.

Related Topics

See also the descriptions of **mi_typeid_equals( ), mi_typeid_is_builtin( ), mi_typeid_is_complex( ), mi_typeid_is_distinct( ), mi_typeid_is_list( ), mi_typeid_is_multiset( ), mi_typeid_is_row( ),** and **mi_typeid_is_set( ).**

mi_typeid_is_complex( )

The **mi_typeid_is_complex( )** function determines whether a type identifier is for a complex data type.

Syntax

```
mi_boolean mi_typeid_is_complex(typeid_ptr)
   MI_TYPEID *typeid_ptr;
```

typeid_ptr	is a pointer to the type identifier to check.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_typeid_is_complex( )** function determines if the data type in the type identifier that *typeid_ptr* references is a complex data type. Valid complex data types are collection data types (SET, MULTISET, and LIST) and row types (named and unnamed).

Important: The type identifier is an opaque structure. Do not access its value directly. To determine if a type identifier contains a complex data type, always use **mi_typeid_is_complex( )**.

Return Values

MI_TRUE	indicates that the type identifier that *typeid_ptr* references is a complex type.
MI_FALSE	indicates that the type identifier that *typeid_ptr* references is *not* a complex type.

Related Topics

See also the descriptions of **mi_typeid_equals( ), mi_typeid_is_builtin( ), mi_typeid_is_collection( ), mi_typeid_is_distinct( ), mi_typeid_is_list( ), mi_typeid_is_multiset( ), mi_typeid_is_row( ),** and **mi_typeid_is_set( ).**

mi_typeid_is_distinct( )

The **mi_typeid_is_distinct( )** function determines whether a type identifier is for a distinct data type.

Syntax

```
mi_boolean mi_typeid_is_distinct(typeid_ptr)
   MI_TYPEID *typeid_ptr;
```

typeid_ptr	is a pointer to the type identifier to check.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_typeid_is_distinct( )** function determines if the data type in the type identifier that *typeid_ptr* references is a distinct data type.

Important: The type identifier is an opaque structure. Do not access its value directly. To determine if a type identifier contains a distinct data type, always use **mi_typeid_is_distinct( )**.

Return Values

MI_TRUE	indicates that the type identifier that *typeid_ptr* references is a distinct data type.
MI_FALSE	indicates that the type identifier that *typeid_ptr* references is *not* a distinct data type.

Related Topics

See also the descriptions of, **mi_get_transaction_id( ), mi_typeid_equals( ), mi_typeid_is_builtin( ), mi_typeid_is_collection( ), mi_typeid_is_complex( ), mi_typeid_is_list( ), mi_typeid_is_multiset( ), mi_typeid_is_row( ),** and **mi_typeid_is_set( ).**

mi_typeid_is_list( )

The **mi_typeid_is_list( )** function determines whether a type identifier is for a LIST collection data type.

Syntax

```
mi_boolean mi_typeid_is_list(typeid_ptr)
   MI_TYPEID *typeid_ptr;
```

typeid_ptr	is a pointer to the type identifier to check.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_typeid_is_list( )** function determines if the data type in the type identifier that *typeid_ptr* references is a LIST collection data type.

Important: The type identifier is an opaque structure. Do not access its value directly. To determine if a type identifier contains a LIST data type, always use **mi_typeid_is_list( )**. To determine if a type identifier contains any collection data type, including LIST, use **mi_typeid_is_collection( ).**

Return Values

MI_TRUE	indicates that the type identifier that *typeid_ptr* references is a LIST data type.
MI_FALSE	indicates that the type identifier that *typeid_ptr* references is *not* a LIST data type.

Related Topics

See also the descriptions of **mi_typeid_equals( ), mi_typeid_is_builtin( ), mi_typeid_is_collection( ), mi_typeid_is_complex( ), mi_typeid_is_distinct( ), mi_typeid_is_multiset( ), mi_typeid_is_row( ),** and **mi_typeid_is_set( ).**

mi_typeid_is_multiset( )

The **mi_typeid_is_multiset( )** function determines whether a type identifier is for a MULTISET collection data type.

Syntax

```
mi_boolean mi_typeid_is_multiset(typeid_ptr)
   MI_TYPEID *typeid_ptr;
```

typeid_ptr	is a pointer to the type identifier to check.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_typeid_is_multiset( )** function determines if the data type in the type identifier that *typeid_ptr* references is a MULTISET collection data type.

Important: The type identifier is an opaque structure. Do not access its value directly. To determine if a type identifier contains a MULTISET data type, always use **mi_typeid_is_multiset( )**. To determine if a type identifier contains any collection data type, including MULTISET, use **mi_typeid_is_collection( ).**

Return Values

MI_TRUE	indicates that the type identifier that *typeid_ptr* references is a MULTISET data type.
MI_FALSE	indicates that the type identifier that *typeid_ptr* references is *not* a MULTISET data type.

Related Topics

See also the descriptions of **mi_typeid_equals( ), mi_typeid_is_builtin( ), mi_typeid_is_collection( ), mi_typeid_is_complex( ), mi_typeid_is_distinct( ), mi_typeid_is_list( ), mi_typeid_is_row( ),** and **mi_typeid_is_set( ).**

mi_typeid_is_row( )

The **mi_typeid_is_row( )** function determines whether a type identifier is for an SQL row data type.

Syntax

```
mi_boolean mi_typeid_is_row(typeid_ptr)
   MI_TYPEID *typeid_ptr;
```

typeid_ptr	is a pointer to the type identifier to check.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_typeid_is_row( )** function determines if the data type in the type identifier that *typeid_ptr* references is an SQL row data type:

- Named row type: *named_row* name
- Unnamed row type: ROW

Important: The type identifier is an opaque structure. Do not access its value directly. To determine if a type identifier contains a ROW data type, always use **mi_typeid_is_row( )**. To determine if a type identifier contains any complex data type, use **mi_typeid_is_complex( ).**

Return Values

MI_TRUE	indicates that the type identifier that *typeid_ptr* references is a row type.
MI_FALSE	indicates that the type identifier that *typeid_ptr* references is *not* a row data type.

Related Topics

See also the descriptions of **mi_typeid_equals( ), mi_typeid_is_builtin( ), mi_typeid_is_collection( ), mi_typeid_is_complex( ), mi_typeid_is_distinct( ), mi_typeid_is_list( ), mi_typeid_is_multiset( ),** and **mi_typeid_is_set( ).**

mi_typeid_is_set( )

The **mi_typeid_is_set( )** function determines whether a type identifier is for a SET collection data type.

Syntax

```
mi_boolean mi_typeid_is_set(typeid_ptr)
   MI_TYPEID *typeid_ptr;
```

typeid_ptr	is a pointer to the type identifier to check.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_typeid_is_set( )** function determines if the data type in the type identifier that *typeid_ptr* references is a SET collection data type.

Important: The type identifier is an opaque structure. Do not access its value directly. To determine if a type identifier contains a SET data type, always use **mi_typeid_is_set( )**. To determine if a type identifier contains any collection data type, including SET, use **mi_typeid_is_collection( ).**

Return Values

MI_TRUE	indicates that the type identifier that *typeid_ptr* references is a SET data type.
MI_FALSE	indicates that the type identifier that *typeid_ptr* references is *not* a SET data type.

Related Topics

See also the descriptions of **mi_typeid_equals( ), mi_typeid_is_builtin( ), mi_typeid_is_collection( ), mi_typeid_is_complex( ), mi_typeid_is_distinct( ), mi_typeid_is_list( ), mi_typeid_is_multiset( ),** and **mi_typeid_is_row( ).**

mi_typename_to_id( )

The **mi_typename_to_id( )** function creates a type identifier for a data type, given the type name in LVARCHAR format.

Syntax

```
MI_TYPEID *mi_typename_to_id(conn, type_lvname)
   MI_CONNECTION *conn;
   mi_lvarchar *type_lvname;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
type_lvname	is the name of the SQL data type, in LVARCHAR format.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

The **mi_typename_to_id( )** function converts the name of the data type in the varying-length structure that *type_lvname* references into a type identifier. For a list of SQL data type names, see the *IBM Informix DataBlade API Programmer's Guide*.

Return Values

An **MI_TYPEID** pointer	is a pointer to the type identifier of the *type_lvname* data type.
NULL	indicates that the function was not successful; a type identifier for *type_lvname* was not found.

Related Topics

See also the descriptions of **mi_typedesc_typeid( ), mi_typename_to_typedesc( ),** and **mi_typestring_to_id( ).**

mi_typename_to_typedesc( )

The **mi_typename_to_typedesc( )** function creates a type descriptor for a data type, given the type name in LVARCHAR format.

Syntax

```
MI_TYPE_DESC *mi_typename_to_typedesc(conn, type_lvname)
   MI_CONNECTION *conn;
   mi_lvarchar *type_lvname;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
type_lvname	is the name of the SQL data type, in LVARCHAR format.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_typename_to_typedesc( )** function converts the name of the data type in the varying-length structure that *type_lvname* references into a type descriptor. For a list of SQL data type names, see the *IBM Informix DataBlade API Programmer's Guide*.

Return Values

An **MI_TYPE_DESC** pointer	is a pointer to the type descriptor for the *type_lvname* data type.
NULL	indicates that the function was not successful; the *type_lvname* data type was not found.

Related Topics

See also the descriptions of **mi_type_typedesc( ), mi_typename_to_id( ),** and **mi_typestring_to_typedesc( ).**

mi_typestring_to_id( )

The **mi_typestring_to_id( )** function creates a type identifier for a data type, given the type name as a null-terminated string.

Syntax

```
MI_TYPEID *mi_typestring_to_id(conn, type_name)
   MI_CONNECTION *conn;
   mi_string *type_name;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
type_name	is the name of the SQL data type. It can be in the form *owner*.*type_name* or simply *type_name*.

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

The **mi_typestring_to_id( )** function converts the name of the data type in the null-terminated string that *type_name* references into a type identifier. For a list of SQL data type names, see the *IBM Informix DataBlade API Programmer's Guide*.

Return Values

An **MI_TYPEID** pointer	is a pointer to the type identifier for the *type_name* data type.
NULL	indicates that the function was not successful; the *type_name* data type was not found.

Related Topics

See also the descriptions of **mi_typedesc_typeid( ), mi_typename_to_id( ),** and **mi_typestring_to_typedesc( ).**

mi_typestring_to_typedesc( )

The **mi_typestring_to_typedesc( )** function creates a type descriptor for a data type, given the type name as a null-terminated string.

Syntax

```
MI_TYPE_DESC *mi_typestring_to_typedesc(conn, type_name)
   MI_CONNECTION *conn;
   mi_string *type_name;
```

conn	is a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
type_name	is the name of the SQL data type. It can be in the form *owner*.*type_name* or simply *type_name*.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_typestring_to_typedesc( )** function converts the name of the data type in the null-terminated string that *type_name* references into a type descriptor. For a list of SQL data type names, see the *IBM Informix DataBlade API Programmer's Guide*.

Return Values

An **MI_TYPE_DESC** pointer	is a pointer to the type descriptor for the *type_name* data type.
NULL	indicates that the function was not successful; the *type_name* data type was not found.

Related Topics

See also the descriptions of **mi_type_typedesc( ), mi_typename_to_typedesc( ),** and **mi_typestring_to_id( ).**

mi_udr_lock( )

The **mi_udr_lock( )** function locks an instance of a UDR onto the virtual processor (VP) on which it begins execution.

Syntax

```
mi_integer mi_udr_lock(lock_flag)
   mi_integer lock_flag;
```

lock_flag	is the integer value to set the VP lock flag for the current instance of the UDR. This flag can have either of the following values:	
	MI_TRUE	Sets the VP lock flag to prevent the routine manager from migrating the UDR instance to another VP
	MI_FALSE	Unsets the VP lock flag to tell the routine manager that it can migrate the UDR when necessary

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_udr_lock( )** function sets the VP lock flag to the value that *lock_flag* specifies. The VP lock flag indicates whether to lock the routine instance of a UDR to the current VP. The current VP is the VP on which the current UDR is running. When the VP lock flag is MI_TRUE, the routine manager does not allow the UDR instance to migrate to another VP.

Important: A value of MI_TRUE in the VP lock flag does not prevent another instance of the UDR from executing on another VP.

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the description of **mi_module_lock( ).**

mi_unlock_memory( )

The **mi_unlock_memory( )** function unlocks a named-memory block specified by name and memory duration.

Syntax

```
mi_integer mi_unlock_memory(mem_name, duration)
   mi_string *mem_name;
   MI_MEMORY_DURATION duration;
```

mem_name	is the null-terminated name of the named-memory block to unlock.
duration	is a value that specifies the memory duration of the named-memory block to unlock. Valid values for *duration* follow:

PER_ROUTINE	For the duration of one iteration of the UDR
PER_COMMAND	For the duration of the execution of the current subquery
PER_STATEMENT *(Deprecated)*	For the duration of the current SQL statement
PER_STMT_EXEC	For the duration of the *execution* of the current SQL statement
PER_STMT_PREP	For the duration of the current prepared SQL statement
PER_TRANSACTION	For the duration of one transaction
PER_SESSION	For the duration of the current client session
PER_SYSTEM	For the duration of the database server execution

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Warning: This advanced function can adversely affect your UDR if you use the function incorrectly. Use it only when no regular DataBlade API function can perform the task you need done.

Usage

The **mi_unlock_memory( )** function releases a lock on the named-memory block based on its memory duration of *duration* and its name, which *mem_name* references. 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 the *IBM Informix DataBlade API Programmer's Guide*.)
- *Before* the session ends
- *Before* the memory duration of the named memory expires

- *Before* you attempt to free the named memory with **mi_named_free( )**

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.

Return Values

MI_OK	indicates that the function successfully unlocked the specified named-memory block.
MI_NO_SUCH_NAME	indicates that the requested named-memory block does not exist for the specified duration.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_lock_memory( ), mi_named_alloc( ), mi_named_free( ), mi_named_get( ), mi_named_zalloc( ),** and **mi_try_lock_memory( ).**

mi_unregister_callback( )

The **mi_unregister_callback( )** function unregisters a callback for an event type or for all event types.

Syntax

```
mi_integer mi_unregister_callback(conn, event_type, cback_handle)
   MI_CONNECTION *conn;
   MI_EVENT_TYPE event_type;
   MI_CALLBACK_HANDLE *cback_handle;
```

conn	is either NULL or a pointer to a connection descriptor established by a previous call to **mi_open( ), mi_server_connect( ),** or **mi_server_reconnect( ).**
event_type	is the event type for the callback. For a list of valid event types, see the *IBM Informix DataBlade API Programmer's Guide*.
cback_handle	is the callback handle that a previous call to **mi_register_callback( )** returned.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_unregister_callback( )** function unregisters the callback that *cback_handle* identifies for the event that *event_type* specifies.

Server Only

For a C UDR, *conn* must be NULL for the following event types:

- 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

Client Only

For a client LIBMI application, you must provide a valid connection descriptor to **mi_retrieve_callback( )** for callbacks that handle the following event types:

- MI_Exception
- MI_Xact_State_Change
- MI_Client_Library_Error

End of Client Only

The **mi_register_callback( )** function registers a callback. The database server automatically unregisters a callback when either of the following occurs:

- For a state-transition callback, an end-of-statement (MI_EVENT_END_STMT), end-of-transaction (MI_EVENT_END_XACT), or end-of-session (MI_EVENT_END_SESSION) event occurs and the associated callback completes.
- The associated connection is closed (either the UDR exits or the **mi_close( )** function executes).

Use the **mi_unregister_callback( )** function to explicitly unregister a callback so that the database server does not invoke it when the *event_type* event occurs.

Important: It is recommended that you explicitly unregister a callback function with **mi_unregister_callback( )** once you no longer need it. Otherwise, the registration remains in effect until the associated session closes.

Return Values

MI_OK	indicates that the function was successful; the callback was unregistered.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the description of **mi_register_callback( ).**

For a description of how to register and unregister a callback, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_value( )

The **mi_value( )** function retrieves a column value from a row, given the column identifier.

Syntax

```
mi_integer mi_value(row, column_id, value_buf, value_len)
   MI_ROW *row;
   mi_integer column_id;
   MI_DATUM *value_buf;
   mi_integer *value_len;
```

row	is the row from which values are being extracted.
column_id	indicates the column identifier of the column, which specifies the position of the column in the specified row. Column numbering follows C programming conventions: the first column in the row is at position zero (0).
value_buf	is a pointer to an **MI_DATUM** structure that contains the column value. This function allocates the **MI_DATUM** structure. You do not need to supply the buffer to contain the returned value.
value_len	is the length of the returned column value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_value( )** function returns the value for the column that the *column_id* column identifier specifies. It retrieves this column value from the row structure that *row* references. This function is typically called in a loop that terminates when **mi_value( )** has retrieved values for all the columns in the row.

Important: The "value_buf" parameter points to the representation of the value that corresponds to the control mode of the query: a character string for text representation and the internal format for binary representation.

The way to interpret the data returned in the *value_buf* depends on the **mi_value( )** return value, as follows.

Return Value From mi_value( )	Contents of value_buf
MI_NORMAL_VALUE	A pointer to an **MI_DATUM** structure that holds the value for the column The format of this value depends on whether the control mode for the retrieved data is text or binary representation. In binary mode, the format also depends on whether the **MI_DATUM** value is passed by reference or by value.
MI_ROW_VALUE	A pointer to a row structure (**MI_ROW**)
MI_COLLECTION_VALUE	A pointer to a collection structure (**MI_COLLECTION**)

MI_NULL_VALUE	A pointer to a value that indicates the SQL NULL value for the column

The pointer returned in *value_buf* is valid until a new **mi_exec( )** function is run or until the statement is dropped with **mi_drop_prepared_statement( ).** However, the DataBlade API can overwrite the data when the **mi_next_row( )** function is called on the same connection.

However, the DataBlade API can overwrite the 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.
- The **mi_row_desc_free( )** function is called on the row descriptor.

If you need to save the data for later use, you must create your own copy of the data that *value_buf* references. For example, if the column is a character column, the data in *value_buf* is a pointer to an **mi_lvarchar** structure. To save the data in the **mi_lvarchar** structure, you can copy it with the **mi_var_copy( )** function. You can use a save set to save an entire row.

Return Values

MI_NORMAL_VALUE	indicates that the *retbuf* value is not a row type or a collection.
MI_COLLECTION_VALUE	indicates that the *retbuf* value is a pointer to a collection structure (**MI_COLLECTION**).
MI_ROW_VALUE	indicates that the *retbuf* value is a pointer to a row structure (**MI_ROW**).
MI_NULL_VALUE	indicates that the *retbuf* value is an SQL NULL value.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_collection_fetch( ), mi_next_row( ), mi_save_set_insert( ),** and **mi_value_by_name( ).**

For more information about how to retrieve values, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_value_by_name( )

The **mi_value_by_name( )** function retrieves a column value from a row, given the column name.

Syntax

```
mi_integer mi_value_by_name(row, column_name, retbuf, retlen)
   MI_ROW *row;
   char *column_name;
   MI_DATUM *retbuf;
   mi_integer *retlen;
```

row	is the row from which values are being extracted.
column_name	is name of the column for which the value is to be returned.
retbuf	is a pointer to a location in the user space that will be set to the address of a buffer that contains the returned value. This function allocates the buffer. You do not need to supply the buffer to contain the returned value.
retlen	is the length of the returned column value.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_value_by_name( )** function returns the value for the column that the *column_name* column name specifies. It retrieves this column value from the row structure that *row* references. The only difference between **mi_value_by_name( )** and **mi_value( )** is that the former accesses a column by its name while the latter accesses it by column identifier. Like **mi_value( ),** the **mi_value_by_name( )** function is typically called in a loop that terminates when **mi_value_by_name( )** has retrieved values for all the columns in the row. For more information about how to get a value from a column, see the description of **mi_value( ).**

Return Values

MI_NORMAL_VALUE	indicates that the *retbuf* value is not a row type or a collection. The **mi_value_by_name( )** function places the value in *retbuf*.
MI_COLLECTION_VALUE	indicates that the *retbuf* value is a pointer to a collection structure (**MI_COLLECTION**).
MI_ROW_VALUE	indicates that the *retbuf* value is a pointer to a row structure (**MI_ROW**).
MI_NULL_VALUE	indicates that the *retbuf* value is a NULL value.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_collection_fetch( ), mi_next_row( ), mi_save_set_insert( ),** and **mi_value( ))**.

For more information about how to retrieve values, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_var_copy( )

The **mi_var_copy( )** function creates a copy of a varying-length structure.

Syntax

```
mi_lvarchar *mi_var_copy(varlen_ptr)
   mi_lvarchar *varlen_ptr;
```

varlen_ptr	is a pointer to the varying-length structure to copy.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_var_copy( )** function copies the varying-length structure that *varlen_ptr* references and returns a pointer to the newly allocated copy. The function is a constructor function for a varying-length structure. It creates a varying-length structure with a new data portion whose size is the same as the original varying-length structure (which *varlen_ptr* references).

Server Only

The **mi_var_copy( )** function allocates a new varying-length structure with the current memory duration.

End of Server Only

Use the **mi_var_free( )** function to free this structure when it is no longer needed.

Important: Do not use the DataBlade API memory-management functions such as **mi_alloc( )** to allocate a varying-length structure.

Return Values

An **mi_lvarchar** pointer	is a pointer to a newly allocated varying-length structure.
NULL	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_lvarchar_to_string( ), mi_new_var( ), mi_string_to_lvarchar( ), mi_var_free( ),** and **mi_var_to_buffer( ).**

mi_var_free( )

The **mi_var_free( )** function frees the specified varying-length structure.

Syntax

```
mi_integer mi_var_free (varlen_ptr)
   mi_lvarchar *varlen_ptr
```

varlen_ptr	is the pointer to the varying-length structure to free. ;

Valid in Client LIBMI Application?	**Valid in User-Defined Routine?**
Yes	Yes

Usage

The **mi_var_free( )** function is a destructor function for a varying-length structure. It frees the varying-length structure that *varlen_ptr* references. Use **mi_var_free( )** to explicitly free varying-length structures that you have allocated with the **mi_new_var( )** function or that the **mi_string_to_lvarchar( )** function has allocated.

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

Return Values

MI_OK	indicates that the function was successful.
MI_ERROR	indicates that the function was *not* successful.

Related Topics

See also the descriptions of **mi_lvarchar_to_string( ), mi_new_var( ), mi_string_to_lvarchar( ), mi_var_copy( ),** and **mi_var_to_buffer( ).**

mi_var_to_buffer( )

The **mi_var_to_buffer( )** function copies data from a varying-length structure to a buffer.

Syntax

```
void mi_var_to_buffer(varlen_ptr, buffer)
   mi_lvarchar *varlen_ptr;
   char *buffer;
```

varlen_ptr	is a pointer to the varying-length structure.
buffer	is a pointer to the user-allocated buffer.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_var_to_buffer( )** function copies data from the varying-length structure that *varlen_ptr* references to a user-allocated buffer that *buffer* references. 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. You must ensure that the buffer that *buffer* references is large enough to hold the data.

Return Values

None.

Related Topics

See also the descriptions of **mi_lvarchar_to_string( ), mi_new_var( ), mi_string_to_lvarchar( ), mi_var_copy( ),** and **mi_var_to_buffer( ).**

mi_version_comparison( )

The **mi_version_comparison( )** function compares the version of two instances of Dynamic Server.

Syntax

```
mi_integer mi_version_comparison(s1, s2)
   mi_char1 *s1;
   mi_char *s2;
```

s1	is the starting string address of the version number to compare.
s2	is the ending string address of the version number to compare.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_version_comparison( )** function compares the version for the major, minor, and interim version numbers of two database servers. For example, for IBM Informix Dynamic Server 9.40.UC6, the major version is 9, the minor version is 40, and the interim number is 6. If the interim number is omitted in one of the strings, then both strings will only be compared for only the major and minor release numbers.

Important: You should always use the **mi_version_comparison( )** function instead of using a string comparison.

Return Values

MI_ERROR (-1)	indicates an error in the strings that are passed.
2	indicates that the value of *s1* is greater than the value of *s2*.
0	indicates that the values of *s1* and *s2* are equal.
-2	indicates that the value of *s1* is less than the value of *s2*.

Related Topics

See also the description of **mi_server_library_version( )**.

mi_vpinfo_classid( )

The **mi_vpinfo_classid( )** function obtains the VP-class identifier for the VP class in which the current UDR is running.

Syntax

```
mi_integer mi_vpinfo_classid(void)
```

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Warning: This advanced function can adversely affect your UDR if you use the function incorrectly. Use it only when no regular DataBlade API function can perform the task you need done.

Usage

The **mi_vpinfo_classid( )** function returns the VP-class identifier for the current VP. The current VP is the VP on which the current UDR is running. The VP-class identifier is a unique integer that the database server assigns to each defined VP class.

Tip: You can obtain the VP-class identifier for a specified VP with the **mi_class_id( )** function.

After you have a VP-class identifier, you can use the following DataBlade API functions to obtain additional information about the VP class.

DataBlade API Function	VP-Class Information
mi_class_name( )	VP-class name
mi_class_maxvps( )	Maximum number of VPs in the VP class
mi_class_numvp( )	Number of active VPs in the VP class

Return Values

>=0	is the VP-class integer for the VP class in which the current UDR is running.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_class_id( ), mi_class_maxvps( ), mi_class_name( ), mi_class_numvp( ), mi_vpinfo_isnoyield( ),** and **mi_vpinfo_vpid( ).**

For information about how to obtain information on VPs and VP classes, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_vpinfo_isnoyield( )

The **mi_vpinfo_isnoyield( )** function determines whether the virtual processor (VP) on which the current UDR is executing is part of a nonyielding user-defined VP class.

Syntax

```
mi_integer mi_vpinfo_isnoyield(void)
```

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Warning: This advanced function can adversely affect your UDR if you use the function incorrectly. Use it only when no regular DataBlade API function can perform the task you need done.

Usage

The **mi_vpinfo_isnoyield( )** function determines whether the VP class associated with the current VP has been defined as a nonyielding user-defined VP class. The current VP is the VP on which the current UDR is running. A nonyielding user-defined VP class executes the UDR to completion, without allowing execution to yield control of the VP.

Return Values

0 (MI_FALSE)	indicates that the current VP is not part of a nonyielding VP class. The current VP class is a yielding VP class.
1 (MI_TRUE)	indicates that the current VP is part of a nonyielding VP class.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_vpinfo_classid( )** and **mi_vpinfo_vpid( ).**

For information about how to obtain information on the VP environment or about when to use a nonyielding user-defined VP, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_vpinfo_vpid( )

The **mi_vpinfo_vpid( )** function obtains the VP identifier of the virtual processor (VP) on which the user-defined routine (UDR) is executing.

Syntax

```
mi_integer mi_vpinfo_vpid(void)
```

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Warning: This advanced function can adversely affect your UDR if you use the function incorrectly. Use it only when no regular DataBlade API function can perform the task you need done.

Usage

The **mi_vpinfo_vpid( )** function returns the VP identifier for the current VP. The current VP is the VP on which the current UDR is running. The VP identifier is a unique integer that the database server assigns to each active VP. The output of the **onstat -g glo** command displays the VP identifier for a VP in its first output column.

After you have a VP identifier, you can use the following DataBlade API functions to obtain additional information about the VP environment.

DataBlade API Function	VP-Environment Information
mi_vpinfo_classid( )	VP-class identifier
mi_vpinfo_isnoyield( )	Does the current VP belong to a nonyielding VP class?

Return Values

>=0	is the VP identifier for the VP on which the current UDR is running.
MI_ERROR	indicates that the function was not successful.

Related Topics

See also the descriptions of **mi_vpinfo_classid( )** and **mi_vpinfo_isnoyield( ).**

For information about how to obtain information on the VP environment, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_xa_get_current_xid( )

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.

Syntax

```
XID * mi_xa_get_current_xid ( )
```

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_xa_get_current_xid( )** function must be invoked from within the applicable transaction.

If you receive an error, check for any of the following problems:

1. Make sure that the **mi_xa_get_current_xid( )** function is called from within the transaction.
2. Make sure that the **mi_xa_get_current_xid( )** function is not called
   - From the sub-ordinator of a distributed transaction.
   - In a non-logging database

Return Values

NULL	indicates that an error occurred while getting the resource manager ID.

Return values that are other than NULL are pointers to the current XID structure.

Related Topics

See the descriptions of **mi_xa_register_xadatasource( )** and **mi_xa_unregister_xadatasource( )**.

For more information on working with XA data sources, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_xa_get_xadatasource_rmid( )

The **mi_xa_get_xadatasource_rmid( )** function gets the resource manager ID that was previously created in the database for an XA-compliant, external data source.

Syntax

```
mi_integer mi_xa_get_xadatasource_rmid(mi_string *xasrc)
```

xasrc	is the user-defined name of the XA data source.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_xa_get_xadatasource_rmid( )** function can be used while invoking the **ax_reg( )** or the **ax_unreg( )** function in subsequent calls. If successful, this function returns the resource manger ID.

The format of the *xarsc* name is *[owner].xadatasourcename*. If the owner is specified, the **mi_xa_get_xadatasource_rmid( )** function searches for the owner and the data source name. If the owner is not specified, the function:

- Searches for the XA data source name in a non-ANSI database.
- Adds the current user to the XA data source name when searching an ANSI database.

If you receive an error, check for any of the following problems:

1. Make sure the value for *xasrc* is correct.
2. Make sure that the **mi_xa_get_xadatasource_rmid( )** function is not called:
   - From the sub-ordinator of a distributed transaction.
   - In a non-logging database
3. Make sure that the *xasrc* XA data source is created in the database.

Return Values

> 0	is the resource manager ID.
MI_ERROR	indicates that an error occurred while getting the resource manager ID.
MI_NOSUCH_XASOURCE	indicates that the XA data source does not exist in the system.
MI_INVALID_XANAME	indicates that the user-defined name (*owner.xadatasourcename*) for an instance of XA data source is not valid.

Related Topics

See the descriptions of **ax_reg( )** and **ax_unreg( )**.

For more information on working with XA data sources, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_xa_register_xadatasource( )

The **mi_xa_register_xadatasource( )** function registers an XA data source with the current transaction. The registration is dynamic and is applicable for the current transaction only. The DataBlade should register participating data sources into each transaction.

Syntax

```
mi_integer mi_xa_register_xadatasource(mi_string *xasrc)
```

xasrc	is the user-defined name of the XA data source.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The format of the *xarsc* name is *[owner].xadatasourcename*. If the owner is specified, the **mi_xa_get_xadatasource_rmid( )** function searches for the owner and the data source name. If the owner is not specified, the function:

- Searches for the XA data source name in a non-ANSI database.
- Adds the current user to the XA data source name when searching an ANSI database.

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 **mi_xa_unregister_xadatasource( )** unregisters the data source from the transaction.

You must be sure that the XA data source was created using this SQL statement:

CREATE XADATASOURCE *xadatassourcename* USING *xads type*

For more information on this statement, see the *IBM Informix Guide to SQL: Syntax*.

If you receive an error, check for any of the following problems:

1. Make sure the value for *xasrc* is correct.
2. Make sure that the **mi_xa_register_xadatasource( )** function is called from within the transaction.
3. Make sure that the **mi_xa_register_xadatasource( )** function is not called:
   - 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, which creates a type of XA-compliant external data source.

The **ax_reg( )** function also allows DataBlade modules to register XA-compliant, external data sources. However, the **ax_reg( )** function and the **mi_xa_register_xadatasource( )** function use different parameters and have different return values.

Return Values

MI_OK	indicates that the XA data source is registered.
MI_ERROR	indicates that an error occurred and the XA data source is not registered.
MI_NOSUCH_XASOURCE	indicates that the XA data source does not exist in the system.
MI_INVALID_XANAME	indicates that the user-defined name for an instance of the XA data source is not valid.
MI_NOTINTX	indicates that the function was called from outside the transaction. The function must be called from within the transaction.
MI_XAOPEN_ERROR	indicates that the **xa_open** purpose function of the XA data source returned an error.

Related Topics

See the descriptions of **mi_xa_get_current_xid( )**, **mi_xa_unregister_xadatasource( )**, **ax_reg( )**, and **ax_unreg( )**.

For more information on working with XA data sources, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_xa_unregister_xadatasource( )

The **mi_xa_unregister_xadatasource( )** function unregisters the previously registered XA data source from the transaction.

Syntax

```
mi_integer mi_xa_unregister_xadatasource(mi_string *xasrc)
```

xasrc	is the user-defined name of the XA data source. The format of the `xarsc` name is *owner.xadatasourcename*

Usage

Because the **mi_xa_unregister_xadatasource( )** function unregisters the data source from the transaction, none of the transactional events that the data source would have instigated occur.

The format of the *xarsc* name is *[owner].xadatasourcename*. If the owner is specified, the **mi_xa_get_xadatasource_rmid( )** function searches for the owner and the data source name. If the owner is not specified, the function:

- Searches for the XA data source name in a non-ANSI database.
- Adds the current user to the XA data source name when searching an ANSI database.

Multiple registrations of the same XA data source in a transaction have the same effect as a single registration. Since Dynamic Server does not maintain a count of the number of times an application has registered, a single call to **mi_xa_unregister_xadatasource( )** unregisters the data source from the transaction.

If you receive an error, check for any of the following problems:

1. Make sure the value for *xasrc* is correct.
2. Make sure that the **mi_xa_unregister_xadatasource( )** function is called from within the transaction.
3. Make sure that the **mi_xa_unregister_xadatasource( )** function is not called:
   - 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, which creates a type of XA-compliant external data source.
4. Make sure that you are not unregistering an XA data source that is not registered.

The **ax_unreg( )** function also allows DataBlade modules to register XA-compliant, external data sources. However, the **ax_unreg( )** function and the **mi_xa_unregister_xadatasource( )** function use different parameters and have different return values.

Return Values

MI_OK	indicates that the data source is unregistered.
MI_ERROR	indicates that an error occurred and the data source was not unregistered.

MI_NOSUCH_XASOURCE
: indicates that the XA data source does not exist in the system.

MI_INVALID_XANAME
: indicates that the user-defined name for an instance of the XA data source is not valid.

MI_NOTINTX
: indicates that the function was called from outside the transaction. The function must be called from within the transaction.

Related Topics

See the descriptions of **mi_xa_register_xadatasource( )**, **mi_xa_get_current_xid( )**, **ax_reg( )**, and **ax_unreg( )**.

For more information on working with XA data sources, see the *IBM Informix DataBlade API Programmer's Guide*.

mi_yield( )

The **mi_yield( )** routine yields processing to other database server threads.

Syntax

```
void mi_yield( )
```

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
No	Yes

Usage

The **mi_yield( )** function causes the thread that is executing the C UDR to yield control of the virtual processor that is executing it. When the virtual processor is released, it can execute other threads.

Use the **mi_yield( )** call periodically when the routine does CPU or I/O intensive work. The **mi_yield( )** routine is useful in portions of code, such as tight loops, that would otherwise tie up the processor.

Return Values

None.

mi_zalloc( )

The **mi_zalloc( )** function allocates and zero-fills a block of memory of the given size and returns a pointer to a block of user memory of the specified size.

Syntax

```
void *mi_zalloc (size)
   mi_integer size;
```

size is the number of bytes to allocate and fill with zeroes.

Valid in Client LIBMI Application?	Valid in User-Defined Routine?
Yes	Yes

Usage

The **mi_zalloc( )** function allocates *size* number of bytes of user memory for a DataBlade API module. The **mi_zalloc( )** function behaves exactly like **mi_alloc( )** except that **mi_zalloc( )** fills the allocated block of memory with zeroes before the function returns the pointer to the memory. The **mi_zalloc( )** function is a constructor function for user memory.

Server Only

The **mi_zalloc( )** function allocates the memory in the current memory duration. By default, the current default duration is PER_ROUTINE. In C UDRs, the database server also automatically frees memory allocated with **mi_zalloc( )** when an exception is raised.

Important: Use DataBlade API memory-management functions, such as **mi_zalloc( )**, to allocate memory in C UDRs. Use of a DataBlade API memory-management function guarantees that the database server automatically frees the memory, especially in the cases of return values or exceptions, where the UDR would not otherwise be able to free the memory.

End of Server Only

Client Only

In client LIBMI applications, **mi_zalloc( )** works exactly as **malloc( )** does: it allocates storage on the heap of the client process. However, the database server does *not* perform any automatic garbage collection. Therefore, the client LIBMI application *must* use **mi_free( )** to explicitly free all allocations that **mi_zalloc( )** makes.

Important: Client LIBMI applications ignore memory duration.

Client LIBMI applications can use either DataBlade API memory-management functions or system memory-management functions (such as **malloc( )**).

End of Client Only

The **mi_zalloc( )** function returns a pointer to the newly allocated memory. Cast this pointer to match the structure of the user-defined buffer or structure that you allocate. A DataBlade API module can use **mi_free( )** to free memory allocated by **mi_zalloc( )** when that memory is no longer needed.

Return Values

A **void** pointer	is a pointer to the newly allocated memory. Cast this pointer to match the user-defined buffer or structure for which the memory was allocated.
NULL	indicates that the function was unable to allocate the memory.

The **mi_zalloc( )** function does *not* throw an MI_Exception event when it encounters a runtime error. Therefore, it does not cause callbacks to be invoked.

Related Topics

See also the descriptions of **mi_alloc( ), mi_dalloc( ), mi_free( ),** and **mi_switch_mem_duration( ).**

For more information about memory allocation and memory durations, see the *IBM Informix DataBlade API Programmer's Guide*.

rdatestr( )

The **rdatestr( )** function converts an internal DATE value to a character string.

Syntax

```
mint rdatestr(jdate, outbuf)
   int4 jdate;
   char *outbuf;
```

jdate	is the internal representation of the date to format.
outbuf	is a pointer to the buffer to contain the string for the *jdate* value.

Usage

For the default locale, U.S. English, the **rdatestr( )** function determines how to interpret the format of the character string with the following precedence:

1. The format that the **DBDATE** environment variable specifies (if **DBDATE** is set)

   For more information, see the *IBM Informix Guide to SQL: Reference*.

Global Language Support

2. The format that the **GL_DATE** environment variable specifies (if **GL_DATE** is set)

   For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

3. The default date form: *mm/dd/yyyy*

Global Language Support

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*.

End of Global Language Support

Return Values

0	The conversion was successful.
<0	The conversion failed.
-1210	The internal date could not be converted to the character string format.
1212	Data conversion format must contain a month, day, or year component. **DBDATE** specifies the data conversion format.

rdayofweek( )

The **rdayofweek( )** function returns the day of the week as an integer value for an internal DATE.

Syntax

```
mint rdayofweek(jdate)
   int4 jdate;
```

jdate	is the internal representation of the date.

Return Values

0	Sunday
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday

rdefmtdate( )

The **rdefmtdate( )** function uses a formatting mask to convert a character string to an internal DATE format.

Syntax

```
mint rdefmtdate(jdate, fmtstring, inbuf)
   int4 *jdate;
   char *fmtstring;
   char *inbuf;
```

jdate	is a pointer to an **int4** variable that receives the internal DATE value for the *inbuf* string.
fmtstring	is a pointer to a buffer that contains the formatting mask to use for the *inbuf* string.
inbuf	is a pointer to the buffer that contains the date string to convert.

Usage

The *fmtstring* argument of the **rdefmtdate( )** function points to the date-formatting mask, which contains formats that describe how to interpret the date string.

The *input* string and the *fmtstring* must be in the same sequential order in terms of month, day, and year. They need not, however, contain the same literals or the same representation for month, day, and year.

You can include the weekday format (`ww`), in *fmtstring*, but the database server ignores that format. Nothing from the *inbuf* corresponds to the weekday format.

The following combinations of *fmtstring* and *input* are valid.

Formatting Mask	**Input**
mmddyy	Dec. 25th, 2006
mmddyyyy	Dec. 25th, 2006
mmm. dd. yyyy	dec 25 2006
mmm. dd. yyyy	DEC-25-2006
mmm. dd. yyyy	122506
mmm. dd. yyyy	12/25/06
yy/mm/dd	06/12/25
yy/mm/dd	2006, December 25th
yy/mm/dd	In the year 2006, the month of December, it is the 25th day
dd-mm-yy	This 25th day of December, 2006

If the value stored in *inbuf* is a four-digit year, the **rdefmtdate( )** function uses that value. If the value stored in *inbuf* is a two-digit year, the **rdefmtdate( )** function uses the value of the **DBCENTURY** environment variable to determine which century to use. If you do not set **DBCENTURY**, the DataBlade API uses the current century. For information on how to set **DBCENTURY**, see the *IBM Informix Guide* *to SQL: Reference*.

Global Language Support

When you use a nondefault locale whose dates contain eras, you can use extended-format strings in the *fmtstring* argument of **rdefmtdate( )**. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Return Values

If you use an invalid date-string format, **rdefmtdate( )** returns an error code and sets the internal DATE to the current date. Possible return values follow.

0	The operation was successful.
-1204	The *input* parameter specifies an invalid year.
-1205	The *input* parameter specifies an invalid month.
-1206	The *input* parameter specifies an invalid day.
-1209	Because *input* does not contain delimiters between the year, month, and day, the length of *input* must be exactly six or eight bytes.
-1212	The *fmtstring* parameter does not specify a year, a month, and a day.

rdownshift( )

The **rdownshift( )** function changes all the uppercase characters within a null-terminated string to lowercase characters.

Syntax

```
void rdownshift(s)
   char *s;
```

s	is a pointer to a null-terminated string.

Usage

The **rdownshift( )** function refers to the current locale to determine uppercase and lowercase letters. For the default locale, U.S. English, **rdownshift( )** uses the ASCII lowercase (a to z) and uppercase (A to Z) letters.

Global Language Support

If you use a nondefault locale, **rdownshift( )** uses the lowercase and uppercase letters that the locale defines. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

rfmtdate( )

The **rfmtdate( )** function uses a formatting mask to convert an internal DATE value to a character string.

Syntax

```
mint rfmtdate(jdate, fmtstring, outbuf)
   int4 jdate;
   char *fmtstring;
   char *outbuf;
```

jdate	is the internal representation of the date to convert.
fmtstring	is a pointer to a buffer that contains the formatting mask to use for the *jdate* value.
outbuf	is a pointer to a buffer to contain the formatted string for the *jdate* value.

Usage

The *fmtstring* argument of the **rfmtdate( )** function points to the date-formatting mask, which contains formats that describe how to format the date string.

The examples in the following table use the formatting mask in *fmtstring* to convert the integer *jdate*, whose value corresponds to December 25, 2006, to a formatted string *outbuf*. You must specify one or more fields.

Formatting Mask	**Formatted Result**
″mmdd″	1225
″mmddyy″	122506
″ddmmyy″	251206
″yydd″	0625
″yymmdd″	061225
″dd″	25
″yy/mm/dd″	06/12/25
″yy mm dd″	06 12 25
″yy-mm-dd″	06-12-25
″mmm. dd, yyyy″	Dec. 25, 2006
″mmm dd yyyy″	Dec 25 2006
″yyyy dd mm″	2006 25 12
″mmm dd yyyy″	Dec 25 2006
″ddd, mmm. dd, yyyy″	Mon, Dec. 25, 2006
″ww mmm. dd, yyyy″	Mon Dec. 25, 2006
″(ddd) mmm. dd, yyyy″	(Mon) Dec. 25, 2006
″mmyyddmm″	25061225
″″	unpredictable result

Global Language Support

When you use a nondefault locale whose dates contain eras, you can use extended-format strings in the *fmtstring* argument of **rfmtdate( )**. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Return Values

0	The conversion was successful.
-1210	The internal date cannot be converted to month-day-year format.
-1211	The program ran out of memory (memory-allocation error).
-1212	Format string is NULL or invalid.

rfmtdec( )

The **rfmtdec( )** function uses a formatting mask to convert a **decimal** value to a character string.

Syntax

```
int rfmtdec(dec_val, fmtstring, outbuf)
   dec_t *dec_val;
   char *fmtstring;
   char *outbuf;
```

dec_val	is a pointer to the **decimal** value to format.
fmtstring	is a pointer to a character buffer that contains the formatting mask to use for the *dec_val* value.
outbuf	is a pointer to a character buffer to contain the formatted string for the *dec_val* value.

Usage

The *fmtstring* argument of the **rfmtdec( )** function points to the numeric-formatting mask, which contains characters that describe how to format the **decimal** value.

Global Language Support

When you use **rfmtdec( )** to format MONEY values, the function uses the currency symbols that the **DBMONEY** environment variable specifies. If you do not set this environment variable, **rfmtdec( )** uses 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*).

When you use a nondefault locale that has a multibyte code set, **rfmtdec( )** supports multibyte characters in the format string. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Return Values

0	The conversion was successful.
-1211	The program ran out of memory (memory-allocation error).
-1217	The format string is too large.

rfmtdouble( )

The **rfmtdouble( )** function uses a formatting mask to convert a C **double** value to a character string.

Syntax

```
mint rfmtdouble(dbl_val, fmtstring, outbuf)
   double dbl_val;
   char *fmtstring;
   char *outbuf;
```

dbl_val	is the **double** number to format.
fmtstring	is a pointer to a character buffer that contains the formatting mask for the value in *dbl_val*.
outbuf	is a pointer to a character buffer to contain the formatted string for the value in *dbl_val*.

Usage

The *fmtstring* argument of the **rfmtdouble( )** function points to the numeric-formatting mask, which contains characters that describe how to format the **double** value.

Global Language Support

When you use **rfmtdouble( )** to format MONEY values, the function uses the currency symbols that the **DBMONEY** environment variable specifies. If you do not set this environment variable, **rfmtdouble( )** uses 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*.)

When you use a nondefault locale that has a multibyte code set, **rfmtdouble( )** supports multibyte characters in the format string. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Return Values

0	The conversion was successful.
-1211	The program ran out of memory (memory-allocation error).
-1217	The format string is too large.

rfmtlong( )

The **rfmtlong( )** function uses a formatting mask to convert a C **long** value to a character string.

Syntax

```
mint rfmtlong(lng_val, fmtstring, outbuf)
   int4 lng_val;
   char *fmtstring;
   char *outbuf;
```

lng_val	is the **int4** integer to convert to a character value.
fmtstring	is a pointer to a character buffer that contains the formatting mask for the value in *lng_val*.
outbuf	is a pointer to a character buffer to contain the formatted string for the value in *lng_val*.

Usage

The *fmtstring* argument of the **rfmtlong( )** function points to the numeric-formatting mask, which contains characters that describe how to format the **long integer** value.

Global Language Support

When you use **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, **rfmtlong( )** uses 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*.)

When you use a nondefault locale that has a multibyte code set, **rfmtlong( )** supports multibyte characters in the format string. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

Return Values

0	The conversion was successful.
-1211	The program ran out of memory (memory-allocation error).
-1217	The format string is too large.

rjulmdy( )

The **rjulmdy( )** function creates an array of three **short** integer values that represent the month, day, and year from an internal DATE value.

Syntax

```
mint rjulmdy(jdate, mdy)
   int4 jdate;
   int2 mdy[3];
```

jdate	is the internal representation of the date.
mdy	is an array of **short** integers, in which *mdy[0]* is the month (1 to 12), *mdy[1]* is the day (1 to 31), and *mdy[2]* is the year (1 to 9999).

Return Values

0	The operation was successful.
< 0	The operation failed.
-1210	The internal date could not be converted to the character string format.

rleapyear( )

The **rleapyear( )** function returns 1 (TRUE) when the argument that is passed to it is a leap year and 0 (FALSE) when it is not.

Syntax

```
mint rleapyear(year)
   mint year;
```

year	is an integer.

Usage

The argument *year* must be the year component of a date and not the date itself. You must express the *year* in full form (2006) and not abbreviated form (06).

Return Values

1	The year is a leap year.
0	The year is not a leap year.

rmdyjul( )

The **rmdyjul( )** function creates an internal DATE from an array of three **short** integer values that represent month, day, and year.

Syntax

```
mint rmdyjul(mdy, jdate)
   int2 mdy[3];
   int4 *jdate;
```

mdy	is an array of **short** integer values, in which *mdy[0]* is the month (1 to 12), *mdy[1]* is the day (1 to 31), and *mdy[2]* is the year (1 to 9999).
jdate	is a pointer to a **long** integer to contain the internal DATE value for the *mdy* array.

Usage

You can express the year in full form (2006) or abbreviated form (06).

Return Values

0	The operation was successful.
-1204	The *mdy[2]* variable contains an invalid year.
-1205	The *mdy[0]* variable contains an invalid month.
-1206	The *mdy[1]* variable contains an invalid day.

rstod( )

The **rstod( )** function converts a null-terminated string into a C double value.

Syntax

```
mint rstod(string, double_val)
   char *string;
   double *double_val;
```

string	is a pointer to the null-terminated string to convert.
double_val	is a pointer to a double variable to contain the converted value.

Return Values

=0	The conversion was successful.
!=0	The conversion failed.

rstoi( )

The **rstoi( )** function converts a null-terminated string into a short integer value.

Syntax

```
mint rstoi(string, ival)
   char *string;
   mint *ival;
```

string	is a pointer to the null-terminated string to convert.
ival	is a pointer to an mint variable to contain the converted value.

Usage

The legal range of values is from `-32767` to `32767`. The value `-32768` is *not* valid because this value is a reserved value that indicates null.

If *string* corresponds to a null integer, *ival* points to the representation for a SMALLINT null. To convert a string that corresponds to a long integer, use **rstol( )**. Failure to do so can result in corrupt data representation.

Return Values

=0	The conversion was successful.
!=0	The conversion failed.

rstol( )

The **rstol( )** function converts a null-terminated string into a long integer value.

Syntax

```
mint rstol(string, long_int)
   char *string;
   mlong *long_int;
```

string	is a pointer to the null-terminated string to convert.
long_int	is a pointer to an mlong variable to contain the converted value.

Usage

The legal range of values is from `-2,147,483,647` to `2,147,483,647`. The value `-2,147,483,648` is *not* valid because this value is a reserved value that indicates null.

Return Values

=0	The conversion was successful.
!=0	The conversion failed.

rstrdate( )

The **rstrdate( )** function converts a character string to an internal DATE.

Syntax

```
mint rstrdate(inbuf, jdate)
   char *inbuf;
   int4 *jdate;
```

inbuf	is a pointer to the string that contains the date to convert.
jdate	is a pointer to an **int4** variable to contain the internal DATE value for the *inbuf* string.

Usage

For the default locale, U.S. English, the **rstrdate( )** function determines how to format the character string with the following precedence:

1. The format that the **DBDATE** environment variable specifies (if **DBDATE** is set)

   For more information, see the *IBM Informix Guide to SQL: Reference*.

Global Language Support

2. The format that the **GL_DATE** environment variable specifies (if **GL_DATE** is set)

   For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

3. The default date form: *mm/dd/yyyy*

   You can use any nonnumeric character as a separator between the month, day, and year. You can express the year as four digits (2007) or as two digits (95).

Global Language Support

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

End of Global Language Support

When you use a two-digit year in the date string, the **rstrdate( )** function uses the value of the **DBCENTURY** environment variable to determine which century to use. If you do not set **DBCENTURY**, **rstrdate( )** assumes the current century for two-digit years. For information on how to set **DBCENTURY**, see the *IBM Informix Guide to SQL: Reference*.

Return Values

0	The conversion was successful.
< 0	The conversion failed.
-1204	The *inbuf* parameter specifies an invalid year.
-1205	The *inbuf* parameter specifies an invalid month.
-1206	The *inbuf* parameter specifies an invalid day.

-1212	Data conversion format must contain a month, day, or year component. **DBDATE** specifies the data conversion format.
-1218	The date specified by the *inbuf* argument does not properly represent a date.

rtoday( )

The **rtoday( )** function returns the system date as a long integer value.

Syntax

```
void rtoday(today)
   int4 *today;
```

today	is a pointer to an **int4** variable to contain the internal DATE.

Usage

The **rtoday( )** function obtains the system date on the client computer, not the server computer.

rupshift( )

The **rupshift( )** function changes all the characters within a null-terminated string to uppercase characters.

Syntax

```
void rupshift(s)
      char *s;
```

s	is a pointer to a null-terminated string.

Usage

The **rupshift( )** function refers to the current locale to determine uppercase and lowercase letters. For the default locale, U.S. English, **rupshift( )** uses the ASCII lowercase (a-z) and uppercase (A-Z) letters.

Global Language Support

If you use a nondefault locale, **rupshift( )** uses the lowercase and uppercase letters that the locale defines. For more information, see the *IBM Informix GLS User's Guide*.

End of Global Language Support

stcat( )

The **stcat( )** function concatenates one null-terminated string to the end of another.

Syntax

```
void stcat(s, dest)
   char *s, *dest;
```

s	is a pointer to the start of the string to place at the end of the destination string.
dest	is a pointer to the start of the null-terminated destination string.

stchar( )

The **stchar( )** function stores a null-terminated string in a fixed-length string, padding the end with blanks, if necessary.

Syntax

```
void stchar(from, to, count)
   char *from;
   char *to;
   mint count;
```

from	is a pointer to the first byte of a null-terminated source string.
to	is a pointer to the fixed-length destination string. This argument can point to a location that overlaps the location to which the *from* argument points. In this case, the function discards the value to which *from* points.
count	is the number of bytes in the fixed-length destination string.

stcmpr( )

The **stcmpr( )** function compares two null-terminated strings.

Syntax

```
mint stcmpr(s1, s2)
   char *s1, *s2;
```

s1	is a pointer to the first null-terminated string.
s2	is a pointer to the second null-terminated string.

Important: When “s1” appears after “s2” in the ASCII collation sequence, “s1” is greater than “s2”.

Return Values

=0	The two strings are identical.
<0	The first string is less than the second string.
>0	The first string is greater than the second string.

stcopy( )

The **stcopy( )** function copies a null-terminated string from one location in memory to another location.

Syntax

```
void stcopy(from, to)
   char *from, *to;
```

from	is a pointer to the null-terminated string to copy.
to	is a pointer to a location in memory to which to copy the string.

stleng( )

The **stleng( )** function returns the length, in bytes, of a null-terminated string that you specify.

Syntax

```
mint stleng(string)
   char *string;
```

string	is a pointer to a null-terminated string.

Usage

The length does not include the null terminator.

Appendix. 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.

© Copyright IBM Corp. 1996, 2008

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 ™), 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

© Copyright IBM Corp. 1996, 2008

Printed in USA

SC23-9428-01

Spine information:

IBM Informix **Version 11.50** **IBM Informix DataBlade API Function Reference**