InterBase 5

API Guide

InterBase®

SOFTWARE CORPORATION

100 Enterprise Way, Suite B2 Scotts Valley, CA 95066 http://www.interbase.com

1INT0055WW21001 5E4R0898

9899000102-9 8 7 6 5 4 3 2 1

D4

Table of Contents

List of Tables

CHAPTER

1

Using the API Guide

The InterBase API Guide is a task-oriented explanation of how to write, preprocess, compile, and link database applications using the InterBase Applications Programming Interface (API), and a *host programming language*, either C or C++.

This chapter describes the focus of this book, and provides a brief overview of its chapters.

Who should use this guide

The InterBase *API Guide* is intended for knowledgeable database applications programmers. It assumes full knowledge of:

- SQL and dynamic SQL (DSQL).
- Relational database programming.
- C programming.

Topics covered in this guide

The *API Guide* is divided into two parts:

- A task-oriented user's guide that explains how to use API function calls to perform related database tasks, such as attaching to and detaching from a database.
- An API function call reference that describes the purpose of each function, its syntax, its parameters, requirements, restrictions, and return values, as well as examples of use and cross-references to related functions.

The following table provides a brief description of each chapter in the *API Guide*:

Chapter	Description
Chapter 2, "Application Requirements"	Describes support structures and elements common to programming with API calls
Chapter 3, "Programming with the InterBase API"	Describes special requirements for programming InterBase applications with the InterBase API
Chapter 4, "Working with Databases"	Describes how to attach to and detach from databases, and how to request information about attachments
Chapter 5, "Working with Transactions"	Explains how to start transactions in different modes, and how to commit them or roll them back
Chapter 6, "Working with Dynamic SQL"	Describes how to process DSQL data definition and data manipulation statements using API calls
Chapter 7, "Working with Blob Data"	Describes how to select, insert, update, and delete Blob data in applications
Chapter 8, "Working with Array Data"	Describes how to select, insert, update, and delete array data in applications
Chapter 9, "Working with Conversions"	Describes how to select, insert, update, and delete DATE data in applications, and how to reverse the byte order of numbers with **isc_vax_integer()**
Chapter 10, "Handling Error Conditions"	Describes how to trap and handle database errors in applications
Chapter 11, "Working with Events"	Explains how triggers interact with applications and describes how to register interest in events, wait on them, and respond to them in applications

TABLE 1.1 *API Guide* chapters

Chapter	Description
Chapter 12, "API Function Reference"	Describes the syntax of each function call in detail.
Appendix A, "InterBase Document Conventions"	Lists typefaces and special characters used in this book to describe syntax and identify object types.
Appendix B, "Data Structures"	Lists and describes the data structures, constants, and buffers that are defined in *ibase.h*.

TABLE 1.1 *API Guide* chapters (*continued*)

Sample database and applications

The InterBase *Examples* subdirectory contains a sample database and sample application source code. The examples in this *API Guide* make use of this sample database and source code wherever possible.

PART I

API User’s Guide

CHAPTER

2

Application Requirements

This chapter summarizes programming requirements for using categories of API functions in database applications, and provides cross-references to more detailed information in later chapters.

All API applications must use certain API functions and support structures. For example, all applications connect to at least one database, and run at least one transaction. All applications, therefore, must declare and initialize database handles and transaction handles. They may also need to declare and populate database parameter buffers (DPBs) and transaction parameter buffers (TPBs). This chapter outlines those requirements, and points you to more detailed information later in this book.

Some API applications may use specific API functions, such as the functions that permit an application to process dynamic SQL (DSQL) statements. These applications have additional requirements that are also outlined in this chapter along with pointers to more detailed information elsewhere in this book.

Requirements for all applications

The following sections outline these requirements for all API applications:

- Including *ibase.h*
- Database requirements

- Transaction requirements

Including *ibase.h*

The InterBase subdirectory, *include*, contains the *ibase.h* header file, which should be included in all source code modules for API applications. *ibase.h* contains API function prototypes. It also contains structure typedefs, parameter definitions, and macros required by various API functions.

To include *ibase.h* in a source code module, insert the following #include near the start of the source code:

```
#include <ibase.h>
```

If *ibase.h* is not on your compiler's search path, you may need to provide a full path specification and enclose the file name in quotation marks.

Failure to include *ibase.h* can prevent the successful compilation and linking of an application.

Database requirements

All applications that work with databases must provide one database handle for each database to be accessed. A *database handle* is a long pointer that is used in API functions to attach to a database and to reference it in subsequent API calls. The InterBase header file, *ibase.h*, contains a #define useful for declaring database handles.

When establishing a connection to a database, optional database attachment characteristics, such as a user name and password combination, can be passed to the attachment through a database parameter buffer (DPB). Usually, one DPB is set up for each database attachment, although database attachments can also share a DPB.

Declaring database handles

A database handle must be declared and initialized to zero before use. The following code illustrates how to declare and initialize a database handle:

```
#include <ibase.h>
. . .
/* Declare a database handle. */
isc_db_handle db1;
. . .
/* Initialize the handle. */
```

```
db1 = 0L;
```

For more information about declaring, initializing, and using database handles, see **Chapter 4, "Working with Databases."**

Setting up a DPB

A DPB is a byte array describing optional database attachment characteristics. A DPB must be set up and populated before attaching to a database. Parameters that can be passed to the DPB are defined in *ibase.h*.

For more information about setting up, populating, and using a DPB, see **Chapter 4, "Working with Databases."**

Transaction requirements

All applications must provide one transaction handle for each transaction to be accessed. A *transaction handle* is a long pointer that is used in API functions to start a transaction and to reference it in subsequent API calls. The InterBase header file, *ibase.h*, contains a #define useful for declaring transaction handles.

When starting a transaction, optional transaction characteristics, such as access method and isolation level, can be passed to the start-up call through a transaction parameter buffer (TPB). Usually, one TPB is set up for each transaction, although transactions with the same operating characteristics can also share a TPB.

Declaring transaction handles

A transaction handle must be declared and initialized to zero before use. The following code illustrates how to declare and initialize a transaction handle:

```
#include "ibase.h"
. . .
/* Declare a transaction handle. */
isc_tr_handle tr1;
. . .
/* Initialize the handle. */
tr1 = 0L;
```

For more information about declaring, initializing, and using transaction handles, see **Chapter 5, "Working with Transactions."**

Setting up a TPB

A TPB is a byte array containing parameters that describe optional transaction characteristics. In these cases, the TPB must be set up and populated before starting a transaction. Parameters that can be passed to the TPB are defined in *ibase.h*.

For more information about setting up, populating, and using a TPB, see **Chapter 5, "Working with Transactions."**

Additional requirements

The following sections outline possible additional requirements for API applications developed on certain system platforms, such as Microsoft Windows, and for general classes of API functions, such as those that process DSQL statements.

Microsoft Windows requirements

InterBase client applications for Microsoft Windows have programming requirements specific to that environment and the C/C++ compilers available there.

The InterBase header file, *ibase.h*, provides prototypes of all API functions. For Windows applications, these prototypes make use of the following declarations:

```
#define ISC_FAR __far
#define ISC_EXPORT ISC_FAR __cdecl __loadds __export
```

For example, the **isc_attach_database()** prototype in *ibase.h* is:

```
ISC_STATUS ISC_EXPORT isc_attach_database(
ISC_STATvUS ISC_FAR *,
short,
char ISC_FAR,
isc_db_handle ISC FAR *,
short,
char ISC_FAR *);
```

When Windows client applications make calls and cast C datatypes, they should make explicit use of the ISC_FAR declaration.

Note The ISC_EXPORT keyword is omitted from the API function reference because on all non-Windows platforms it is undefined.

For more information about Windows requirements, see **Chapter 3, "Programming with the InterBase API."**

DSQL requirements

API applications that build or prompt for DSQL queries at run time require careful declaration, initialization, and population of extended SQL descriptor area (XSQLDA) structures for data transfer to and from the database. In addition, many API functions, such as **isc_dsql_allocate_statement()** and **isc_dsql_describe()**, also make use of statement handles for DSQL processing.

ibase.h provides typedefs for the XSQLDA structure, and its underlying structure, the XSQLVAR. It also provides a #define for the statement handle, a macro for allocating the appropriate amount of space for an instance of an XSQLDA in an application, and #defines for DSQL information parameters passed to **isc_dsql_sql_info()**.

The following code illustrates how to declare an XSQLDA structure for use in an application, and how to declare a statement handle:

```
#include <ibase.h>
. . .
XSQLDA *insqlda;
isc_stmt_handle sql_stmt;
. . .
```

For more information about DSQL programming with the API, see **Chapter 6, "Working with Dynamic SQL."**

Blob requirements

To work with Blob data that must be filtered, an API application must set up a Blob parameter buffer (BPB) for each Blob. A BPB is a variable-length byte vector declared in an application to store control information that dictates Blob access. The BPB can contain a number of constants, defined in *ibase.h*, that describe the Blob and the Blob subtypes that specify Blob filtering.

IMPORTANT Blob filtering is not available on NetWare servers.

Applications that work with Blob data in an international environment must also declare and populate a Blob descriptor that contains character set information for the Blob. The Blob descriptor structure is defined in *ibase.h*. To declare a Blob descriptor, an application must provide code like this:

```
#include <ibase.h>
. . .
ISC_BLOB_DESC to_desc;
```

Except on NetWare servers, where they are not supported, Blob filters enable a Blob to be translated from one format to another, such as from a compressed state to an decompressed state or vice versa. If Blob filters are desired, separate filter functions must be created and defined to the database to ensure their use when Blob data is accessed.

Finally, to access Blob data, applications must make extensive use of API DSQL functions.

For more information about working with Blob data and Blob filters, see **Chapter 7, "Working with Blob Data."**For more information about DSQL, see **Chapter 6, "Working with Dynamic SQL."**

Array requirements

API functions that handle array processing require the use of an array descriptor structure and array IDs, defined in *ibase.h*. In addition, applications accessing arrays must make extensive use of API DSQL functions.

The following code illustrates how to declare an array descriptor and array ID variable, and how to initialize an array ID to zero before use:

```
#include <ibase.h>
. . .
ISC_ARRAY_DESC desc;
ISC_QUAD array_id;
. . .
array_id = 0L;
. . .
```

For more information about working with arrays, see **Chapter 8, "Working with Array Data."**For more information about DSQL, see **Chapter 6, "Working with Dynamic SQL."**

Event requirements

InterBase events are messages passed from a trigger or stored procedure to an application to announce the occurrence of specified conditions or actions, usually database changes such as insertions, modifications, or deletions of records.

Before an application can respond to an event, it must register interest in an event. To register interest in an event, the application must establish and populate two event parameter buffers (EPBs), one for holding the initial occurrence count values for each event of interest, and another for holding the changed occurrence count values. These buffers are passed as parameters to several API event functions, and are used to determine which events have occurred.

In C, each EPB is declared as a char pointer, as follows:

```
char *event_buffer, *result_buffer;
```

Once the buffers are declared, **isc_event_block()** is called to allocate space for them, and to populate them with starting values.

For more information about events, see **Chapter 11, “Working with Events.”**

Error-handling requirements

Most API functions return status information in an *error status vector*, an array of 20 longs. To handle InterBase error conditions, should they arise, applications should declare a status vector as follows:

```
#include <ibase.h>
. . .
ISC_STATUS status_vector[20];
```

ISC_STATUS is a #define in *ibase.h* provided for programming convenience and platform independence.

ibase.h also contains #defines for all InterBase error conditions. Applications can use API error-handling functions to construct error messages from the status vector that are based on these error conditions, or can examine the status vector directly for particular error conditions using the #defines in place of error numbers. Using #defines in this manner makes source code easier to understand and maintain.

For more information about error handling, see **Chapter 11, “Working with Events.”**

Compiling and linking

On most development platforms, an API application is compiled like any standard C or C++ application. For more information about a particular compiler, consult the compiler's documentation.

On most platforms, InterBase supports dynamic linking of its library at run time. One exception to this scenario is on Microsoft Windows, where an application must explicitly link to the InterBase library (*gds32.lib* or *gds32_ms.lib*).

On Microsoft Windows, there are particular compiling options to be aware of.For more information about linking under Windows, see **Chapter 3, "Programming with the InterBase API."**

For all other platforms, see the InterBase *Programmer's Guide* for specific compiling and linking guidelines.

CHAPTER

3 Programming with the InterBase API

This chapter provides information specific to programming InterBase applications on a client with C/C++. It assumes familiarity with Borland C/C++ or Microsoft C/C++, InterBase, and the InterBase documentation set, particularly the *Language Reference*.

Basic procedure for application development

The basic steps in application development on the InterBase Windows Client are:

- Determine which client and server platforms the application will run on. InterBase clients and servers include Microsoft Windows 95, Windows NT, and Unix. An older version of the InterBase server is available for Novell NetWare 4.
- Code the application in C or C++. On UNIX, InterBase also supports compilers for COBOL, ADA, and FORTRAN.
- Compile and link the application.
- Test and debug the application.
- Deploy the application on the production client platform.

Supported development environments

The InterBase client library enables developers to design InterBase SQL client applications that connect to remote InterBase servers on Windows 95, Windows NT, Unix, or NetWare.

See the *Operations Guide* for more specific information about this topic.

User name and password requirements

When an InterBase client application is compiled, linked, and run, the client must *always* send a valid user name and password combination to the InterBase server. The server checks the user name and password against the user name and password combinations stored in its security database. If a match is found, the client can attach to InterBase databases on the server. If a match is not found, attachment is denied.

For a successful attachment to occur, the following steps must be taken:

1. A user with SYSDBA privileges must add a client's user name and password to the server's security database (*isc4.gdb*). Use the Server Manager to do this on Windows platforms. On UNIX, use the **gsec** utility.
2. The client must send a valid user name and password combination to the server. Password is case-sensitive.

Note Under some circumstances, you can connect to a database even if you don't have a user name in the InterBase security database. In order for this to happen, the following things must be true:

· Both the client and server are running under UNIX

· Your current login exists on the server host

· You are logging in from a trusted client; a trusted client is one that is listed in the */etc/hosts.equiv* or */etc/gds_hosts.equiv* file on the server or in the *.rhosts* file in your home directory on the server

· You have *not* specified a user name and password in the connect string

Specifying user name and password

A client application must specify a user name and password when it attaches to a database. Failure to provide a valid user name and password combination results in an error. Use the following methods to provide user names and passwords:

- Create a database parameter block (DPB) with *isc_dpb_user_name* and *isc_dpb_password*, and pass the parameter block using **isc_attach_database()**.
- Add *isc_dpb_user_name* and *isc_dpb_password* parameters to an existing DPB with **isc_expand_dpb()**.

For more information about the DPB, **isc_attach_database()**, and **isc_expand_dpb()**, see **Chapter 4, "Working with Databases."**

Using environment variables

InterBase client applications can use three environment variables to establish program parameters. These variables must be set so that they are available to the application when it is running. For example, setting these variables within a DOS window after Windows has been started does not affect any Windows programs, but affects DOS applications in that window.

The following table summarizes these variables and their uses:

Variable	Purpose	Example
ISC_DATABASE	Specifies a default server and database directory to use on the remote server	SET ISC_DATABASE = *ingold:/usr/interbase/examples*
ISC_USER	Specifies a user name for the PC client application	SET ISC_USER = HERMES
ISC_PASSWORD	Specifies a case-sensitive password for the PC client application	SET_PASSWORD = Ichneumon

TABLE 3.1 Environment variables used by InterBase

The ISC_USER and ISC_PASSWORD environment variables are used together to establish a valid user name and password combination to pass to the remote InterBase database server.

IMPORTANT Do not use the ISC_PASSWORD environment variable when security is a concern. Anyone with access to a client where an ISC_PASSWORD environment variable is defined in a file such as *autoexec.bat* can easily view the password.

Setting a default database directory

To connect automatically to a default database directory on a remote server, create the ISC_DATABASE environment variable and set it to the full path specification for the desired database directory, including host and path names.

Note Host name specification is specific to the server's operating system and network protocol. The host syntax in the previous example is for a generic Unix server. For other servers and operating systems, see that system's reference manuals.

Setting a user name and password

To set up a default user name and password for use on a PC client, create two environment variables, ISC_USER, and ISC_PASSWORD.

Even if ISC_USER and ISC_PASSWORD are set, a different user name and password may be specified in a DPB used as an argument to **isc_attach_database()**. A user name or password specified in a database parameter block overrides the OS environment variables.

Note Using environment variables in this manner is not secure, and therefore not recommended.

Datatypes

InterBase supports a wide variety of datatypes for application development. These datatypes are defined in a typedef to be platform-independent. The InterBase client libraries are also compiled with packed data structures to be compatible with a variety of platforms.

For more information about InterBase datatypes, see the *Language Reference*.

Calling conventions

Conventions for calling functions vary from platform to platform. Specifically:

- On UNIX platforms, use the C calling conventions (cdecl) in all cases.
- On Windows 95 and Windows NT, use the standard calling conventions (_stdcall) for all functions that have a fixed number of arguments. There are only three functions that have a variable number of arguments. For these three—**isc_start_transaction(), isc_expand_dpb()**, and **isc_event_block()**—use the cdecl conventions.

Building applications

This section discusses compilers and libraries that are needed to build InterBase applications.

HELP WITH LINKING AND COMPILING On each platform, there is a *makefile* in the *examples* directory that contains detailed platform-specific information about linking and compiling. Open the makefile in a text editor to access the information.

Compilers

The import libraries included with InterBase have been tested with the following compilers:

Windows platforms

- Borland C++ 5.0
- Microsoft Visual C++ 2.0
- Microsoft Visual C++ 4.0

Solaris

C	SPARCWorks SC4.2 C compiler
C++	SPARCWorks SC3.0.1 C++ compiler
COBOL	MicroFocus Cobol 4.0
ADA	SPARCWorks SC4.0 Ada compiler
FORTRAN	SPARCWorks SC4.0 Fortran compiler

HP-UX

C	HP C/HP-UX Version A.10.32
C++	HP C++/HP-UX Version A.10.22
COBOL	MicroFocus Cobol 4.0
ADA	Alsys Ada - AdaWorld V5.5.4
FORTRAN	HP Fortran/9000 10.20 Release

Linking

The InterBase library files reside in the *lib* subdirectory of the installation directory. Applications must link with the InterBase client library. This library name varies depending on the platform and the compiler.

Platform/compiler	InterBase library file
Windows/Borland C++	*gds32.lib*
Windows/Microsoft Visual C++ 2.0 and 4.0	*gds32_ms.lib*
Solaris/all	*gdsmt*
HPUX/all	*gds*

TABLE 3.2 InterBase library file names

Borland compilers earlier than 5.0 do not work with *gds32.lib*.

Include files

Applications must include the *ibase.h* header file to pick up the InterBase type definitions and function prototypes. This file is in the *include* subdirectory of the InterBase install directory.

On UNIX platforms, the *gds.h* file is available in the installation directory for backward compatibility.

Using Microsoft C++

Use the following options when compiling applications with Microsoft C++:

Option	Action
c	Compile without linking (DLLs only)
Zi	Generate complete debugging information
DWIN32	Defines "WIN32" to be the null string
D_MT	Use a multi-thread, statically-linked library

TABLE 3.3 Microsoft C compiler options

For example, these commands use the Microsoft compiler to build a DLL that uses InterBase:

```
cl -c -Zi -DWIN32 -D_MT -LD udf.c
   lib -out:udf.lib -def:funclib.def -machine:i586 -subsystem:console
   link -DLL -out:funclib.dll -DEBUG:full,mapped -DEBUGTYPE:CV
   -machine:i586 -entry:_DllMainCRTStartup@12 -subsystem:console
   -verbose udf.obj udf.exp gds32.lib ib_util_ms.lib crtdll.lib
```

This command builds an InterBase executable using the Microsoft compiler:

```
cl -Zi -DWIN32 -D_MT -MD udftest.c udf.lib gds32.lib
   ib_util_ms.lib crtdll.lib
```

Note See **"Creating user-defined functions" on page 188** of the *Data Definition Guide*, **Chapter 5, "User-Defined Functions"** of the *Language Reference*, and **Chapter 10, "Working with User-Defined Functions"** in the *Programmer's Guide* for more about compiling and linking user-defined libraries.

Using the Dynamic Runtime Library If you are

· using a Microsoft Visual C++ 2.0 or Microsoft Visual C++ 4.0

· compiling and linking separately, and

· using the Dynamic Runtime Library (*msvcrt20.dll* or *msvcrt40.dll*)

you need to use the /MD compiler flag to compile with the run time library (RTL), as well as linking with the correct import library.

Using Borland C/C++

Use the following options when compiling applications with Borland C++:

Option	Action
v	Turns on source debugging
a4	Structure padding/byte alignment
DWIN32	Defines the string “WIN32”; with no argument, it defines it to the null string
tWM	Makes the target multi-threaded
tWC:	Makes the target a console .EXE with all functions exportable; cannot be used with the *-tWCD* option
tWCD	Makes the target a console .DLL with all functions exportable; cannot be used with the *-tWC* option

TABLE 3.4 Borland C compiler options

The following command creates a DLL named *funclib.dll* from a source file named *udf.c*:

```
implib mygds32.lib \interbas\bin\gds32.dll
bcc32 -v -a4 -DWIN32 -tWM -tWCD -efunclib.dll udf.c mygds32.lib
```

The following commands create an InterBase executable named *udftest.exe* (which calls into *funclib.dll*) from a source file named *udftest.e* containing embedded SQL commands.

```
implib udf.lib funclib.dll
gpre -e udftest.e
bcc32 -v -a4 -DWIN32 -tWM -tWC udftest.c udf.lib mygds32.lib
```

When linking applications with Borland C command line linker, use the */c* option (case sensitive link).

Note There are equivalent general linker options within the Borland Integrated Development Environment (IDE). The default in the IDE is case-sensitive link (*/c* option) alone, which causes unresolved linker errors for all of the InterBase entry points.

Setting up the Integrated Development Environment (IDE)

The Borland Integrated Development Environment (IDE) offers options that are equivalent to the command line options.

IDE default

The case-sensitive link (*/c* option) is the default in the IDE.

IDE Project Options dialog box

Choose the following options from the IDE Project Options dialog box. The corresponding command-line option is also listed.

DIRECTORIES

Include directory: *c:\Program Files\InterBase Corp\InterBase\include*

Library directory: *c:\Program Files\InterBase Corp\InterBase\lib*

Note This path specification assumes that InterBase was installed in the *c:\Program Files\InterBase Corp\InterBase* directory.

COMPILER

Source language compliance: Borland extensions

32-bit Compiler

Data alignment: Byte (*-a4* option for 4 byte alignment)

LINKER

Choose Case-sensitive link ON (*/c* option).

The module definition file

Creating a module definition file can solve certain issues that arise during linking and compiling with the Borland C++ Builder:

- Set the STACKSIZE parameter to at least 10 kilobytes (10,240 bytes); 16 kilobytes (16,384 bytes) is recommended. A sample *.def* file is included in the *examples* subdirectory of the InterBase installation directory.
- Because the Borland C++Builder prepends an underscore to some API functions that *gds32.dll* exports without the underscore, you may need to add aliases for these functions to your module definition file, as in the following example:

```
IMPORTS
    _isc_start_transaction = GDS32.isc_start_transaction
```

Using dynamic link libraries (DLLs)

InterBase applications use the *gds32.dll* dynamic link library, which in turn loads the appropriate network DLLs. These DLLs unload automatically when the last calling application terminates. If the calling application exits abnormally (for example, from a protection fault), it is possible that DLLs will not be unloaded from memory. If this occurs, exit and restart Windows to free the resources.

Example programs

Example programs demonstrating how to use the InterBase API are included in the *examples* subdirectory of the InterBase installation directory. There is also a sample *.def* file.

On NT, there are two make files, *makefile.bc* for the Borland compiler and linker, and *makefile.msc* for the Microsoft compiler and linker. In both files, you must modify the IBASE environment variable to point to an absolute path.

In the *.bc* make file, modify the BCDIR variable to point to the absolute path to the Borland compiler and linker.

In the *.msc* make file, modify the MSCDIR variable to point to the absolute path to the Microsoft compiler and linker.

To build the example applications on NT using Borland C++, use the following command:

```
make -B -f makefile.bc all
```

To build the example applications using Microsoft C++, use this command:

```
nmake -B -f makefile.msc all
```

On UNIX systems, the command to build the example applications is as follows:

```
make all
```

CHAPTER

4

Working with Databases

This chapter describes how to set up a *database parameter buffer* (DPB) that specifies database attachment parameters, how to set up and initialize database handles, and how to use the five API functions that control database access. It also explains how to set up item request and return buffers prior to retrieving information about an attached database.

The following table lists the API functions for working with databases. The functions are listed in the order that they typically appear in an application.

Call	Purpose
isc_expand_dpb()	Specifies additional parameters for database access, such as user names and passwords elicited from a user at run time; uses a previously declared and populated DPB
isc_attach_database()	Connects to a database and establishes initial parameters for database access, such as number of cache buffers to use; uses a previously declared and populated DPB
isc_database_info()	Retrieves requested information about an attached database, such as the version of the on-disk structure (ODS) that it uses
isc_detach_database()	Disconnects from an attached database and frees system resources allocated to that attachment
isc_drop_database()	Deletes a database and any support files, such as shadow files

TABLE 4.1 API database functions

Connecting to databases

Connecting to one or more databases is a four-step process:

1. Creating and initializing a database handle for each database to be attached.
2. Creating and populating a DPB for each database to be attached.
3. Optionally calling **isc_expand_dpb()** prior to actual attachment to add more database parameters to a previously created and populated DPB.
4. Calling **isc_attach_database()** for each database to which to connect.

These steps are described in the following sections of this chapter.

Creating database handles

Every database that is accessed in an application must be associated with its own *database handle*, a pointer to a FILE structure that is used by all API database functions. The *ibase.h* header file contains the following C typedef declaration for database handles:

```
typedef void ISC_FAR *isc_db_handle;
```

To use this typedef for declaring database handles in an application, include *ibase.h* in each source file module:

```
#include <ibase.h>
```

Declaring database handles

To establish database handles for use, declare a variable of type *isc_db_handle* for each database that will be accessed at the same time. The following code declares two handles:

```
#include <ibase.h>
. . .
isc_db_handle db1;
iac_db_handle db2;
```

Once a database is no longer attached, its handle can be assigned to a different database in a subsequent attachment. If an application accesses several databases, but only accesses a subset of databases at the same time, it is only necessary to declare as many handles as there will be simultaneous database accesses. For example, if an application accesses a total of three databases, but only attaches to two of them at a time, only two database handles need be declared.

Initializing database handles

Before a database handle can be used to attach to a database, it must be set to zero. The following code illustrates how two database handles are set to zero:

```
#include <ibase.h>
. . .
isc_db_handle db1;
isc_db_handle db2;
. . .
/* Set database handles to zero before attaching to a database. */
db1 = 0L;
db2 = 0L;
```

Once a database handle is initialized to zero, it can be used in a call to **isc_attach_database()** to establish a database connection. If a nonzero database handle is passed to **isc_attach_database()**, the connection fails and an error code is returned. For more information about establishing a database connection with **isc_attach_database()**, see **"Attaching to a database" on page 47**.

Creating and populating a DPB

Database attachments can optionally be tailored in many ways by creating a database parameter buffer (DPB), populating it with desired database characteristics, and passing the address of the DPB to **isc_attach_database()**.

For example, the DPB can contain a user name and password for attaching to a database on a remote server, and it might also contain a parameter that activates a database shadow file. For a list of all possible DPB parameters, see **Table 4.2, "DPB parameters," on page 43**.

Usually a separate DPB is created for each database attachment, but if different attachments use the same set of parameters, they can share a DPB. If a DPB is not created or is not passed to **isc_attach_database()**, the database attachment uses a default set of parameters.

TIP Some of the DPB parameters correspond directly to gfix options. In fact, that's how gfix is implemented: it sets certain DPB parameters and attaches to a database, where it performs the requested operation (sweep, set async writes, shutdown, or whatever).

A DPB is a char array variable, specifically declared in an application, that consists of the following parts:

1. A byte specifying the version of the parameter buffer, always the compile-time constant, *isc_dpb_version1*.
2. A contiguous series of one or more *clusters* of bytes, each describing a single parameter.

Each cluster consists of the following parts:

1. A one-byte parameter type. There are compile-time constants defined for all the parameter types (for example, *isc_dpb_num_buffers*).
2. A one-byte number specifying the number of bytes that follow in the remainder of the cluster.
3. A variable number of bytes, whose interpretation (for example, as a number or as a string of characters) depends on the parameter type.

For example, the following code creates a DPB with a single parameter that sets the number of cache buffers to use when connecting to a database:

```
char dpb_buffer[256], *dpb, *p;
short dpb_length;
/* Construct the database parameter buffer. */
dpb = dpb_buffer;
*dpb++ = isc_dpb_version1;
```

```
*dpb++ = isc_num_buffers;
*dpb++ = 1;
*dpb++ = 90;
dpb_length = dpb - dpb_buffer;
```

IMPORTANT All numbers in the database parameter buffer must be represented in a generic format, with the least significant byte first, and the most significant byte last. Signed numbers should have the sign in the last byte. The API function **isc_vax_integer()** can be used to reverse the byte order of a number. For more information, see **"isc_vax_integer()" on page 332**.

The following table lists DPB items by purpose:

User validation	
User name	*isc_dpb_user_name*
Password	*isc_dpb_password*
Encrypted password	*isc_dpb_password_enc*
Role name	*isc_dpb_sql_role_name*
System database administrator's user name	*isc_dpb_sys_user_name*
Authorization key for a software license	*isc_dpb_license*
Database encryption key	*isc_dpb_encrypt_key*
Environmental control	
Number of cache buffers	*isc_dpb_num_buffers*
dbkey context scope	*isc_dpb_dbkey_scope*
System management	
Force writes to the database to be done asynchronously or synchronously	*isc_dpb_force_write*
Specify whether or not to reserve a small amount of space on each database page for holding backup versions of records when modifications are made	*isc_dpb_no_reserve*
System management	
Specify whether or not the database should be marked as damaged	*isc_dpb_damaged*
Perform consistency checking of internal structures	*isc_dpb_verify*

TABLE 4.2 DPB parameters

Shadow control	
Activate the database shadow, an optional, duplicate, in-sync copy of the database	*isc_dpb_activate_shadow*
Delete the database shadow	*isc_dpb_delete_shadow*
Replay logging system control	
Activate a replay logging system to keep track of all database calls	*isc_dpb_begin_log*
Deactivate the replay logging system	*isc_dpb_quit_log*
Character set and message file specification	
Language-specific message file	*isc_dpb_lc_messages*
Character set to be utilized	*isc_dpb_lc_ctype*

TABLE 4.2 DPB parameters (*continued*)

The following table lists DPB parameters in alphabetical order. For each parameter, it lists its purpose, the length, in bytes, of any values passed with the parameter, and the value to pass.

Parameter	**Purpose**	**Length**	**Value**
isc_dpb_activate_shadow	Directive to activate the database shadow, which is an optional, duplicate, in-sync copy of the database	1 (Ignored)	0 (Ignored)
isc_dpb_damaged	Number signifying whether or not the database should be marked as damaged 1 = mark as damaged 0 = do *not* mark as damaged	1	0 or 1
isc_dpb_dbkey_scope	Scope of *dbkey* context. 0 limits scope to the current transaction, 1 extends scope to the database session	1	0 or 1
isc_dpb_delete_shadow	Directive to delete a database shadow that is no longer needed	1(Ignored)	0 (Ignored)
isc_dpb_encrypt_key	String encryption key, up to 255 characters	Number of bytes in string	String containing key

TABLE 4.3 Alphabetical list of DPB parameters

Parameter	Purpose	Length	Value
isc_dpb_force_write	Specifies whether database writes are synchronous or asynchronous. 0 = asynchronous; 1 = synchronous	1	0 or 1
isc_dpb_lc_ctype	String specifying the character set to be utilized	Number of bytes in string	String containing character set name
isc_dpb_lc_messages	String specifying a language-specific message file	Number of bytes in string	String containing message file name
isc_dpb_license	String authorization key for a software license	Number of bytes in string	String containing key
isc_dpb_no_reserve	Specifies whether or not a small amount of space on each database page is reserved for holding backup versions of records when modifications are made; keep backup versions on the same page as the primary record to optimize update activity 0 (default) = reserve space 1= do not reserve space	1	0 or 1
isc_dpb_num_buffers	Number of database cache buffers to allocate for use with the database; default=75	1	Number of buffers to allocate
isc_dpb_password	String password, up to 255 characters	Number of bytes in string	String containing password
isc_dpb_password_enc	String encrypted password, up to 255 characters	Number of bytes in string	String containing password
isc_dpb_sys_user_name	String system DBA name, up to 255 characters	Number of bytes in string	String containing SYSDBA name
isc_dpb_user_name	String user name, up to 255 characters	Number of bytes in string	String containing user name

TABLE 4.3 Alphabetical list of DPB parameters (*continued*)

Note Some parameters, such as *isc_dpb_delete_shadow*, are directives that do not require additional parameters. Even so, you *must* still provide length and value bytes for these parameters. Set length to 1 and value to 0. InterBase ignores these parameter values, but they are required to maintain the format of the DPB.

Adding parameters to a DPB

Sometimes it is useful to add parameters to an existing DPB at run time. For example, when an application runs, it might determine a user's name and password and supply those values dynamically. The **isc_expand_dpb()** function can be used to pass the following additional parameters to a previously created and populated DPB at run time:

Parameter	Purpose
isc_dpb_user_name	String user name, up to 255 characters
isc_dpb_password	String password, up to 255 characters
isc_dpb_lc_messages	String specifying a language-specific message file
isc_dpb_lc_ctype	String specifying the character set to be utilized

TABLE 4.4 DPB parameters recognized by **isc_expand_dpb()**

IMPORTANT If you expect to add any of these parameters at run time, then create a larger than necessary DPB before calling **isc_expand_dpb()**, so that this function does not need to reallocate DPB storage space at run time. **isc_expand_dbp()** can reallocate space, but that space is not automatically freed when the database is detached.

isc_expand_dpb() requires the following parameters:

Parameter	Type	Description
dpb	char **	Pointer to a DPB
dpb_size	unsigned short *	Pointer to the current size, in bytes, of the DPB
…	char *	Pointers to item type and items to add to the DPB

TABLE 4.5 **isc_expand_dbp()** parameters

The third parameter in the table, …, indicates a variable number of replaceable parameters, each with different names, but each a character pointer.

The following code demonstrates how **isc_expand_dpb()** is called to add a user name and password to the DPB after they are elicited from a user at run time:

```
char dpb_buffer[256], *dpb, *p;
char uname[256], upass[256];
short dpb_length;

/* Construct a database parameter buffer. */
dpb = dpb_buffer;
*dpb++ = isc_dpb_version1;
*dpb++ = isc_num_buffers;
*dpb++ = 1;
*dpb++ = 90;
dpb_length = dpb - dpb_buffer;
/* Now ask user for name and password. */
prompt_user("Enter your user name: ");
gets(uname);
prompt_user("\nEnter your password: ");
gets(upass);
/* Add user name and password to DPB. */
dpb = dbp_buffer;
isc_expand_dpb(&dpb, &dpb_length,
   isc_dpb_user_name, uname,
   isc_dpb_password, upass,
   NULL);
```

Attaching to a database

After creating and initializing a database handle, and optionally setting up a DPB to specify connection parameters, use **isc_attach_database()** to establish a connection to an existing database. Besides allocating system resources for the database connection, **isc_attach_database()** also associates a specific database with a database handle for use in subsequent API calls that require a handle.

isc_attach_database() expects six parameters:

- A pointer to an error status array, where attachment errors can be reported should they occur.
- The length, in bytes, of the database name for the database to open. If the database name includes a node name and path, these elements must be counted in the length argument.

- A string containing the name of the database to attach. The name can include a node name and path specification.
- A pointer to a previously declared and initialized database handle with which to associate the database to attach. All subsequent API calls use the handle to specify access to this database.
- The length, in bytes, of the DPB. If no DPB is passed, set this value to zero.
- A pointer to the DPB. If no DPB is passed, set this to NULL.

Each database attachment requires a separate call to **isc_attach_database()**.

The following code establishes an attachment to the InterBase example database, *employee.gdb*, and specifies a DPB to use for the attachment:

```
#include <ibase.h>
. . .
isc_db_handle db1;
char dpb_buffer[256], *dpb, *p;
short dpb_length;
char *str = "employee.gdb";
ISC_STATUS status_vector[20];
. . .
/* Set database handle to zero before attaching to a database. */
db1 = 0L;
/* Initialize the DPB. */
dpb = dpb_buffer;
*dpb++ = isc_dpb_version1;
*dpb++ = isc_num_buffers;
*dpb++ = 1;
*dpb++ = 90;
dpb_length = dpb - dpb_buffer;
/* Attach to the database. */
isc_attach_database(status_vector, strlen(str), str, &db1,
dpb_length,
   dbp_buffer);
if (status_vector[0] == 1 && status_vector[1])
{
   error_exit();
}
. . .
```

The following code illustrates how to attach to a database without passing a DPB:

```
#include <ibase.h>
```

```
. . .
isc_db_handle db1;
char *str = "employee.gdb";
ISC_STATUS status_vector[20];
. . .
/* Set database handle to zero before attaching to a database. */
db1 = 0L;
/* Attach to the database. */
isc_attach_database(status_vector, strlen(str), str, &db1, 0, NULL);
if (status_vector[0] == 1 && status_vector[1])
{
   error_exit();
}
. . .
```

Requesting information about an attachment

After an application attaches to a database, it may need information about the attachment. The **isc_database_info()** call enables an application to query for attachment information, such as the version of the on-disk structure (ODS) used by the attachment, the number of database cache buffers allocated, the number of databases pages read from or written to, or write-ahead log information.

In addition to a pointer to the error status vector and a database handle, **isc_database_info()** requires two application-provided buffers, a request buffer, where the application specifies the information it needs, and a result buffer, where InterBase returns the requested information. An application populates the request buffer with information prior to calling **isc_database_info()**, and passes it both a pointer to the request buffer, and the size, in bytes, of that buffer.

The application must also create a result buffer large enough to hold the information returned by InterBase. It passes both a pointer to the result buffer, and the size, in bytes, of that buffer, to **isc_database_info()**. If InterBase attempts to pass back more information than can fit in the result buffer, it puts the value, *isc_info_truncated*, defined in *ibase.h*, in the final byte of the result buffer.

Requesting buffer items and result buffer values

The request buffer is a char array into which is placed a sequence of byte values, one per requested item of information. Each byte is an *item type*, specifying the kind of information desired. Compile-time constants for all item types are defined in *ibase.h*.

The result buffer returns a series of *clusters* of information, one per item requested. Each cluster consists of three parts:

1. A one-byte *item return type*. There are compile-time constants defined for all the item return types in *ibase.h*.
2. A two-byte number specifying the number of bytes that follow in the remainder of the cluster.
3. A *value*, stored in a variable number of bytes, whose interpretation (for example, as a number or as a string of characters) depends on the item return type.

A calling program is responsible for interpreting the contents of the result buffer and for deciphering each cluster as appropriate. In many cases, the value simply contains a number or a string (sequence of characters). But in other cases, the value is a number of bytes whose interpretation depends on the item return type.

The clusters returned to the result buffer are not aligned. Furthermore, all numbers are represented in a generic format, with the least significant byte first, and the most significant byte last. Signed numbers have the sign in the last byte. Convert the numbers to a datatype native to your system, if necessary, before interpreting them. The API call, **isc_vax_integer()**, can be used to perform the conversion.

Database characteristics

Several items are available for determining database characteristics, such as its size and major and minor ODS version numbers. The following table lists the request buffer items that can be passed, and the information returned in the result buffer for each item type:

Request buffer item	Result buffer contents
isc_info_allocation	Number of database pages allocated
isc_info_base_level	Database version (level) number: • 1 byte containing the number 1 • 1 byte containing the version number

TABLE 4.6 Database information items for database characteristics

Request buffer item	Result buffer contents
isc_info_db_id	• Database file name and site name: • 1 byte containing the number 2 for a local connection or 4 for a remote connection • 1 byte containing the length, *d*, of the database file name in bytes • A string of *d* bytes, containing the database file name • 1 byte containing the length, *l*, of the site name in bytes • A string of *l* bytes, containing the site name
isc_info_implementation	Database implementation number: • 1 byte containing a 1 • 1 byte containing the implementation number • 1 byte containing a "class" number, either 1 or 12
isc_info_no_reserve	0 or 1 • 0 indicates space is reserved on each database page for holding backup versions of modified records [Default] • 1 indicates no space is reserved for such records
isc_info_ods_minor_version	On-disk structure (ODS) minor version number; an increase in a minor version number indicates a non-structural change, one that still allows the database to be accessed by database engines with the same major version number but possibly different minor version numbers
isc_info_ods_version	ODS major version number; databases with different major version numbers have different physical layouts A database engine can access only databases with a particular ODS major version number; trying to attach to a database with a different ODS number results in an error
isc_info_page_size	Number of bytes per page of the attached database; use with *isc_info_allocation* to determine the size of the database
isc_info_version	Version identification string of the database implementation: • 1 byte containing the number 1 • 1 byte specifying the length, *n*, of the following string • *n* bytes containing the version identification string

TABLE 4.6 Database information items for database characteristics (*continued*)

Environmental characteristics

Several items are provided for determining environmental characteristics, such as the amount of memory currently in use, or the number of database cache buffers currently allocated. These items are described in the following table:

Request buffer item	Result buffer contents
isc_info_current_memory	Amount of server memory (in bytes) currently in use
isc_info_forced_writes	Number specifying the mode in which database writes are performed (0 for asynchronous, 1 for synchronous)
isc_info_max_memory	Maximum amount of memory (in bytes) used at one time since the first process attached to the database
isc_info_num_buffers	Number of memory buffers currently allocated
isc_info_sweep_interval	Number of transactions that are committed between "sweeps" to remove database record versions that are no longer needed
isc_info_user_names	Names of all the users currently attached to the database; for *each* such user, the result buffer contains an *isc_info_user_names* byte followed by a 1-byte length specifying the number of bytes in the user name, followed by the user name

TABLE 4.7 Database information items for environmental characteristics

Note Not all environmental information items are available on all platforms.

Performance statistics

There are four items that request performance statistics for a database. These statistics accumulate for a database from the moment it is first attached by any process until the last remaining process detaches from the database.

For example, the value returned for *isc_info_reads* is the number of reads since the current database was first attached, that is, an *aggregate* of all reads done by all attached processes, rather than the number of reads done for the calling program since it attached to the database.

Table 4.8 lists the request performance statistics are summarized in the following table:

Request buffer item	**Result buffer contents**
isc_info_fetches	Number of reads from the memory buffer cache
isc_info_marks	Number of writes to the memory buffer cache
isc_info_reads	Number of page reads
isc_info_writes	Number of page writes

TABLE 4.8 Database information items for performance statistics

Database operation counts

Several information items are provided for determining the number of various database operations performed by the currently attached calling program. These values are calculated on a per-table basis.

When any of these information items is requested, InterBase returns to the result buffer:

- 1 byte specifying the item type (for example, *isc_info_insert_count*).
- 2 bytes telling how many bytes compose the subsequent value pairs.
- A pair of values for each table in the database on which the requested type of operation has occurred since the database was last attached.

Each pair consists of:

- 2 bytes specifying the table ID.
- 4 bytes listing the number of operations (for example, inserts) done on that table.

TIP To determine an actual table name from a table ID, query the system table, RDB$RELATION.

The following table describes the items which return count values for operations on the database:

Request buffer item	Result buffer contents
isc_info_backout_count	Number of removals of a version of a record
isc_info_delete_count	Number of database deletes since the database was last attached
isc_info_expunge_count	Number of removals of a record and all of its ancestors, for records whose deletions have been committed
isc_info_insert_count	Number of inserts into the database since the database was last attached
isc_info_purge_count	Number of removals of old versions of fully mature records (records committed, resulting in older—ancestor—versions no longer being needed)
isc_info_read_idx_count	Number of reads done via an index since the database was last attached
isc_info_read_seq_count	Number of sequential database reads, that is, the number of sequential table scans (row reads) done on each table since the database was last attached
isc_info_update_count	Number of database updates since the database was last attached

TABLE 4.9 Database information items for operation counts

isc_database_info() call example

The following code requests the page size and the number of buffers for the currently attached database, then examines the result buffer:

```
char db_items[] = {
   isc_info_page_size, isc_info_num_buffers,
   isc_info_end};
char res_buffer[40], *p, item;
int length;
SLONG page_size = 0L, num_buffers = 0L;
ISC_STATUS status_vector[20];

isc_database_info(
   status_vector,
   &handle,  /* Set in previous isc_attach_database() call. */
   sizeof(db_items),
   db_items,
   sizeof(res_buffer),
   res_buffer);
if (status_vector[0] == 1 && status_vector[1]) {
   /* An error occurred. */
   isc_print_status(status_vector);
   return(1);
};
/* Extract the values returned in the result buffer. */
for (p = res_buffer; *p != isc_info_end ; ) {
   item = *p++
   length = isc_vax_integer(p, 2);
   p += 2;
   switch (item){
       case isc_info_page_size:
           page_size = isc_vax_integer(p, length);
           break;
       case isc_info_num_buffers:
           num_buffers = isc_vax_integer(p, length);
           break;
       default:
           break;
   }
   p += length;
};
```

Disconnecting from databases

When an application is finished accessing a database, and any changes are committed or rolled back, the application should disconnect from the database, release system resources allocated for the attachment, and set the database handle to zero with a call to **isc_detach_database()**.

isc_detach_database() requires two arguments: a pointer to the error status vector, and a pointer to the handle of the database from which to detach. For example, the following statement detaches from the database pointed to by the database handle, *db1*:

```
isc_detach_database(status_vector, &db1);
```

Each database to detach requires a separate call to **isc_detach_database()**.

Deleting a database

To remove a database from the system if it is no longer needed, use **isc_drop_database()**. This function permanently wipes out a database, erasing its data, metadata, and all of its supporting files, such as secondary files, shadow files, and write-ahead log files.

A database can only be deleted if it is previously attached with a call to **isc_attach_database()**. The call to **isc_attach_database()** establishes a database handle for the database. That handle must be passed in the call to **isc_drop_database()**.

For example, the following code deletes the database pointed to by the database handle, *db1*:

```
#include <ibase.h>
. . .
isc_db_handle db1;
char *str = "employee.gdb";
ISC_STATUS status_vector[20];
. . .
/* Set database handle to zero before attaching to a database. */
db1 = 0L;
/* Attach to the database. */
isc_attach_database(status_vector, strlen(str), str, &db1, 0, NULL);
if (status_vector[0] == 1 && status_vector[1])
{
   error_exit();
}
isc_drop_database(status_vector, &db1);
if (status_vector[0] == 1 && status_vector[1])
{
   error_exit();
}
. . .
```

CHAPTER

5

Working with Transactions

This chapter describes how to set up a *transaction parameter buffer* (TPB) that contains parameters, how to set up and initialize transaction handles, and how to use the API functions that control transactions. It also explains how to retrieve a transaction ID.

All data definition and data manipulation in an application takes place in the context of one or more *transactions*, one or more statements that work together to complete a specific set of actions that must be treated as an atomic unit of work.

The following table summarizes the API functions most commonly used when working with transactions. Functions are listed in the order they typically appear in an application.

Function	Purpose
isc_start_transaction()	Starts a new transaction against one or more databases.; use a previously declared and populated TPB
isc_commit_retaining()	Commits a transaction's changes, and preserves the transaction context for further transaction processing
isc_commit_transaction()	Commits a transaction's changes, and ends the transaction
isc_rollback_transaction()	Rolls back a transaction's changes, and ends the transaction

TABLE 5.1 API transaction functions

In addition to these functions, the following table lists less frequently used API transaction functions in the order they typically appear when used:

Function	Purpose
isc_start_multiple()	Starts a new transaction against one or more databases; used instead of **isc_start_transaction()** for programming languages such as FORTRAN, that do not support variable numbers of arguments to functions
isc_prepare_transaction()	Performs the first phase of a two-phase commit, prior to calling **isc_commit_transaction()**; used only when it is absolutely necessary to override InterBase's automatic two-phase commit
isc_prepare_transaction2()	Performs the first phase of a two-phase commit, prior to calling **isc_commit_transaction()**; used only when absolutely necessary to override InterBase's automatic two-phase commit

TABLE 5.2 Additional API transaction functions

Starting transactions

Starting transactions is a three-step process:

1. Creating and initializing a transaction handle for each simultaneous transaction to be started.
2. Optionally creating and populating a TPB for each transaction.
3. Calling **isc_start_transaction()** for each transaction to start.

These steps are described in the following sections of this chapter.

Note Programmers writing applications that do not permit function calls to pass a variable number of parameters must use **isc_start_multiple()** instead of **isc_start_transaction()**.

Creating transaction handles

Every transaction that is used in an application must be associated with its own *transaction handle*, a pointer to an address that is used by all API transaction functions. The *ibase.h* header file contains the following C typedef declaration for transaction handles:

```
typedef void ISC_FAR *isc_tr_handle;
```

To use this typedef for declaring transaction handles in an application, include *ibase.h* in each source file module:

```
#include <ibase.h>
```

Declaring transaction handles

To establish transaction handles for use, declare a variable of type *isc_tr_handle* for each simultaneously active transaction. The following code declares two handles:

```
#include <ibase.h>
. . .
isc_tr_handle tr1;
iac_tr_handle tr2;
```

Once a transaction is committed or rolled back, its handle can be assigned to a different transaction in a subsequent call to **isc_start_transaction()**. If an application uses several transactions, but only starts a subset of transactions at the same time, it is only necessary to declare as many handles as there will be simultaneously active transactions. For example, if an application starts a total of three transactions, but only runs two of them at the same time, only two transaction handles need be declared.

Initializing transaction handles

Before a transaction handle can be used to start a new transaction, it must be set to zero. The following code illustrates how two transaction handles are set to zero:

```
#include <ibase.h>
. . .
isc_tr_handle tr1;
isc_tr_handle tr2;
. . .
/* Set transaction handles to zero before starting a transaction. */
tr1 = 0L;
tr2 = 0L;
```

Once a transaction handle is initialized to zero, it can be used in a call to **isc_start_transaction()** to establish a new transaction. If a nonzero transaction handle is passed to **isc_start_transaction()**, the startup fails and an error code is returned. For more information about starting a new transaction with **isc_start_transaction()**, see **"Calling isc_start_transaction( )" on page 67**.

Creating a transaction parameter buffer

The *transaction parameter buffer* (TPB) is an optional, application-defined byte vector, passed as an argument to **isc_start_transaction()**, that sets up a transaction's *attributes*, its operating characteristics, such as whether the transaction has read and write access to tables, or read-only access, and whether or not other simultaneously active transactions can share table access with the transaction. Each transaction may have its own TPB, or transactions that share operating characteristics can use the same TPB.

Note If a TPB is not created for a transaction, a NULL pointer must be passed to **isc_start_transaction()** in its place. A default set of attributes is automatically assigned to such transactions. For more information about the default TPB, see **"Using the default TPB" on page 67**.

A TPB is declared in a C program as a char array of one-byte elements. Each element is a parameter that describes a single transaction attribute. A typical declaration is as follows:

```
static char isc_tpb[] = {isc_tpb_version3,
    isc_tpb_write,
    isc_tpb_read_committed,
    isc_tpb_no_rec_version,
    isc_tpb_wait};
```

This example makes use of parameter constants defined in the InterBase header file, *ibase.h*. The first element in every TPB must be the *isc_tpb_version3* constant. The following table lists available TPB constants, describes their purposes, and indicates which constants are assigned as a default set of attributes when a NULL TPB pointer is passed to **isc_start_transaction()**:

Parameter	Description
isc_tpb_version3	InterBase version 3 transaction
isc_tpb_consistency	Table-locking transaction model
isc_tpb_concurrency	High throughput, high concurrency transaction with acceptable consistency; use of this parameter takes full advantage of the InterBase multi-generational transaction model [Default]
isc_tpb_shared	Concurrent, shared access of a specified table among all transactions; use in conjunction with *isc_tpb_lock_read* and *isc_tpb_lock_write* to establish the lock option [Default]

TABLE 5.3 TPB constants

Parameter	Description
isc_tpb_protected	Concurrent, restricted access of a specified table; use in conjunction with *isc_tpb_lock_read* and *isc_tpb_lock_write* to establish the lock option
isc_tpb_wait	Lock resolution specifies that the transaction is to wait until locked resources are released before retrying an operation [Default]
isc_tpb_nowait	Lock resolution specifies that the transaction is not to wait for locks to be released, but instead, a lock conflict error should be returned immediately
isc_tpb_read	Read-only access mode that allows a transaction only to select data from tables
isc_tpb_write	Read-write access mode of that allows a transaction to select, insert, update, and delete table data [Default]
isc_tpb_lock_read	Read-only access of a specified table. Use in conjunction with *isc_tpb_shared, isc_tpb_protected*, and *isc_tpb_exclusive* to establish the lock option.
isc_tpb_lock_write	Read-write access of a specified table. Use in conjunction with *isc_tpb_shared*, *isc_tpb_protected*, and *isc_tpb_exclusive* to establish the lock option [Default]
isc_tpb_read_committed	High throughput, high concurrency transaction that can read changes committed by other concurrent transactions. Use of this parameter takes full advantage of the InterBase multi-generational transaction model.
isc_tpb_rec_version	Enables an *isc_tpb_read_committed* transaction to read the most recently committed version of a record even if other, uncommitted versions are pending.
isc_tpb_no_rec_version	Enables an *isc_tpb_read_committed* transaction to read only the latest committed version of a record. If an uncommitted version of a record is pending and *isc_tpb_wait* is also specified, then the transaction waits for the pending record to be committed or rolled back before proceeding. Otherwise, a lock conflict error is reported at once.

TABLE 5.3 TPB constants (*continued*)

TPB parameters specify the following classes of information:

- *Transaction version number* is used internally by the InterBase engine. It is always be the first attribute specified in the TPB, and must always be set to *isc_tpb_version3.*

- *Access mode* describes the actions that can be performed by the functions associated with the transaction. Valid access modes are:

  isc_tpb_read

  isc_tpb_write

- *Isolation level* describes the view of the database given a transaction as it relates to actions performed by other simultaneously occurring transactions. Valid isolation levels are:

  isc_tpb_concurrency

  isc_tpb_consistency

  isc_tpb_read_committed, *isc_tpb_rec_version*

  isc_tpb_read_committed, *isc_tpb_no_rec_version*

- *Lock resolution* describes how a transaction should react if a lock conflict occurs. Valid lock resolutions are:

  isc_tpb_wait

  isc_tpb_nowait

- *Table reservation* optionally describes an access method and lock resolution for a specified table that the transaction accesses. When table reservation is used, tables are reserved for the specified access when the transaction is started, rather than when the transaction actually accesses the table. Valid reservations are:

  isc_tpb_shared, *isc_tpb_lock_write*

  isc_tpb_shared, *isc_tpb_lock_read*

  isc_tpb_protected, *isc_tpb_lock_write*

  isc_tpb_protected, *isc_tpb_lock_read*

TPB parameters are described in detail in the following sections.

Specifying the transaction version number

The first parameter in a TPB must always specify the version number for transaction processing. It must always be set to *isc_tpb_version3*. The following TPB declaration illustrates the correct use and position of this parameter:

```
static char isc_tpb[] = {isc_tpb_version3, ...};
```

Specifying access mode

The access mode parameter describes the actions a transaction can perform against a table. The default access mode, *isc_tpb_write*, enables a transaction to read data from a table and write data to it. A second access mode, *isc_tpb_read*, restricts table access to read only. For example, the following TPB declaration specifies a read-only transaction:

```
static char isc_tpb[] = {isc_tpb_version3, isc_tpb_read};
```

A TPB should only specify one access mode parameter. If more than one is specified, later declarations override earlier ones.

If a TPB is declared that omits the access mode parameter, InterBase interprets transaction access as read and write.

Specifying isolation level

The isolation level parameter specifies the view of the database permitted a transaction as it relates to actions performed by other simultaneously occurring transactions.

ISC_TPB_CONCURRENCY

By default, after a transaction starts it cannot access committed changes to a table made by other simultaneous transactions, even though it shares access to the table with them. Such a transaction has an isolation level of *isc_tpb_concurrency*, meaning it can have concurrent access to tables also accessed simultaneously by other transactions. The following declaration creates a TPB specifying an isolation level of *isc_tpb_concurrency*:

```
static char isc_tpb[] = {isc_tpb_version3,
    isc_tpb_write,
    isc_tpb_concurrency};
```

ISC_TPB_READ_COMMITTED

A second isolation level, *isc_tpb_read_committed*, offers all the advantages of the *isc_tpb_concurrency* isolation level and additionally enables a transaction to access changes committed by other simultaneous transactions. Two other parameters, *isc_tpb_rec_version*, and *isc_tpb_no_rec_version*, should be used with the *isc_tpb_read_committed* parameter. They offer refined control over the committed changes a transaction is permitted to access:

- *isc_tpb_no_rec_version*, the default refinement, specifies that a transaction can only read the latest version of a row. If a change to a row is pending, but not yet committed, the row cannot be read.
- *isc_tpb_rec_version* specifies that a transaction can read the latest committed version of a row, even if a more recent uncommitted version is pending.

The following declaration creates a TPB with a read committed isolation level, and specifies that the transaction can read the latest committed version of a row:

```
static char isc_tpb[] = {isc_tpb_version3,
    isc_tpb_write,
    isc_tpb_read_committed,
    isc_tpb_rec_version};
```

ISC_TPB_CONSISTENCY

InterBase also supports a restrictive isolation level. *isc_tpb_consistency* prevents a transaction from accessing tables if they are written to by other transactions; it also prevents other transactions from writing to a table once this transaction writes to it. This isolation level is designed to guarantee that if a transaction writes to a table before other simultaneous read and write transactions, then only it can change a table's data. Because it essentially restricts (and often prevents) shared access to tables, *isc_tpb_consistency* should be used with care.

A TPB should only specify one isolation mode parameter (and one refinement parameter, if isolation mode is *isc_tpb_read_committed*). If more than one is specified, later declarations override earlier ones.

If a TPB is declared that omits the isolation mode parameter, InterBase interprets it as *isc_tpb_concurrency.*

ISOLATION LEVEL INTERACTIONS

To determine the possibility for lock conflicts between two transactions accessing the same database, each transaction's isolation level and access mode must be considered. The following table summarizes possible combinations:

		isc_tpb_concurrency, isc_tpb_read_committed		**isc_tpb_consistency**	
		isc_tpb_write	**isc_tpb_read**	**isc_tpb_write**	**isc_tpb_read**
concurrency, read_committed	isc_tpb_write	Some simultaneous updates may conflict	—	Conflicts	Conflicts
	isc_tpb_read	—	—	—	—
consistency	isc_tpb_write	Conflicts	—	Conflicts	Conflicts
	isc_tpb_read	Conflicts	—	Conflicts	—

TABLE 5.4 Isolation level interaction with read and write operations

As this table illustrates, *isc_tpb_concurrency* and *isc_tpb_read_committed* transactions offer the least chance for conflicts. For example, if *t1* is an *isc_tpb_concurrency* transaction with *isc_tpb_write* access, and *t2* is an *isc_tpb_read_committed* transaction with *isc_tpb_write* access, *t1* and *t2* only conflict when they attempt to update the same rows. If *t1* and *t2* have *isc_tpb_read* access, they never conflict with other transactions.

An *isc_tpb_consistency* transaction with *isc_tpb_write* access is guaranteed that if it gains access to a table that it alone can update a table, but it conflicts with all other simultaneous transactions except for *isc_tpb_concurrency* and *isc_tpb_read_committed* transactions running in *isc_tpb_read* mode. An *isc_tpb_consistency* transaction with *isc_tpb_read* access is compatible with any other read-only transaction, but conflicts with any transaction that attempts to insert, update, or delete data.

Specifying lock resolution

The lock resolution parameter describes what happens if a transaction encounters an access conflict during a write operation (update and delete operations on existing rows). There are two possible choices for this parameter:

- *isc_tpb_wait*, the default, specifies that the transaction should wait until locked resources are released. Once the resources are released, the transaction retries its operation.
- *isc_tpb_nowait* specifies that the transaction should return a lock conflict error without waiting for locks to be released.

For example, the following declaration creates a TPB with write access, a concurrency isolation mode, and a lock resolution of *isc_tpb_nowait*:

```
static char isc_tpb[] = {isc_tpb_version3,
    isc_tpb_write,
    isc_tpb_concurrency,
    isc_tpb_nowait};
```

A TPB should only specify one lock resolution parameter. If more than one is specified, later declarations override earlier ones.

If a TPB is declared that omits the lock resolution parameter, InterBase interprets it as *isc_tpb_concurrency.*

Specifying table reservation

Ordinarily, transactions gain specific access to tables only when they actually read from or write to them. Optional table reservation parameters can be passed in the TPB. Table reservation optionally describes an access method and lock resolution for a specified table that the transaction accesses. When table reservation is used, tables are reserved for the specified access when the transaction is started, rather than when the transaction actually accesses the table. Table reservation is only useful in an environment where simultaneous transactions share database access. It has three main purposes:

- Prevent possible deadlocks and update conflicts that can occur if locks are taken only when actually needed (the default behavior).
- Provide for *dependency locking*, the locking of tables that may be affected by triggers and integrity constraints. While explicit dependency locking is not required, it can assure that update conflict*s* do not occur because of indirect table conflicts.
- Change the level of shared access for one or more individual tables in a transaction. For example, an *isc_tpb_write* transaction with an isolation level of *isc_tpb_concurrency* may need exclusive update rights for a single table, and could use a reservation parameter to guarantee itself sole write access to the table.

Valid reservations are:

· *isc_tpb_shared*, *isc_tpb_lock_write*, which permits any transaction with an access mode of *isc_tpb_write* and isolation levels of *isc_tpb_concurrency* or *isc_tpb_read_committed*, to update, while other transactions with these isolation levels and an access mode of *isc_tpb_read* can read data.

· *isc_tpb_shared*, *isc_tpb_lock_read*, which permits any transaction to read data, and any transaction with an access mode of *isc_tpb_write* to update. This is the most liberal reservation mode.

· *isc_tpb_protected*, *isc_tpb_lock_write*, which prevents other transactions from updating. Other transactions with isolation levels of *isc_tpb_concurrency* or *isc_tpb_read_committed* can read data, but only this transaction can update.

· *isc_tpb_protected*, *isc_tpb_lock_read*, which prevents all transactions from updating, but permits all transactions to read data.

The name of the table to reserve must immediately follow the reservation parameters. For example, the following TPB declaration reserves a table, EMPLOYEE, for protected read access:

```
static char isc_tpb[] = {isc_tpb_version3,
   isc_tpb_write,
   isc_tpb_concurrency,
   isc_tpb_nowait,
   isc_tpb_protected, isc_tpb_lock_read, "EMPLOYEE"};
```

Several tables can be reserved at the same time. The following declaration illustrates how two tables are reserved, one for protected read, the other for protected write:

```
static char isc_tpb[] = {isc_tpb_version3,
   isc_tpb_write,
   isc_tpb_concurrency,
   isc_tpb_nowait,
   isc_tpb_protected, isc_tpb_lock_read, "COUNTRY",
   isc_tpb_protected, isc_tpb_lock_write, "EMPLOYEE"};
```

Using the default TPB

Providing a TPB for a transaction is optional. If one is not provided, then a NULL pointer must be passed to **isc_start_transaction()** in place of a pointer to the TPB. In this case, InterBase treats a transaction as if the following TPB had been declared for it:

```
static char isc_tpb[] = {isc_tpb_version3,
   isc_tpb_write,
   isc_tpb_concurrency,
   isc_tpb_wait};
```

Calling isc_start_transaction( )

Once transaction handles and TPBs are prepared, a transaction can be started by calling **isc_start_transaction()** using the following syntax:

```
ISC_STATUS isc_start_transaction(
ISC_STATUS *status vector,
isc_tr_handle *trans_handle,
short db_count,
isc_db_handle *&db_handle,
unsigned short tpb_length,
char *tpb_ad);
```

For a transaction that runs against a single database, set *db_count* to 1. *db_handle* should be a database handle set with a previous call to **isc_attach_database()**. *tpb_length* is the size of the TPB passed in the next parameter, and *tpb_ad* is the address of the TPB. The following code illustrates a typical call to **isc_start_transaction()**:

```
#include <ibase.h>
. . .
```

```
ISC_STATUS status_vector[20];
isc_db_handle db1;
isc_tr_handle tr1;
static char isc_tbp[] = {isc_tpb_version3,
   isc_tpb_write,
   isc_tpb_concurrency,
   isc_tpb_wait};
. . .
/* Initialize database and transaction handles here. */
db1 = 0L;
tr1 = 0L;
. . .
/* Code for attaching to database here is omitted. */
isc_start_transaction(status_vector,
   &tr1,
   1,
   &db1,
   (unsigned short) sizeof(isc_tpb),
   isc_tpb);
```

A transaction can be opened against multiple databases. To do so, set the *db_count* parameter to the number of databases against which the transaction runs, then for each database, repeat the *db_handle*, *tpb_length*, and *tpb_ad* parameters as a group once for each database. For example, the following code fragment assumes that two databases are connected when the transaction is started:

```
isc_start_transaction(status_vector,
&tr1,
   2,
   &db1,
   (unsigned short) sizeof(isc_tpb),
   &tpb);
   &db2,
   (unsigned short) sizeof(isc_tpb),
   &tpb);
```

For the complete syntax of **isc_start_transaction()**, see **page 327**.

Calling isc_start_multiple( )

An alternate method for starting a transaction against multiple databases is to use **isc_start_multiple()**. Using **isc_start_multiple()** is not recommended unless you:

- Are using a language that does not support a variable number of arguments in a function call.
- Do not know how many databases you want to attach to when coding the start of a transaction.

C programmers should seldom need to use this function.

isc_start_multiple() passes information about each target database to InterBase through an array of transaction existence blocks (TEBs). There must be one TEB for each database against which a transaction runs. A TEB is a structure you must declare in your applications as follows:

```
typdef struct {
   long *db_ptr;
   long tpb_len;
   char *tpb_ptr;
} ISC_TEB;
```

db_ptr is a pointer to a previously declared, initialized, and populated database handle. *tpb_len* is the size, in bytes, of the transaction parameter buffer (TPB) to use for the database, and *tpb_ptr* is a pointer to the TPB itself. For information about declaring, initializing, and populating a database handle, see **"Creating database handles" on page 40**. For more information about creating and populating a TPB, see **"Creating a transaction parameter buffer" on page 60**.

To use a TEB structure in an application, declare an array variable of type ISC_TEB. The number of array dimensions should correspond to the number of databases that the transaction runs against. For example, the following declaration creates an array of two TEBs, capable of supporting two databases:

```
ISC_TEB teb_array[2];
```

Once an array of TEBs is declared, and corresponding TBPs are created and populated for each database, values may be assigned to the appropriate fields in the TEBs. For example, the following code illustrates how two TEBs are filled:

```
. . .
ISC_STATUS status_vector[20];
isc_db_handle db1, db2;
isc_tr_handle trans;
ISC_TEB teb_array[2];
```

```
. . .
db1 = db2 = 0L;
trans = 0L;
/* Code assumes that two TPBs, isc_tpb1, and isc_tpb2, are created
here. */
/* Code assumes databases are attached here. */
/* assign values to TEB array */
teb_array[0].db_ptr = &db1;
teb_array[0].tpb_len = sizeof(isc_tpb1);
teb_array[0].tpb_ptr = isc_tpb1;
teb_array[1].db_ptr = &db2;
teb_array[1].tpb_len = sizeof(isc_tpb2);
teb_array[1].tpb_ptr = isc_tpb2;
. . .
```

After the TEBs are loaded with values, **isc_start_multiple()** can be called using the following syntax:

```
ISC_STATUS isc_start_multiple(
   ISC_STATUS *status_vector,
   isc_tr_handle *trans_handle,
   short db_handle_count,
   void *teb_vector_address);
```

For example, the following statements starts a two-database transaction:

```
. . .
ISC_STATUS status_vector[20];
isc_db_handle db1, db2;
isc_tr_handle trans;
ISC_TEB teb_array[2];
. . .
db1 = db2 = 0L;
trans = 0L;
/* Code assumes that two TPBs, isc_tpb1, and isc_tpb2, are created
here. */
/* Code assumes databases are attached here. */
/* assign values to TEB array */
teb_array[0].db_ptr = &db1;
teb_array[0].tpb_len = sizeof(isc_tpb1);
teb_array[0].tpb_ptr = isc_tpb1;
teb_array[1].db_ptr = &db2;
teb_array[1].tpb_len = sizeof(isc_tpb2);
teb_array[1].tpb_ptr = isc_tpb2;
```

```
/* Start the transaction */
isc_start_multiple(status_vector, &trans, 2, teb_array);
. . .
```

Ending transactions

When a transaction's tasks are complete, or an error prevents a transaction from completing, the transaction must be ended to set the database to a consistent state. There are two API functions that end transactions:

- **isc_commit_transaction()** makes a transaction's changes permanent in the database. For transactions that span databases, this function performs an automatic, two-phase commit to ensure that all changes are made successfully.
- **isc_rollback_transaction()** undoes a transaction's changes, returning the database to its previous state, before the transaction started. This function is typically used when one or more errors occur that prevent a transaction from completing successfully.

Both **isc_commit_transaction()** and **isc_rollback_transaction()** close the record streams associated with the transaction, reinitialize the transaction name to zero, and release system resources allocated for the transaction. Freed system resources are available for subsequent use by any application or program.

isc_rollback_transaction() is frequently used inside error-handling routines to clean up transactions when errors occur. It can also be used to roll back a partially completed transaction prior to retrying it, and it can be used to restore a database to its prior state if a program encounters an unrecoverable error.

The API offers three additional functions for controlling transactions:

- **isc_commit_retaining()** commits a transaction but retains the current transaction's context—the system resources and cursor states used in the transaction—without requiring the overhead of ending a transaction, starting a new one, and reestablishing cursor states. In a busy, multi-user environment, maintaining transaction context for each user speeds up processing and uses fewer system resources than closing a transaction and opening a new one.
- **isc_prepare_transaction()** and **isc_prepare_transaction2()** enable an application to perform the first phase of an automatic, two-phase commit in its own time, then issue a call to **isc_commit_transaction()** to complete the commit.

IMPORTANT If the program ends before a transaction ends, a transaction is automatically rolled back, but databases are not closed. If a program ends without closing the database, data loss or corruption is possible. Therefore, open databases should always be closed by issuing an explicit call to **isc_detach_database()**.

For more information about detaching from a database, see **Chapter 4, "Working with Databases."**

Using isc_commit_transaction( )

Use **isc_commit_transaction()** to write transaction changes permanently to a database. **isc_commit_transaction()** closes the record streams associated with the transaction, resets the transaction name to zero, and frees system resources assigned to the transaction for other uses. The complete syntax for **isc_commit_transaction()** is:

```
ISC_STATUS isc_commit_transaction(
ISC_STATUS *status_vector,
isc_tr_handle *trans_handle);
```

For example, the following call commits a transaction:

```
isc_commit_transaction(status_vector, &trans);
```

where *status_vector* is a pointer to a previously declared error status vector, and *trans* is a pointer to a previously declared and initialized transaction handle.

TIP Even transactions started with an access mode of *isc_tpb_read* should be ended with a call to **isc_commit_transaction()** rather than **isc_rollback_transaction()**. The database is not changed, but the overhead required to start subsequent transactions is greatly reduced.

Using isc_commit_retaining()

To write transaction changes to the database without establishing a new *transaction context*—the names, system resources, and current state of cursors used in a transaction—use **isc_commit_retaining()** instead of **isc_commit_transaction()**. In a busy, multi-user environment, maintaining the transaction context for each user speeds up processing and uses fewer system resources than closing and starting a new transaction for each action. The complete syntax for **isc_commit_retaining()** is:

```
ISC_STATUS isc_commit_retaining(
ISC_STATUS *status_vector,
isc_tr_handle *trans_handle);
```

isc_commit_retaining() writes all pending changes to the database, ends the current transaction *without* closing its record stream and cursors and without freeing its system resources, then starts a new transaction and assigns the existing record streams and system resources to the new transaction.

For example, the following call commits a specified transaction, preserving the current cursor status and system resources:

```
isc_commit_retaining(status_vector, &trans);
```

where *status_vector* is a pointer to a previously declared error status vector, and *trans* is a pointer to a previously declared and initialized transaction handle.

A call to **isc_rollback_transaction()** issued after **isc_commit_retaining()** only rolls back updates and writes occurring *after* the call to **isc_commit_retaining()**.

▶ *Using isc_prepare_transaction()*

When a transaction is committed against multiple databases using **isc_commit_transaction()**, InterBase automatically performs a two-phase commit. During the first phase of the commit, the InterBase engine polls all database participants to make sure they are still available, writes a message describing the transaction to the RDB$TRANSACTION_DESCRIPTION field of the RDB$TRANSACTION system table, then puts the transaction into a limbo state. It is during the second phase that transaction changes are actually committed to the database.

Some applications may have their own, additional requirements to make of the two-phase commit. These applications can call **isc_prepare_transaction()** to execute the first phase of the two-phase commit, then perform their own, additional tasks before completing the commit with a call to **isc_commit_transaction()**.

The syntax for **isc_prepare_transaction()** is:

```
ISC_STATUS isc_prepare_transaction(
ISC_STATUS *status_vector,
isc_tr_handle *trans_handle);
```

For example, the following code fragment illustrates how an application might call **isc_prepare_transaction()**, then its own routines, before completing a commit with **isc_commit_transaction()**:

```
ISC_STATUS status_vector[20];
isc_db_handle db1;
isc_tr_handle trans;
. . .
/* Initialize handles. */
db1 = 0L;
```

```
trans = 0L;
. . .
/* Code assumes a database is attached here, */
/* and a transaction started. */
. . .
/* Perform first phase of two-phase commit. */
isc_prepare_transaction(status_vector, &trans);
/* Application does its own processing here. */
my_app_function();
/* Now complete the two-phase commit. */
isc_commit_transaction(status_vector, &trans);
```

IMPORTANT It is generally a dangerous practice to delay the second phase of the commit after completing the first, because delays increase the chance that network or server problems can occur between phases.

Using isc_prepare_transaction2( )

Like **isc_prepare_transaction()**, **isc_prepare_transaction2()** performs the first phase of a two-phase commit, except that **isc_prepare_transaction2()** enables an application to supply its own transaction description for insertion into the RDB$TRANSACTION_DESCRIPTION field of the RDB$TRANSACTION system table.

IMPORTANT Do not use this call without first examining and understanding the information InterBase stores in RDB$TRANSACTION_DESCRIPTION during an automatic, two-phase commit. Storage of improper or incomplete information can prevent database recovery if the two-phase commit fails.

See **page 312** for the complete syntax of **isc_prepare_transaction2()**.

Using isc_rollback_transaction( )

Use **isc_rollback_transaction()** to restore the database to its condition prior to the start of the transaction. **isc_rollback_transaction()** also closes the record streams associated with the transaction, resets the transaction name to zero, and frees system resources assigned to the transaction for other uses. **isc_rollback_transaction()** typically appears in error-handling routines. The syntax for **isc_rollback_transaction()** is:

```
ISC_STATUS isc_rollback_transaction(
ISC_STATUS *status_vector,
isc_tr_handle *trans_handle);
```

For example, the following call rolls back a transaction:

```
isc_rollback_transaction(status_vector, &trans);
```

where *status_vector* is a pointer to a previously declared error status vector, and *trans* is a pointer to a previously declared and initialized transaction handle.

CHAPTER

6

Working with Dynamic SQL

This chapter describes how to use API dynamic SQL (DSQL) functions to handle dynamically created SQL statements for data definition and manipulation. Using low-level API calls enables client applications to build SQL statements or solicit them from end users at runtime, providing end users with a familiar database interface. It also provides applications developers low-level access to InterBase features, such as multiple databases, not normally available at a higher level with embedded DSQL statements. For example, the InterBase **isql** utility is a DSQL application built on low-level API calls.

All API DSQL function names begin with "isc_dsql" to make it easier to distinguish them from other API calls.

Overview of the DSQL programming process

Building and executing DSQL applications with the API involve the following general steps:

- Embedding DSQL API functions in an application.
- Using host-language facilities, such as datatypes and macros, to provide input and output areas for passing statements and parameters at runtime.
- Programming methods that use these statements and facilities to process SQL statements at runtime.

These steps are described in detail throughout this chapter.

DSQL API limitations

Although DSQL offers many advantages, it also has the following limitations:

- Dynamic transaction processing is not permitted; all named transactions must be declared at compile time.
- Dynamic access to Blob and array data is not supported; Blob and array data can be accessed, but only through standard, statically processed SQL statements, or through low-level API calls.
- Database creation is restricted to CREATE DATABASE statements executed within the context of EXECUTE IMMEDIATE.

For more information about database access in DSQL, see **"Accessing databases" on page 78**. For more information about handling transactions in DSQL applications, see **"Handling transactions" on page 79**. For more information about working with Blob data in DSQL, see **"Processing Blob data" on page 81**. For more information about handling array data in DSQL, see **"Processing array data" on page 81**. For more information about dynamic creation of databases, see **"Creating a database" on page 80**.

Accessing databases

The InterBase API permits applications to attach to multiple databases simultaneously using database handles. Database handles must be declared and initialized when an application is compiled. Separate database handles should be supplied and initialized for each database accessed simultaneously. For example, the following code creates a single handle, *db1*, and initializes it to zero:

```
#include <ibase.h>
isc_db_handle db1;
. . .
db1 = 0L;
```

Once declared and initialized, a database handle can be assigned dynamically to a database at runtime as follows:

```
#include <ibase.h>
. . .
char dbname[129];
```

```
ISC_STATUS status_vector[20];
. . .
prompt_user("Name of database to open: ");
gets(dbname);
isc_attach_database(status_vector, 0, dbname, &db1, NULL, NULL);
```

A database handle can be used to attach to different databases as long as a previously attached database is first detached with **isc_detach_database()**, which automatically sets database handles to NULL. The following statements detach from a database, set the database handle to zero, and attach to a new database:

```
isc_detach_database(status_vector, &db1);
isc_attach_database(status_vector, 0, "employee.gdb", &db1, NULL,
NULL);
```

For more information about API function calls for databases, see **Chapter 4, "Working with Databases."**

Handling transactions

InterBase requires that all transaction handles be declared when an application is compiled. Once fixed at compile time, transaction handles cannot be changed at runtime, nor can new handles be declared dynamically at runtime. Most API functions that process SQL statements at runtime, such as **isc_dsql_describe()**, **isc_dsql_describe_bind()**, **isc_dsql_execute()**, **isc_dsql_execute2()**, **isc_dsql_execute_immediate()**, **isc_dsql_exec_immed2()**, and **isc_dsql_prepare()**, support the inclusion of a transaction handle parameter. The SQL statements processed by these functions cannot pass transaction handles even if the SQL syntax for the statement permits the use of a TRANSACTION clause.

Before a transaction handle can be used, it must be declared and initialized to zero. The following code declares, initializes, and uses a transaction handle in an API call that allocates and prepares an SQL statement for execution:

```
#include <ibase.h>
. . .
isc_tr_handle trans; /* Declare a transaction handle. */
isc_stmt_handle stmt; /* Declare a statement handle. */
char *sql_stmt = "SELECT * FROM EMPLOYEE";
isc_db_handle db1;
ISC_STATUS status_vector[20];
. . .
trans = 0L; /* Initialize the transaction handle to zero. */
stmt = NULL; /* Set handle to NULL before allocation. */
```

```
/* This code assumes both that a database attachment is made, */
/* and a transaction is started here. */
. . .
/* Allocate the SQL statement handle. */
isc_dsql_allocate_statement(status_vector, &db1, &stmt);
/* Prepare the statement for execution. */
isc_dsql_prepare(status_vector, &trans, &stmt, 0, sql_stmt, 1, NULL);
```

Note The SQL SET TRANSACTION statement cannot be prepared with **isc_dsql_prepare()**, but it can be processed with **isc_dsql_execute_immediate()** if:

1. Previous transactions are first committed or rolled back.
2. The transaction handle is set to NULL.

For more information about using SQL statements, see the *Programmer's Guide*. For more information about SQL statement syntax, see the *Language Reference*.

Creating a database

To create a new database in an API application:

1. Detach from any currently attached databases with **isc_detach_database()**. Detaching from a database automatically sets its database handle to NULL.
2. Build the CREATE DATABASE statement to process.
3. Execute the statement with **isc_dsql_execute_immediate()** or **isc_dsql_exec_immed2()**.

For example, the following statements disconnect from any currently attached databases, and create a new database. Any existing database handles are set to NULL, so that they can be used to connect to the new database in future DSQL statements.

```
char *str = "CREATE DATABASE \"new_emp.gdb\"";
. . .
isc_detach_database(status_vector, &db1);
isc_dsql_execute_immediate(status_vector, &db1, &trans, 0, str, 1,
   NULL);
```

Processing Blob data

Blob processing is not directly supported using DSQL, nor are Blob cursors supported. Applications that process SQL statements can use API calls to handle Blob processing. For more information about processing Blob data, see **Chapter 7, "Working with Blob Data."**

Processing array data

Array processing is not directly supported using DSQL. DSQL applications can use API calls to process array data. For more information about array calls, see **Chapter 8, "Working with Array Data."**

Writing an API application to process SQL statements

Writing an API application that processes SQL statements enables a developer to code directly to InterBase at a low level, while presenting end users a familiar SQL interface. API SQL applications are especially useful when any of the following are not known until runtime:

- The text of the SQL statement
- The number of host variables
- The datatypes of host variables
- References to database objects

Writing an API DSQL application is more complex than programming embedded SQL applications with regular SQL because for most DSQL operations, the application needs explicitly to allocate and process an extended SQL descriptor area (XSQLDA) data structure to pass data to and from the database.

To use the API to process a DSQL statement, follow these basic steps:

1. Determine if API calls can process the SQL statement.
2. Represent the SQL statement as a character string in the application.
3. If necessary, allocate one or more XSQLDAs for input parameters and return values.
4. Use appropriate API programming methods to process the SQL statement.

Determining if API calls can process an SQL statement

Except as noted earlier in this chapter, DSQL functions can process most SQL statements. For example, DSQL can process data manipulation statements such as DELETE and INSERT, data definition statements such as ALTER TABLE and CREATE INDEX, and SELECT statements.

The following table lists SQL statements that cannot be processed by DSQL functions:

Statement	Statement
CLOSE	DECLARE CURSOR
DESCRIBE	EXECUTE
EXECUTE IMMEDIATE	FETCH
OPEN	PREPARE

TABLE 6.1 SQL statements that cannot be processed by the API

These statements are used to process DSQL requests or to handle SQL cursors, which must always be specified when an application is written. Attempting to use them with DSQL results in run-time errors.

Representing an SQL statement as a character string

Within a DSQL application, an SQL statement can come from different sources. It might come directly from a user who enters a statement at a prompt, as does **isql**. Or it might be generated by the application in response to user interaction. Whatever the source of the SQL statement, it must be represented as an *SQL statement string*, a character string that is passed to DSQL for processing.

SQL statement strings do not begin with the EXEC SQL prefix or end with a semicolon (;) as they do in typical embedded applications. For example, the following host-language variable declaration is a valid SQL statement string:

```
char *str = "DELETE FROM CUSTOMER WHERE CUST_NO = 256";
```

Note The semicolon that appears at the end of this char declaration is a C terminator, and not part of the SQL statement string.

Specifying parameters in SQL statement strings

SQL statement strings often include *value parameters*, expressions that evaluate to a single numeric or character value. Parameters can be used anywhere in statement strings where SQL expects a value that is not the name of a database object.

A value parameter in a statement string can be passed as a constant, or passed as a placeholder at runtime. For example, the following statement string passes 256 as a constant:

```
char *str = "DELETE FROM CUSTOMER WHERE CUST_NO = 256";
```

It is also possible to build strings at runtime from a combination of constants. This method is useful for statements where the variable is not a true constant, or it is a table or column name, and where the statement is executed only once in the application.

To pass a parameter as a placeholder, the value is passed as a question mark (?) embedded within the statement string:

```
char *str = "DELETE FROM CUSTOMER WHERE CUST_NO = ?";
```

When a DSQL function processes a statement containing a placeholder, it replaces the question mark with a value supplied in an extended SQL descriptor area (XSQLDA) previously declared and populated in the application. Use placeholders in statements that are prepared once, but executed many times with different parameter values.

Replaceable value parameters are often used to supply values in SQL SELECT statement WHERE clause comparisons and in the UPDATE statement SET clause.

Understanding the XSQLDA

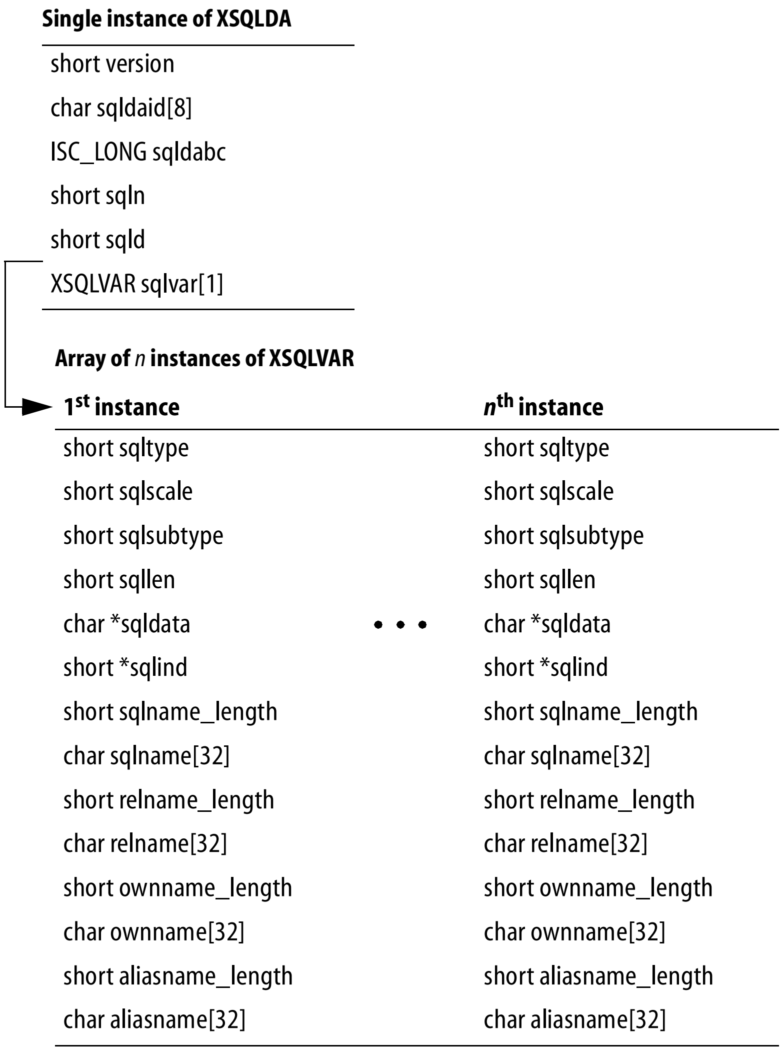
All DSQL applications must declare one or more extended SQL descriptor areas (XSQLDAs). The XSQLDA structure definition can be found in the *ibase.h* header file in the InterBase *include* directory. Applications declare instances of the XSQLDA for use.

The XSQLDA is a host-language data structure that DSQL uses to transport data to or from a database when processing an SQL statement string. There are two types of XSQLDAs: *input* descriptors and *output* descriptors. Both input and output descriptors are implemented using the XSQLDA structure.

One field in the XSQLDA, *sqlvar*, is an XSQLVAR structure. The *sqlvar* is especially important, because one XSQLVAR must be defined for each input parameter or column returned. Like the XSQLDA, the XSQLVAR is a structure defined in *ibase.h* in the InterBase *include* directory.

Applications do not declare instances of the XSQLVAR ahead of time, but must, instead, dynamically allocate storage for the proper number of XSQLVAR structures required for each DSQL statement before it is executed, then deallocate it, as appropriate, after statement execution.

The following figure illustrates the relationship between the XSQLDA and the XSQLVAR:



Single instance of XSQLDA

short version
char sqldaid[8]
ISC_LONG sqldabc
short sqln
short sqld
XSQLVAR sqlvar[1]

Array of *n* instances of XSQLVAR

1st instance		*n*th instance
short sqltype		short sqltype
short sqlscale		short sqlscale
short sqlsubtype		short sqlsubtype
short sqllen		short sqllen
char *sqldata	• • •	char *sqldata
short *sqlind		short *sqlind
short sqlname_length		short sqlname_length
char sqlname[32]		char sqlname[32]
short relname_length		short relname_length
char relname[32]		char relname[32]
short ownname_length		short ownname_length
char ownname[32]		char ownname[32]
short aliasname_length		short aliasname_length
char aliasname[32]		char aliasname[32]

An input XSQLDA consists of a single XSQLDA structure and one XSQLVAR structure for each input parameter. An output XSQLDA also consists of one XSQLDA structure and one XSQLVAR structure for each data item returned by the statement. An XSQLDA and its associated XSQLVAR structures are allocated as a single block of contiguous memory.

The **isc_dsql_prepare()**, **isc_dsql_describe()**, and **isc_dsql_describe_bind()** functions can be used to determine the proper number of XSQLVAR structures to allocate, and the XSQLDA_LENGTH macro can be used to allocate the proper amount of space. For more information about the XSQLDA_LENGTH macro, see **"Using the XSQLDA_LENGTH macro" on page 88**.

XSQLDA field descriptions

The following table describes the fields that comprise the XSQLDA structure:

Field definition	Description
short version	Indicates the version of the XSQLDA structure. Set by an application. The current version is defined in *ibase.h* as SQLDA_VERSION1
char sqldaid[8]	Reserved for future use
ISC_LONG *sqldabc*	Reserved for future use
short sqln	Indicates the number of elements in the *sqlvar* array; the application should set this field whenever it allocates storage for a descriptor
short sqld	Indicates the number of parameters for an input XSQLDA, or the number of select-list items for an output XSQLDA; set by InterBase during an **isc_dsql_describe()**, **isc_dsql_describe_bind()**, or **isc_dsql_prepare()** For an input descriptor, an *sqld* of 0 indicates that the SQL statement has no parameters; for an output descriptor, an *sqld* of 0 indicates that the SQL statement is not a SELECT statement
XSQLVAR *sqlvar*	The array of XSQLVAR structures; the number of elements in the array is specified in the *sqln* field

TABLE 6.2 XSQLDA field descriptions

The following table describes the fields that comprise the XSQLVAR structure:

Field definition	Description
short sqltype	Indicates the SQL datatype of parameters or select-list items; set by InterBase during **isc_dsql_describe()**, **isc_dsql_describe_bind()**, or **isc_dsql_prepare()**
short sqlscale	Provides scale, specified as a negative number, for exact numeric datatypes (DECIMAL, NUMERIC); set by InterBase during **isc_dsql_describe()**, **isc_dsql_describe_bind()**, or **isc_dsql_prepare()**
short sqlsubtype	Specifies the subtype for Blob data; set by InterBase during **isc_dsql_describe()**, **isc_dsql_describe_bind()**, or **isc_dsql_prepare()**
short sqllen	Indicates the maximum size, in bytes, of data in the *sqldata* field; set by InterBase during **isc_dsql_describe()**, **isc_dsql_describe_bind()**, or **isc_dsql_prepare()**
char *sqldata	For input descriptors, specifies either the address of a select-list item or a parameter; set by the application For output descriptors, contains a value for a select-list item; set by InterBase
short *sqlind	On input, specifies the address of an indicator variable; set by an application; on output, specifies the address of column indicator value for a select-list item following a FETCH A value of 0 indicates that the column is not NULL; a value of –1 indicates the column is NULL; set by InterBase
short sqlname_length	Specifies the length, in bytes, of the data in field, *sqlname*; set by InterBase during **isc_dsql_prepare()** or **isc_dsql_describe()**
char sqlname[32]	Contains the name of the column. Not NULL (\0) terminated; set by InterBase during **isc_dsql_prepare()** or **isc_dsql_describe()**
short relname_length	Specifies the length, in bytes, of the data in field, *relname*; set by InterBase during **isc_dsql_prepare()** or **isc_dsql_describe()**

TABLE 6.3 XSQLVAR field descriptions

Field definition	Description
char relname[32]	Contains the name of the table; not NULL (\0) terminated, set by InterBase during **isc_dsql_prepare()** or **isc_dsql_describe()**
short ownname_length	Specifies the length, in bytes, of the data in field, *ownname*; set by InterBase during **isc_dsql_prepare()** or **isc_dsql_describe()**
char ownname[32]	Contains the name of the table owner; not NULL (\0) terminated, set by InterBase during **isc_dsql_prepare()** or **isc_dsql_describe()**
short aliasname_length	Specifies the length, in bytes, of the data in field, *aliasname*; set by InterBase during **isc_dsql_prepare()** or **isc_dsql_describe()**
char aliasname[32]	Contains the alias name of the column. If no alias exists, contains the column name; not NULL (\0) terminated, set by InterBase during **isc_dsql_prepare()** or **isc_dsql_describe()**

TABLE 6.3 XSQLVAR field descriptions (*continued*)

Input descriptors

Input descriptors are used to process SQL statement strings that contain parameters. Before an application can execute a statement with parameters, it must supply values for them. The application indicates the number of parameters passed in the XSQLDA *sqld* field, then describes each parameter in a separate XSQLVAR structure. For example, the following statement string contains two parameters, so an application must set *sqld* to 2, and describe each parameter:

```
char *str = "UPDATE DEPARTMENT SET BUDGET = ? WHERE LOCATION = ?";
```

When the statement is executed, the first XSQLVAR supplies information about the BUDGET value, and the second XSQLVAR supplies the LOCATION value.

For more information about using input descriptors, see **"DSQL programming methods" on page 94**.

Output descriptors

Output descriptors return values from an executed query to an application. The *sqld* field of the XSQLDA indicates how many values were returned. Each value is stored in a separate XSQLVAR structure. The XSQLDA *sqlvar* field points to the first of these XSQLVAR structures. The following statement string requires an output descriptor:

```
char *str = "SELECT * FROM CUSTOMER WHERE CUST_NO > 100";
```

For information about retrieving information from an output descriptor, see **"DSQL programming methods" on page 94**.

Using the XSQLDA_LENGTH macro

The *ibase.h* header file defines a macro, XSQLDA_LENGTH, to calculate the number of bytes that must be allocated for an input or output XSQLDA. XSQLDA_LENGTH is defined as follows:

```
#define XSQLDA_LENGTH (n) (sizeof (XSQLDA) + (n – 1) * sizeof(XSQLVAR))
```

n is the number of parameters in a statement string, or the number of select-list items returned from a query. For example, the following C statement uses the XSQLDA_LENGTH macro to specify how much memory to allocate for an XSQLDA with 5 parameters or return items:

```
XSQLDA *my_xsqlda;
. . .
my_xsqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(5));
. . .
```

For more information about using the XSQLDA_LENGTH macro, see **"DSQL programming methods" on page 94**.

SQL datatype macro constants

InterBase defines a set of macro constants to represent SQL datatypes and NULL status information in an XSQLVAR. An application should use these macro constants to specify the datatype of parameters and to determine the datatypes of select-list items in an SQL statement. The following table lists each SQL datatype, its corresponding macro constant expression, C datatype or InterBase typedef, and whether or not the *sqlind* field is used to indicate a parameter or variable that contains NULL or unknown data:

SQL datatype	Macro expression	C datatype or typedef	*sqlind* used?
Array	SQL_ARRAY	ISC_QUAD	No
Array	SQL_ARRAY + 1	ISC_QUAD	Yes
Blob	SQL_BLOB	ISC_QUAD	No
BLOB	SQL_BLOB + 1	ISC_QUAD	Yes
CHAR	SQL_TEXT	char[]	No
CHAR	SQL_TEXT + 1	char[]	Yes
DATE	SQL_DATE	ISC_QUAD	No
DATE	SQL_DATE + 1	ISC_QUAD	Yes
DECIMAL	SQL_SHORT, SQL_LONG, or SQL_DOUBLE	int, long, or double	No
DECIMAL	SQL_SHORT + 1, SQL_LONG + 1, or SQL_DOUBLE + 1	int, long, or double	Yes
DOUBLE PRECISON	SQL_DOUBLE	double	No
DOUBLE PRECISION	SQL_DOUBLE + 1	double	Yes
INTEGER	SQL_LONG	long	No
INTEGER	SQL_LONG + 1	ISC_LONG	Yes
FLOAT	SQL_FLOAT	float	No
FLOAT	SQL_FLOAT + 1	float	Yes

TABLE 6.4 SQL datatypes, macro expressions, and C datatypes

SQL datatype	Macro expression	C datatype or typedef	*sqlind* used?
NUMERIC	SQL_SHORT, SQL_LONG, or SQL_DOUBLE	int, long, or double	No
NUMERIC	SQL_SHORT + 1, SQL_LONG + 1, or SQL_DOUBLE + 1	int, long, or double	Yes
SMALLINT	SQL_SHORT	short	No
SMALLINT	SQL_SHORT + 1	short	Yes
VARCHAR	SQL_VARYING	First 2 bytes: short containing the length of the character string; remaining bytes: char[]	No
VARCHAR	SQL_VARYING + 1	First 2 bytes: short containing the length of the character string; remaining bytes: char[]	Yes

TABLE 6.4 SQL datatypes, macro expressions, and C datatypes (*continued*)

Note DECIMAL and NUMERIC datatypes are stored internally as SMALLINT, INTEGER, or DOUBLE PRECISION datatypes. To specify the correct macro expression to provide for a DECIMAL or NUMERIC column, use **isql** to examine the column definition in the table to see how InterBase is storing column data, then choose a corresponding macro expression.

The datatype information for a parameter or select-list item is contained in the *sqltype* field of the XSQLVAR structure. The value contained in *sqltype* provides two pieces of information:

- The datatype of the parameter or select-list item.
- Whether *sqlind* is used to indicate NULL values. If *sqlind* is used, its value specifies whether the parameter or select-list item is NULL (–1), or not NULL (0).

For example, if *sqltype* equals SQL_TEXT, the parameter or select-list item is a CHAR that does not use *sqlind* to check for a NULL value (because, in theory, NULL values are not allowed for it). If *sqltype* equals SQL_TEXT + 1, then *sqlind* can be checked to see if the parameter or select-list item is NULL.

TIP The C language expression, *sqltype & 1,* provides a useful test of whether a parameter or select-list item can contain a NULL. The expression evaluates to 0 if the parameter or select-list item cannot contain a NULL, and 1 if the parameter or select-list item can contain a NULL. The following code fragment demonstrates how to use the expression:

```
if (sqltype & 1 == 0)
{
 /* parameter or select-list item that CANNOT contain a NULL */
}
else
{
 /* parameter or select-list item CAN contain a NULL */
}
```

By default, both **isc_dsql_prepare()** and **isc_dsql_describe()** return a macro expression of type + 1, so *sqlind* should always be examined for NULL values with these statements.

Handling varying string datatypes

VARCHAR, CHARACTER VARYING, and NCHAR VARYING datatypes require careful handling in DSQL. The first two bytes of these datatypes contain string length information, while the remainder of the data contains the actual bytes of string data to process.

To avoid having to write code to extract and process variable-length strings in an application, it is possible to force these datatypes to fixed length using SQL macro expressions. For more information about forcing variable-length data to fixed length for processing, see **"Coercing datatypes" on page 92**.

Applications can, instead, detect and process variable-length data directly. To do so, they must extract the first two bytes from the string to determine the byte-length of the string itself, then read the string, byte-by-byte, into a null-terminated buffer.

Handling NUMERIC and DECIMAL datatypes

DECIMAL and NUMERIC datatypes are stored internally as SMALLINT, INTEGER, or DOUBLE PRECISION datatypes, depending on the precision and scale defined for a column definition that uses these types. To determine how a DECIMAL or NUMERIC value is actually stored in the database, use **isql** to examine the column definition in the table. If NUMERIC is reported, then data is actually being stored as DOUBLE PRECISION.

When a DECIMAL or NUMERIC value is stored as a SMALLINT or INTEGER, the value is stored as a whole number. During retrieval in DSQL, the *sqlscale* field of the XSQLVAR is set to a negative number that indicates the factor of 10 by which the whole number (returned in *sqldata*), must be divided in order to produce the correct NUMERIC or DECIMAL value with its fractional part. If *sqlcale* is –1, then the number must be divided by 10, if it is –2, then the number must be divided by 100,
–3 by 1000, and so forth.

Coercing datatypes

Sometimes when processing DSQL input parameters and select-list items, it is desirable or necessary to translate one datatype to another. This process is referred to as *datatype coercion*. For example, datatype coercion is often used when parameters or select-list items are of type VARCHAR. The first two bytes of VARCHAR data contain string length information, while the remainder of the data is the string to process. By coercing the data from SQL_VARYING to SQL_TEXT, data processing can be simplified.

Coercion can only be from one compatible datatype to another. For example, SQL_VARYING to SQL_TEXT, or SQL_SHORT to SQL_LONG.

▶ *Coercing character datatypes*

To coerce SQL_VARYING datatypes to SQL_TEXT datatypes, change the *sqltype* field in the parameter's or select-list item's XSQLVAR structure to the desired SQL macro datatype constant. For example, the following statement assumes that *var* is a pointer to an XSQLVAR structure, and that it contains an SQL_VARYING datatype to convert to SQL_TEXT:

```
var->sqltype = SQL_TEXT;
```

After coercing a character datatype, provide proper storage space for it. The XSQLVAR field, *sqllen,* contains information about the size of the uncoerced data.
Set the XSQLVAR *sqldata* field to the address of the data.

▶ *Coercing numeric datatypes*

To coerce one numeric datatype to another, change the *sqltype* field in the parameter's or select-list item's XSQLVAR structure to the desired SQL macro datatype constant. For example, the following statement assumes that *var* is a pointer to an XSQLVAR structure, and that it contains an SQL_SHORT datatype to convert to SQL_LONG:

```
var->sqltype = SQL_LONG;
```

IMPORTANT Do not coerce a larger datatype to a smaller one. Data can be lost in such a translation.

▶ *Setting a* NULL *indicator*

If a parameter or select-list item contains a NULL value, the *sqlind* field should be used to indicate its NULL status. Appropriate storage space must be allocated for *sqlind* before values can be stored there.

Before insertion, set *sqlind* to –1 to indicate that NULL values are legal. Otherwise, set *sqlind* to 0.

After selection, an *sqlind* of –1 indicates a field contains a NULL value. Other values indicate a field contains non-NULL data.

Aligning numerical data

Ordinarily, when a variable with a numeric datatype is created, the compiler will ensure that the variable is stored at a properly aligned address, but when numeric data is stored in a dynamically allocated buffer space, such as can be pointed to by the XSQLDA and XSQLVAR structures, the programmer must take precautions to ensure that the storage space is properly aligned.

Certain platforms, in particular those with RISC processors, require that numerical data in dynamically allocated storage structures be aligned properly in memory. Alignment is dependent both on datatype and platform.

For example, a short integer on a Sun SPARCstation must be located at an address divisible by 2, while a long on the same platform must be located at an address divisible by 4. In most cases, a data item is properly aligned if the address of its starting byte is divisible by the correct alignment number. Consult specific system and compiler documentation for alignment requirements.

A useful rule of thumb is that the size of a datatype is always a valid alignment number for the datatype. For a given type T, if size of (T) equals *n*, then addresses divisible by *n* are correctly aligned for T. The following macro expression can be used to align data:

```
#define ALIGN(ptr, n) ((ptr + n - 1) & ~(n - 1))
```

where *ptr* is a pointer to char.

The following code illustrates how the ALIGN macro might be used:

```
char *buffer_pointer, *next_aligned;
next_aligned = ALIGN(buffer_pointer, sizeof(T));
```

DSQL programming methods

There are four possible DSQL programming methods for handling an SQL statement string. The best method for processing a string depends on the type of SQL statement in the string, and whether or not it contains placeholders for parameters. The following decision table explains how to determine the appropriate processing method for a given string:

Is it a query?	Does it have placeholders?	Processing method to use:
No	No	Method 1
No	Yes	Method 2
Yes	No	Method 3
Yes	Yes	Method 4

TABLE 6.5 SQL statement strings and recommended processing methods

Method 1: Non-query statements without parameters

There are two ways to process an SQL statement string containing a non-query statement without placeholder parameters:

- Use **isc_dsql_execute_immediate()** to prepare and execute the string a single time.
- Use **isc_dsql_allocate_statement()** to allocate a statement string for the statement to execute, **isc_dsql_prepare()** to parse the statement for execution and assign it a name, then use **isc_dsql_execute()** to carry out the statement's actions as many times as required in an application.

Using isc_dsql_execute_immediate( )

1. To execute a statement string a single time, use **isc_dsql_execute_immediate()**:
2. Elicit a statement string from the user or create one that contains the SQL statement to be processed. For example, the following statement creates an SQL statement string:

   ```
   char *str = "UPDATE DEPARTMENT SET BUDGET = BUDGET * 1.05";
   ```

3. Parse and execute the statement string using **isc_dsql_execute_immediate()**:

```
isc_dsql_execute_immediate(status_vector, &db1, &trans, 0, str, 1,
NULL);
```

Note **isc_dsql_execute_immediate()** also accepts string literals. For example,

```
isc_dsql_execute_immediate(status_vector, &db1, &trans, 0,
"UPDATE DEPARTMENT SET BUDGET = BUDGET * 1.05", 1, NULL);
```

For the complete syntax of **isc_dsq_execute_immediate()** and an explanation of its parameters, see **Chapter 12, "API Function Reference."**

Using isc_dsql_prepare( ) and isc_dsql_execute( )

To execute a statement string several times, use **isc_dsql_allocate_statement()**, **isc_dsql_prepare()**, and **isc_dsql_execute()**:

1. Elicit a statement string from the user or create one that contains the SQL statement to be processed. For example, the following statement creates an SQL statement string:

   ```
   char *str = "UPDATE DEPARTMENT SET BUDGET = BUDGET * 1.05";
   ```

2. Declare and initialize an SQL statement handle, then allocate it with **isc_dsql_allocate_statement()**:

   ```
   isc_stmt_handle stmt; /* Declare a statement handle. */
   stmt = NULL; /* Set handle to NULL before allocation. */
   . . .
   isc_dsql_allocate_statement(status_vector, &db1, &stmt);
   ```

3. Parse the statement string with **isc_dsql_prepare()**. This sets the statement handle (*stmt*) to refer to the parsed format. The statement handle is used in subsequent calls to **isc_dsql_execute()**:

   ```
   isc_dsql_prepare(status_vector, &trans, &stmt, 0, str, 1, NULL);
   ```

Note **isc_dsql_prepare()** also accepts string literals. For example,

```
isc_dsql_prepare(status_vector, &trans, &stmt, 0,
"UPDATE DEPARTMENT SET BUDGET = BUDGET * 1.05", 1, NULL);
```

4. Execute the named statement string using **isc_dsql_execute()**. For example, the following statement executes a statement string named *stmt*:

   ```
   isc_dsql_execute(status_vector, &trans, &stmt, 1, NULL);
   ```

   Once a statement string is prepared, it can be executed as many times as required in an application.

Method 2: Non-query statements with parameters

There are two steps to processing an SQL statement string containing a non-query statement with placeholder parameters:

1. Create an input XSQLDA to process a statement string's parameters.
2. Prepare and execute the statement string with its parameters.

Creating the input XSQLDA

Placeholder parameters are replaced with actual data before a prepared SQL statement string is executed. Because those parameters are unknown when the statement string is created, an input XSQLDA must be created to supply parameter values at execute time. To prepare the XSQLDA, follow these steps:

1. Declare a variable to hold the XSQLDA needed to process parameters. For example, the following declaration creates an XSQLDA called *in_sqlda*:

   ```
   XSQLDA *in_sqlda;
   ```
2. Optionally declare a variable for accessing the XSQLVAR structure of the XSQLDA:

   ```
   XSQLVAR *var;
   ```

   Declaring a pointer to the XSQLVAR structure is not necessary, but can simplify referencing the structure in subsequent statements.
3. Allocate memory for the XSQLDA using the XSQLDA_LENGTH macro. The following statement allocates storage for *in_sqlda*:

   ```
   in_sqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(10));
   ```

   In this statement space for 10 XSQLVAR structures is allocated, allowing the XSQLDA to accommodate up to 10 parameters.
4. Set the *version* field of the XSQLDA to SQLDA_VERSION1, and set the *sqln* field to indicate the number of XSQLVAR structures allocated:

   ```
   in_sqlda->version = SQLDA_VERSION1;
   in_sqlda->sqln = 10;
   ```

Preparing and executing a statement string with parameters

After an XSQLDA is created for holding a statement string's parameters, the statement string can be created and prepared. Local variables corresponding to the placeholder parameters in the string must be assigned to their corresponding *sqldata* fields in the XSQLVAR structures.

To prepare and execute a non-query statement string with parameters, follow these steps:

1. Elicit a statement string from the user or create one that contains the SQL statement to be processed. For example, the following statement creates an SQL statement string with placeholder parameters:

   ```
   char *str = "UPDATE DEPARTMENT SET BUDGET = ?, LOCATION = ?";
   ```

   This statement string contains two parameters: a value to be assigned to the BUDGET column and a value to be assigned to the LOCATION column.

2. Declare and initialize an SQL statement handle, then allocate it with **isc_dsql_allocate()**:

   ```
   isc_stmt_handle stmt; /* Declare a statement handle. */
   stmt = NULL; /* Set handle to NULL before allocation. */
   . . .
   isc_dsql_allocate_statement(status_vector, &db1, &stmt);
   ```

3. Parse the statement string with **isc_dsql_prepare()**. This sets the statement handle (*stmt*) to refer to the parsed format. The statement handle is used in subsequent calls to **isc_dsql_describe_bind()** and **isc_dsql_execute()**:

```
    isc_dsql_prepare(status_vector, &trans, &stmt, 0, str, 1,
in_sqlda);
```

4. Use **isc_dsql_describe_bind()** to fill the input XSQLDA with information about the parameters contained in the SQL statement:

   ```
   isc_dsql_describe_bind(status_vector, &stmt, 1, in_sqlda);
   ```

5. Compare the value of the *sqln* field of the XSQLDA to the value of the *sqld* field to make sure enough XSQLVARs are allocated to hold information about each parameter. *sqln* should be at least as large as *sqld*. If not, free the storage previously allocated to the input descriptor, reallocate storage to reflect the number of parameters specified by *sqld*, reset *sqln* and *version*, then execute **isc_dsql_describe_bind()** again:

   ```
   if (in_sqlda->sqld > in_sqlda->sqln)
   {
       n = in_sqlda->sqld;
       free(in_sqlda);
       in_sqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(n));
       in_sqlda->sqln = n
       in_sqlda->version = SQLDA_VERSION1;
       isc_dsql_describe_bind(status_vector, &stmt, 1, in_sqlda);
   }
   ```

6. Process each XSQLVAR parameter structure in the XSQLDA. Processing a parameter structure involves up to four steps:

   - Coerce a parameter's datatype (optional).
   - Allocate local storage for the data pointed to by the *sqldata* field of the XSQLVAR. This step is only required if space for local variables is not allocated until runtime. The following example illustrates dynamic allocation of local variable storage space.
   - Provide a value for the parameter consistent with its datatype (required).
   - Provide a NULL value indicator for the parameter.

   The following code example illustrates these steps, looping through each XSQLVAR structure in the *in_sqlda* XSQLDA:

```
for (i=0, var = in_sqlda->sqlvar; i < in_sqlda->sqld; i++, var++)
{
    /* Process each XSQLVAR parameter structure here.
    Var points to the parameter structure. */

    dtype = (var->sqltype & ~1) /* drop NULL flag for now */
    switch(dtype)
    {
        case SQL_VARYING: /* coerce to SQL_TEXT */
            var->sqltype = SQL_TEXT;
            /* allocate local variable storage */
            var->sqldata = (char *)malloc(sizeof(char)*var->sqllen);
            . . .
            break;
        case SQL_TEXT:
            var->sqldata = (char *)malloc(sizeof(char)*var->sqllen);
            /* provide a value for the parameter */
            . . .
            break;
        case SQL_LONG:
            var->sqldata = (char *)malloc(sizeof(long));
            /* provide a value for the parameter */
            *(long *)(var->sqldata) = 17;
            break;
        . . .
    }  /* end of switch statement */
    if (sqltype & 1)
    {
        /* allocate variable to hold NULL status */
        var->sqlind = (short *)malloc(sizeof(short));
    }
}       /* end of for loop */
```

For more information about datatype coercion and NULL indicators, see **"Coercing datatypes" on page 92**.

7. Execute the named statement string with **isc_dsql_execute()**. For example, the following statement executes a statement string named *stmt*:

```
isc_dsql_execute(status_vector, &trans, &stmt, 1, in_sqlda);
```

Re-executing the statement string

Once a non-query statement string with parameters is prepared, it can be executed as often as required in an application. Before each subsequent execution, the input XSQLDA can be supplied with new parameter and NULL indicator data.

To supply new parameter and NULL indicator data for a prepared statement, repeat step 6 of **"Preparing and executing a statement string with parameters" on page 96**.

Method 3: Query statements without parameters

There are three steps to processing an SQL query statement string without parameters:

1. Prepare an output XSQLDA to process the select-list items returned when the query is executed.
2. Prepare the statement string.
3. Use a cursor to execute the statement and retrieve select-list items from the output XSQLDA.

Preparing the output XSQLDA

Most queries return one or more rows of data, referred to as a *select-list*. Because the number and kind of items returned are unknown when a statement string is created, an output XSQLDA must be created to store select-list items that are returned at runtime. To prepare the XSQLDA, follow these steps:

1. Declare a variable to hold the XSQLDA needed to store the column data for each row that will be fetched. For example, the following declaration creates an XSQLDA called *out_sqlda*:

```
XSQLDA *out_sqlda;
```

2. Optionally declare a variable for accessing the XSQLVAR structure of the XSQLDA:

   ```
   XSQLVAR *var;
   ```

   Declaring a pointer to the XSQLVAR structure is not necessary, but can simplify referencing the structure in subsequent statements.

3. Allocate memory for the XSQLDA using the XSQLDA_LENGTH macro. The following statement allocates storage for *out_sqlda*:

   ```
   out_sqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(10));
   ```

   Space for 10 XSQLVAR structures is allocated in this statement, enabling the XSQLDA to accommodate up to 10 select-list items.

4. Set the *version* field of the XSQLDA to SQLDA_VERSION1, and set the *sqln* field of the XSQLDA to indicate the number of XSQLVAR structures allocated:

   ```
   out_sqlda->version = SQLDA_VERSION1;
   out_sqlda->sqln = 10;
   ```

Preparing a query statement string without parameters

After an XSQLDA is created for holding the items returned by a query statement string, the statement string can be created, prepared, and described. When a statement string is executed, InterBase creates the select-list of selected rows.

To prepare a query statement string, follow these steps:

1. Elicit a statement string from the user or create one that contains the SQL statement to be processed. For example, the following statement creates an SQL statement string that performs a query:

   ```
   char *str = "SELECT * FROM CUSTOMER";
   ```

   The statement appears to have only one select-list item (*). The asterisk is a wildcard symbol that stands for all of the columns in the table, so the actual number of items returned equals the number of columns in the table.

2. Declare and initialize an SQL statement handle, then allocate it with **isc_dsql_allocate()**:

   ```
   isc_stmt_handle stmt; /* Declare a statement handle. */
   stmt = NULL; /* Set handle to NULL before allocation. */
   . . .
   isc_dsql_allocate_statement(status_vector, &db1, &stmt);
   ```

3. Parse the statement string with **isc_dsql_prepare()**. This sets the statement handle (*stmt*) to refer to the parsed format. The statement handle is used in subsequent calls to statements such as **isc_dsql_describe()** and **isc_dsql_execute()**:

   ```
   isc_dsql_prepare(status_vector, &trans, &stmt, 0, str, 1, NULL);
   ```
4. Use **isc_dsql_describe()** to fill the output XSQLDA with information about the select-list items returned by the statement:

   ```
   isc_dsql_describe(status_vector, &trans, &stmt, out_sqlda);
   ```
5. Compare the *sqln* field of the XSQLDA to the *sqld* field to determine if the output descriptor can accommodate the number of select-list items specified in the statement. If not, free the storage previously allocated to the output descriptor, reallocate storage to reflect the number of select-list items specified by *sqld*, reset *sqln* and *version*, then execute **isc_dsql_describe()** again:

   ```
   if (out_sqlda->sqld > out_sqlda->sqln)
   {
      n = out_sqlda->sqld;
      free(out_sqlda);
      out_sqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(n));
      out_sqlda->sqln = n;
      out_sqlda->version = SQLDA_VERSION1;
      isc_dsql_describe(status_vector, &trans, 1, out_sqlda);
   }
   ```
6. Set up an XSQLVAR structure for each item returned. Setting up an item structure involves the following steps:

   - Coercing an item's datatype (optional).
   - Allocating local storage for the data pointed to by the *sqldata* field of the XSQLVAR. This step is only required if space for local variables is not allocated until runtime. The following example illustrates dynamic allocation of local variable storage space.
   - Providing a NULL value indicator for the parameter.

   The following code example illustrates these steps, looping through each XSQLVAR structure in the *out_sqlda* XSQLDA:

```
for (i=0, var = out_sqlda->sqlvar; i < out_sqlda->sqld; i++, var++)
{
   dtype = (var->sqltype & ~1) /* drop flag bit for now */
   switch(dtype)
   {
      case SQL_VARYING:
         var->sqltype = SQL_TEXT;
```

```
            var->sqldata = (char *)malloc(sizeof(char)*var->sqllen + 2);
            break;
         case SQL_TEXT:
            var->sqldata = (char *)malloc(sizeof(char)*var->sqllen);
            break;
         case SQL_LONG:
            var->sqldata = (char *)malloc(sizeof(long));
            break;
            . . .
            /* process remaining types */
      }  /* end of switch statements */
      if (sqltype & 1)
      {
         /* allocate variable to hold NULL status */
         var->sqlind = (short *)malloc(sizeof(short));
      }
   }     /* end of for loop */
```

For more information about datatype coercion and NULL indicators, see **"Coercing datatypes" on page 92**.

Executing a statement string within the context of a cursor

To retrieve select-list items from a prepared statement string, the string can be executed within the context of a cursor. All cursor declarations in InterBase are fixed statements inserted into the application before it is compiled. DSQL application developers must anticipate the need for cursors when writing the application and declare them ahead of time.

A cursor is only needed to process positioned UPDATE and DELETE statements made against the rows retrieved by **isc_dsql_fetch()** for SELECT statements that specify an optional FOR UPDATE OF clause.

The following descriptions apply to the situations when a cursor is needed. For an example of executing a statement and fetching rows without using a cursor, see **"isc_dsql_fetch()" on page 280**.

A looping construct is used to fetch a single row at a time from the cursor and to process each select-list item (column) in that row before the next row is fetched.

To execute a statement string within the context of a cursor and retrieve rows of select-list items, follow these steps:

1. Execute the prepared statement with **isc_dsql_execute()**:

   ```
   isc_dsql_execute(status_vector, &trans, &stmt, 1, NULL);
   ```

2. Declare and open a cursor for the statement string with **isc_dsql_set_cursor_name()**. For example, the following statement declares a cursor, *dyn_cursor*, for the SQL statement string, *stmt*:

```
    isc_dsql_set_cursor_name(status_vector, &stmt, "dyn_cursor",
NULL);
```

   Opening the cursor causes the statement string to be executed, and an active set of rows to be retrieved.

3. Fetch one row at a time and process the select-list items (columns) it contains with **isc_dsql_fetch()**. For example, the following loops retrieve one row at a time from *dyn_cursor* and process each item in the retrieved row with an application-specific function called **process_column()**:

```
while ((fetch_stat =
          isc_dsql_fetch(status_vector, &stmt, 1, out_sqlda))
        == 0)
{
    for (i = 0; i < out_sqlda->sqld; i++)
    {
        process_column(sqlda->sqlvar[i]);
    }
}
if (fetch_stat != 100L)
{
    /* isc_dsql_fetch returns 100 if no more rows remain to be
        retrieved */
    SQLCODE = isc_sqlcode(status_vector);
    isc_print_sqlerror(SQLCODE, status_vector);
    return(1);
}
```

   The **process_column()** function mentioned in this example processes each returned select-list item. The following skeleton code illustrates how such a function can be set up:

```
void process_column(XSQLVAR *var)
{
    /* test for NULL value */
    if ((var->sqltype & 1) && (*(var->sqlind) = -1))
    {
        /* process the NULL value here */
    }
    else
    {
        /* process the data instead */
    }
. . .
}
```

4. When all the rows are fetched, close the cursor with **isc_dsql_free_statement()**:

   ```
   isc_dsql_free_statement(status_vector, &stmt, DSQL_close);
   ```

Re-executing a query statement string without parameters

Once a query statement string without parameters is prepared, it can be executed as often as required in an application by closing and reopening its cursor.

To reopen a cursor and process select-list items, repeat steps 2 through 4 of **"Executing a statement string within the context of a cursor" on page 102**.

Method 4: Query statements with parameters

There are four steps to processing an SQL query statement string with placeholder parameters:

1. Prepare an input XSQLDA to process a statement string's parameters.
2. Prepare an output XSQLDA to process the select-list items returned when the query is executed.
3. Prepare the statement string and its parameters.
4. Use a cursor to execute the statement using input parameter values from an input XSQLDA, and to retrieve select-list items from the output XSQLDA.

Preparing the input XSQLDA

Placeholder parameters are replaced with actual data before a prepared SQL statement string is executed. Because those parameters are unknown when the statement string is created, an input XSQLDA must be created to supply parameter values at runtime. To prepare the XSQLDA, follow these steps:

1. Declare a variable to hold the XSQLDA needed to process parameters. For example, the following declaration creates an XSQLDA called *in_sqlda*:

   ```
   XSQLDA *in_sqlda;
   ```

2. Optionally declare a variable for accessing the XSQLVAR structure of the XSQLDA:

   ```
   XSQLVAR *var;
   ```

Declaring a pointer to the XSQLVAR structure is not necessary, but can simplify referencing the structure in subsequent statements.

3. Allocate memory for the XSQLDA using the XSQLDA_LENGTH macro. The following statement allocates storage for *in_slqda*:

   ```
   in_sqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(10));
   ```

   In this statement, space for 10 XSQLVAR structures is allocated, allowing the XSQLDA to accommodate up to 10 input parameters. Once structures are allocated, assign values to the *sqldata* fields.

4. Set the *version* field of the XSQLDA to SQLDA_VERSION1, and set the *sqln* field of the XSQLDA to indicate the number of XSQLVAR structures allocated:

   ```
   in_sqlda->version = SQLDA_VERSION1;
   in_sqlda->sqln = 10;
   ```

Preparing the output XSQLDA

Most queries return one or more rows of data, referred to as a *select-list*. Because the number and kind of items returned are unknown when a statement string is executed, an output XSQLDA must be created to store select-list items that are returned at runtime. To prepare the XSQLDA, follow these steps:

1. Declare a variable to hold the XSQLDA needed to process parameters. For example, the following declaration creates an XSQLDA called *out_sqlda*:

   ```
   XSQLDA *out_sqlda;
   ```

2. Optionally declare a variable for accessing the XSQLVAR structure of the XSQLDA:

   ```
   XSQLVAR *var;
   ```

   Declaring a pointer to the XSQLVAR structure is not necessary, but can simplify referencing the structure in subsequent statements.

3. Allocate memory for the XSQLDA using the XSQLDA_LENGTH macro. The following statement allocates storage for *out_sqlda*:

   ```
   out_sqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(10));
   ```

   Space for 10 XSQLVAR structures is allocated in this statement, enabling the XSQLDA to accommodate up to 10 select-list items.

4. Set the *version* field of the XSQLDA to SQLDA_VERSION1, and set the *sqln* field of the XSQLDA to indicate the number of XSQLVAR structures allocated:

   ```
   out_sqlda->version = SQLDA_VERSION1;
   out_sqlda->sqln = 10;
   ```

Preparing a query statement string with parameters

After an input and an output XSQLDA are created for holding a statement string's parameters, and the select-list items returned when the statement is executed, the statement string can be created and prepared. When a statement string is prepared, InterBase replaces the placeholder parameters in the string with information about the actual parameters used. The information about the parameters must be assigned to the input XSQLDA (and perhaps adjusted) before the statement can be executed. When the statement string is executed, InterBase stores select-list items in the output XSQLDA.

To prepare a query statement string with parameters, follow these steps:

1. Elicit a statement string from the user or create one that contains the SQL statement to be processed. For example, the following statement creates an SQL statement string with placeholder parameters:

```
    char *str = "SELECT * FROM DEPARTMENT WHERE BUDGET = ?, LOCATION =
?";
```

   This statement string contains two parameters: a value to be assigned to the BUDGET column and a value to be assigned to the LOCATION column.

2. Declare and initialize an SQL statement handle, then allocate it with **isc_dsql_allocate()**:

```
    isc_stmt_handle stmt; /* Declare a statement handle. */
    stmt = NULL; /* Set handle to NULL before allocation. */
    . . .
    isc_dsql_allocate_statement(status_vector, &db1, &stmt);
```

3. Prepare the statement string with **isc_dsql_prepare()**. This sets the statement handle (*stmt*) to refer to the parsed format. The statement handle is used in subsequent calls to **isc_dsql_describe()**, **isc_dsql_describe_bind()**, and **isc_dsql_execute2()**:

```
    isc_dsql_prepare(status_vector, &trans, &stmt, 0, str, 1,
out_xsqlda);
```

4. Use **isc_dsql_describe_bind()** to fill the input XSQLDA with information about the parameters contained in the SQL statement:

```
    isc_dsql_decribe_bind(status_vector, &stmt, 1, in_xsqlda);
```

5. Compare the *sqln* field of the XSQLDA to the *sqld* field to determine if the input descriptor can accommodate the number of parameters contained in the statement. If not, free the storage previously allocated to the input descriptor, reallocate storage to reflect the number of parameters specified by *sqld*, reset *sqln* and *version*, then execute **isc_dsql_describe_bind()** again:

```
    if (in_sqlda->sqld > in_sqlda->sqln)
    {
       n = in_sqlda->sqld;
       free(in_sqlda);
       in_sqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(n));
       in_sqlda->sqln = n;
       in_sqlda->version = SQLDA_VERSION1;
       isc_dsql_decribe_bind(status_vector, &stmt, 1, in_xsqlda);
    }
```

6. Process each XSQLVAR parameter structure in the input XSQLDA. Processing a parameter structure involves up to four steps:

   - Coercing a parameter's datatype (optional).
   - Allocating local storage for the data pointed to by the *sqldata* field of the XSQLVAR. This step is only required if space for local variables is not allocated until runtime. The following example illustrates dynamic allocation of local variable storage space.
   - Providing a value for the parameter consistent with its datatype (required).
   - Providing a NULL value indicator for the parameter.

   These steps must be followed in the order presented. The following code example illustrates these steps, looping through each XSQLVAR structure in the *in_sqlda* XSQLDA:

```
for (i=0, var = in_sqlda->sqlvar; i < in_sqlda->sqld; i++, var++)
{
    /* Process each XSQLVAR parameter structure here.
    The parameter structure is pointed to by var.*/
    dtype = (var->sqltype & ~1) /* drop flag bit for now */
    switch(dtype)
    {
       case SQL_VARYING: /* coerce to SQL_TEXT */
          var->sqltype = SQL_TEXT;
          /* allocate proper storage */
          var->sqldata = (char *)malloc(sizeof(char)*var->sqllen);
          /* Provide a value for the parameter. See case SQL_LONG. */
          . . .
          break;
       case SQL_TEXT:
          var->sqldata = (char *)malloc(sizeof(char)*var->sqllen);
          /* Provide a value for the parameter. See case SQL_LONG. */
          . . .
          break;
```

```
        case SQL_LONG:
            var->sqldata = (char *)malloc(sizeof(long));
            /* Provide a value for the parameter. */
            *(long *)(var->sqldata) = 17;
            break;
        . . .
    }   /* end of switch statement */
    if (sqltype & 1)
    {
        /* allocate variable to hold NULL status */
        var->sqlind = (short *)malloc(sizeof(short));
    }
}       /* end of for loop */
```

For more information about datatype coercion and NULL indicators, see **"Coercing datatypes" on page 92**.

7. Use **isc_dsql_describe()** to fill the output XSQLDA with information about the select-list items returned by the statement:

```
isc_dsql_describe(status_vector, &trans, &stmt, out_xsqlda);
```

8. Compare the *sqln* field of the XSQLDA to the *sqld* field to determine if the output descriptor can accommodate the number of select-list items specified in the statement. If not, free the storage previously allocated to the output descriptor, reallocate storage to reflect the number of select-list items specified by *sqld*, reset *sqln* and *version*, and execute DESCRIBE OUTPUT again:

```
if (out_sqlda->sqld > out_sqlda->sqln)
{
    n = out_sqlda->sqld;
    free(out_sqlda);
    out_sqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(n));
    out_sqlda->sqln = n;
    out_sqlda->version = SQLDA_VERSION1;
    isc_dsql_describe(status_vector, &trans, &stmt, out_xsqlda);
}
```

9. Set up an XSQLVAR structure for each item returned. Setting up an item structure involves the following steps:

   - Coercing an item's datatype (optional).
   - Allocating local storage for the data pointed to by the *sqldata* field of the XSQLVAR. This step is only required if space for local variables is not allocated until runtime. The following example illustrates dynamic allocation of local variable storage space.

- Providing a NULL value indicator for the parameter (optional).

The following code example illustrates these steps, looping through each XSQLVAR structure in the *out_sqlda* XSQLDA:

```
for (i=0, var = out_sqlda->sqlvar; i < out_sqlda->sqld; i++, var++)
{
   dtype = (var->sqltype & ~1) /* drop flag bit for now */
   switch(dtype)
   {
      case SQL_VARYING:
         var->sqltype = SQL_TEXT;
      break;
      case SQL_TEXT:
         var->sqldata = (char *)malloc(sizeof(char)*var->sqllen);
         break;
      case SQL_LONG:
         var->sqldata = (char *)malloc(sizeof(long));
         break;
         /* process remaining types */
   }  /* end of switch statements */
   if (sqltype & 1)
   {
      /* allocate variable to hold NULL status */
      var->sqlind = (short *)malloc(sizeof(short));
   }
}     /* end of for loop */
```

For more information about datatype coercion and NULL indicators, see **"Coercing datatypes" on page 92**.

Executing a query statement string within the context of a cursor

To retrieve select-list items from a statement string, the string must be executed within the context of a cursor. All cursor declarations in InterBase are fixed, embedded statements inserted into the application before it is compiled. DSQL application developers must anticipate the need for cursors when writing the application and declare them ahead of time.

A looping construct is used to fetch a single row at a time from the cursor and to process each select-list item (column) in that row before the next row is fetched.

To execute a statement string within the context of a cursor and retrieve rows of select-list items, follow these steps:

1. Execute the statement with **isc_dsql_execute2()**:

```
isc_dsql_execute2(status_vector, &trans, &stmt, 1, in_xsqlda,
out_xsqlda);
```

2. Declare and open a cursor for the statement string with **isc_dsql_set_cursor_name()**. For example, the following statement declares a cursor, *dyn_cursor*, for the prepared SQL statement string, *stmt*:

```
   isc_dsql_set_cursor_name(status_vector, &stmt, "dyn_cursor",
NULL);
```

   Opening the cursor causes the statement string to be executed, and an active set of rows to be retrieved.

3. Fetch one row at a time with **isc_dsql_fetch()** and process the select-list items (columns) it contains. For example, the following loops retrieve one row at a time from *dyn_cursor* and process each item in the retrieved row with an application-specific function called **process_column()**:

```
while ((fetch_stat =
   isc_dsql_fetch(status_vector, &stmt, 1, out_sqlda)) == 0)
{
   for (i = 0; i < out_sqlda->sqld; i++)
   {
      process_column(sqlda->sqlvar[i]);
   }
}
if (fetch_stat != 100L)
{
   /* isc_dsql_fetch returns 100 if no more rows remain to be
      retrieved */
   SQLCODE = isc_sqlcode(status_vector);
   isc_print_sqlerror(SQLCODE, status_vector);
   return(1);
}
```

4. When all the rows are fetched, close the cursor with **isc_dsql_free_statement()**:

```
   isc_dsql_free_statement(status_vector, &stmt, DSQL_close);
```

Re-executing a query statement string with parameters

Once a query statement string with parameters is prepared, it can be used as often as required in an application. Before each subsequent use, the input XSQLDA can be supplied with new parameter and NULL indicator data. The cursor must be closed and reopened before processing can occur.

- To provide new parameters to the input XSQLDA, follow steps 3 to 5 of **"Preparing a query statement string with parameters" on page 106**.
- To provide new information to the output XSQLDA, follow steps 6 to 8 of **"Preparing a query statement string with parameters" on page 106**.
- To reopen a cursor and process select-list items, repeat steps 2 to 4 of **"Executing a query statement string within the context of a cursor" on page 109**.

Determining an unknown statement type at runtime

An application can use **isc_dsql_sql_info()** to determine the statement type of an unknown prepared statement, for example, a statement entered by the user at runtime.

Requested information can include:

- Statement type.
- Number of input parameters required by the statement.
- Number of output values returned by the statement.
- Detailed information regarding each input parameter or output value, including its datatype, scale, and length.

To use **isc_dsql_sql_info()**, allocate an item-list buffer that describes the type of information requested, and allocate a result buffer, where the function can return the desired information. For example, to determine the statement type of an unknown, but prepared statement, you would allocate a one-element item-list buffer, and fill it with the macro constant, *isc_info_sql_stmt_type*, defined in *ibase.h*:

```
char type_item[];
type_item[] = {isc_info_sql_stmt_type};
```

Note Additional information item macros for requested items can be found in *ibase.h* under the comment, "SQL information items."

The result buffer must be large enough to contain any data returned by the call. The proper size for this buffer depends on the information requested. If not enough space is allocated, then **isc_dsql_sql_info()** puts the predefined value, *isc_info_truncated*, in the last byte of the result buffer. Generally, when requesting statement type information, 8 bytes is a sufficient buffer size. Declaring a larger than necessary buffer is also safe. A request to identify a statement type returns the following information in the result buffer:

1. One byte containing *isc_info_sql_stmt_type*.
2. Two bytes containing a number, *n*, telling how many bytes compose the subsequent *value*.
3. One or two bytes specifying the statement type. The following table lists the statement types that can be returned:

Type	Numeric value
isc_info_sql_stmt_select	1
isc_info_sql_stmt_insert	2
isc_info_sql_stmt_update	3
isc_info_sql_stmt_delete	4
isc_info_sql_stmt_ddl	5
isc_info_sql_stmt_get_segment	6
isc_info_sql_stmt_put_segment	7
isc_info_sql_stmt_exec_procedure	8
isc_info_sql_stmt_start_trans	9
isc_info_sql_stmt_commit	10
isc_info_sql_stmt_rollback	11
isc_info_sql_stmt_select_for_upd	12

TABLE 6.6 Statement types

4. A final byte containing the value *isc_info_end* (0).

The values placed in the result buffer are not aligned. Furthermore, all numbers are represented in a generic format, with the least significant byte first, and the most significant byte last. Signed numbers have the sign in the last byte. Convert the numbers to a datatype native to your system before interpreting them.

Note All information about a statement except its type can be more easily determined by calling functions other than **isc_dsql_sql_info()**. For example, to determine the information to fill in an input XSQLDA, call **isc_dsql_describe_bind()**. To fill in an output XSQLDA, call **isc_dsql_prepare()** or **isc_dsql_describe()**.

CHAPTER

7

Working with Blob Data

This chapter describes InterBase's dynamically sizable datatype, called a Blob, and describes how to work with it using API functions. Depending on a particular application, you might need to read all or only part of the chapter.

For example, if you plan to request conversion of Blob data from one datatype to another, such as from one bitmapped graphic format to another or from the MIDI sound format to the Wave format, you need to read the entire chapter. To write a conversion routine, called a *filter*, see **"Filtering Blob data" on page 132**. For further information about working with Blob data and filters, see the *Programmer's Guide*.

Note Blob filters are not available on NetWare servers.

If you do not need to request conversion of Blob data, then you only need to read the parts of this chapter up to **"Filtering Blob data" on page 132**.

The following table alphabetically lists the API functions for working with Blob data. The functions will be described and demonstrated in the remainder of this chapter.

Function	Purpose
isc_blob_default_desc()	Loads a Blob descriptor with default information about a Blob, including its subtype, character set, and segment size
isc_blob_gen_bpb()	Generates a Blob parameter buffer (BPB) from source and target Blob descriptors to allow dynamic access to Blob subtype and character set information
isc_blob_info()	Returns information about an open Blob
isc_blob_lookup_desc()	Determines the subtype, character set, and segment size of a Blob, given a table name and Blob column name
isc_blob_set_desc()	Initializes a Blob descriptor from parameters passed to it
isc_cancel_blob()	Discards a Blob
isc_close_blob()	Closes an open Blob
isc_create_blob2()	Creates and opens a Blob for write access, and optionally specifies a filter to be used to translate the Blob from one subtype to another
isc_get_segment()	Retrieves data from a Blob column in a row returned by execution of a SELECT statement
isc_open_blob2()	Opens an existing Blob for retrieval, and optionally specifies a filter to be used to translate the Blob from one subtype to another
isc_put_segment()	Writes data into a Blob

TABLE 7.1 API Blob functions

What is a Blob?

A *Blob* large object that cannot easily be stored in a database as one of the standard datatypes. You can use a Blob to store large amounts of data of various types, including:

- Bitmapped images
- Sounds
- Video segments

- Text

InterBase support of Blob data provides all the advantages of a database management system, including transaction control, maintenance, and access using standard API function calls. Blob data is stored in the database itself. Other systems only store pointers in the database to non-database files. InterBase stores the actual Blob data in the database, and establishes a unique identification handle in the appropriate table to point to the database location of the Blob. By maintaining the Blob data within the database, InterBase greatly improves access to and management of the data.

How are Blob data stored?

Blob is the InterBase datatype that can represent various objects, such as bitmapped images, sound, video, and text. Before you store these items in the database, you create or manage them as platform- or product-specific files or data structures, such as:

- TIFF, PICT, .BMP, .WMF, .GEM, TARGA or other bitmapped or vector-graphic files
- MIDI or .WAV sound files
- Audio Video Interleaved Format (.AVI) or QuickTime video files
- ASCII, .MIF, .DOC, .WPx or other text files
- CAD files

You must programmatically load these files from memory into the database, as you do any other data items or records you intend to store in InterBase. For more information about creating a Blob and storing data into it, see **"Writing data to a Blob" on page 122**.

Blob subtypes

Although you manage Blob data in ways similar to other datatypes, InterBase provides more flexible data typing rules for Blob data. Because there are many native datatypes that you can define as Blob data, InterBase treats them generically and allows you to define your own datatype, known as a *subtype*. InterBase also provides two predefined subtypes: 0, an unstructured subtype generally applied to binary data or data of an indeterminate type, and 1, applied to plain text.

User-defined subtypes must always be represented as negative integers between –1 and –32,678.

A Blob column's subtype is specified when the Blob column is defined.

The application is responsible for ensuring that data stored in a Blob column agrees with its subtype; InterBase does not check the type or format of Blob data.

Blob database storage

Rather than storing Blob data directly in the Blob field of a table record, InterBase stores a *Blob ID* there. A *Blob ID* is a unique numeric value that references Blob data. The Blob data is stored elsewhere in the database, in a series of Blob *segments*, units of Blob data read and written in chunks. Blob segments can be of varying length. The length of an individual segment is specified when it is written.

Segments are handy when working with data that is too large for one application memory buffer. But it is not necessary to use multiple segments; you can put all your Blob data in a single segment.

When an application creates a Blob, it must write data to it a segment at a time. When an application reads a Blob, it reads a segment at a time. For more information about writing segments, see **"Writing data to a Blob" on page 122**. For more information about reading segments, see **"Reading data from a Blob" on page 117**.

Blob data operations

InterBase supports the following operations on Blob data:

- Reading from a Blob
- Writing to a Blob, which involves the following operations:
  1. Inserting a new row that includes Blob data.
  2. Replacing the data referenced by a Blob column of a row.
  3. Updating the data referenced by a Blob column of a row.
- Deleting a Blob

The following sections describe how to perform these operations. These examples do not include the use of filters to convert data from one subtype to another as it is read or written. For information about using filters, see **"Writing an application that requests filtering" on page 139**.

Dynamic SQL (DSQL) API functions and the XSQLDA data structure are needed to execute SELECT, INSERT, and UPDATE statements required to select, insert, or update relevant Blob data. The following sections include descriptions of the DSQL programming methods required to execute the sample statements provided. For more information about DSQL programming, see **Chapter 6, "Working with Dynamic SQL."**

Reading data from a Blob

There are six steps required for reading data from an existing Blob:

1. Create a SELECT statement query that specifies selection of the Blob column (and any other columns desired) in the rows of interest.
2. Prepare an output XSQLDA structure to hold the column data for each row that is fetched.
3. Prepare the SELECT statement for execution.
4. Execute the statement.
5. Fetch the selected rows one by one.
6. Read and processing the Blob data from each row.

Creating the SELECT *statement*

Elicit a statement string from the user or create one that consists of the SQL query that will select rows containing the Blob data of interest. For example, the following creates an SQL query statement string that selects three columns from various rows in the PROJECT table:

```
char *str =
"SELECT PROJ_NAME, PROJ_DESC, PRODUCT FROM PROJECT WHERE \
PRODUCT IN ("software", "hardware", "other") ORDER BY PROJ_NAME";
```

Preparing the output XSQLDA

Most queries return one or more rows of data, referred to as a *select-list*. An output XSQLDA must be created to store the column data for each row that is fetched. For a Blob column, the column data is an internal Blob identifier (*Blob ID*) that is needed to access the actual data. To prepare the XSQLDA, follow these steps:

1. Declare a variable to hold the XSQLDA. For example, the following declaration creates an XSQLDA called *out_sqlda*:

   ```
   XSQLDA *out_sqlda;
   ```

2. Allocate memory for the XSQLDA using the XSQLDA_LENGTH macro. The XSQLDA must contain one XSQLVAR substructure for each column to be fetched. The following statement allocates storage for an output XSQLDA (*out_sqlda*) with three XSQLVAR substructures:

   ```
   out_sqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(3));
   ```

3. Set the *version* field of the XSQLDA to SQLDA_VERSION1, and set the *sqln* field of the XSQLDA to indicate the number of XSQLVAR substructures allocated:

   ```
   out_sqlda->version = SQLDA_VERSION1;
   out_sqlda->sqln = 3;
   ```

Preparing the SELECT *statement for execution*

After an XSQLDA is created for holding the column data for each selected row, the query statement string can be prepared for execution. Follow these steps:

1. Declare and initialize an SQL statement handle, then allocate it with **isc_dsql_allocate_statement()**:

   ```
   isc_stmt_handle stmt; /* Declare a statement handle. */
   stmt = NULL;     /* Set handle to NULL before allocation. */
   isc_dsql_allocate_statement(status_vector, &db_handle, &stmt);
   ```

2. Ready the statement string for execution with **isc_dsql_prepare()**. This checks the string (*str*) for syntax errors, parses it into a format that can be efficiently executed, and sets the statement handle (*stmt*) to refer to this parsed format. The statement handle is used in a later call to **isc_dsql_execute()**.

   If **isc_dsql_prepare()** is passed a pointer to the output XSQLDA, as in the following example, it will fill in most fields of the XSQLDA and all its XSQLVAR substructures with information such as the datatype, length, and name of the corresponding columns in the statement.

   A sample call to **isc_dsql_prepare()** is:

```
    isc_dsql_prepare(
        status_vector,
        &trans,      /* Set by previous isc_start_transaction() call.
*/
        &stmt,       /* Statement handle set by this function call. */
        0,           /* Specifies statement string is null-terminated.
*/
        str,         /* Statement string. */
        SQLDA_VERSION1,/* XSQLDA version number. */
        out_sqlda    /* XSQLDA for storing column data. */
```

```
);
```

3. Set up an XSQLVAR structure for each column. Setting up an XSQLVAR structure involves the following steps:

   For columns whose types are known at compile time:

   - Specify the column's datatype (if it was not set by **isc_dsql_prepare()**, as previously described).
   - Point the *sqldata* field of the XSQLVAR to an appropriate local variable.

   For columns whose types are not known until run time:

   - Coerce the item's datatype (optional), for example, from SQL_VARYING to SQL_TEXT.
   - Dynamically allocate local storage for the data pointed to by the *sqldata* field of the XSQLVAR.

   For both:

   - Specify the number of bytes of data to be retrieved into *sqldata*.
   - Provide a NULL value indicator for the parameter.

   Data retrieval for Blob (and array) columns is different from other types of columns, so the XSQLVAR fields must be set differently. For non-Blob (and non-array) columns, **isc_dsql_prepare()** sets each XSQLVAR *sqltype* field to the appropriate field type, and the data retrieved when a select-list row's data is fetched is placed into the *sqldata* space allocated for the column. For Blob columns, the type must be set to SQL_Blob (or SQL_Blob + 1 if a NULL indicator is desired). InterBase stores the internal Blob identifier (Blob ID), not the Blob data, in the *sqldata* space when a row's data is fetched, so you must point *sqldata* to an area the size of a Blob ID. To see how to retrieve the actual Blob data once you have a Blob ID, see **"Reading data from a Blob" on page 117**.

   The following code example illustrates the assignments for Blob and non-Blob columns whose types are known at compile time. For examples of handling datatypes that are unknown until run time, see **Chapter 6, "Working with Dynamic SQL."**

```
#define PROJLEN 20
#define TYPELEN 12
ISC_QUAD blob_id;
char proj_name[PROJLEN + 1];
char prod_type[TYPELEN + 1];
short flag0, flag1, flag2;
out_sqlda->sqlvar[0].sqldata = proj_name;
out_sqlda->sqlvar[0].sqltype = SQL_TEXT + 1;
out_sqlda->sqlvar[0].sqllen = PROJLEN;
out_sqlda->sqlvar[0].sqlind = &flag0;
```

```
   out_sqlda->sqlvar[1].sqldata = (char *) &blob_id;
   out_sqlda->sqlvar[1].sqltype = SQL_Blob + 1;
   out_sqlda->sqlvar[1].sqllen = sizeof(ISC_QUAD);
   out_sqlda->sqlvar[1].sqlind = &flag1;
   out_sqlda->sqlvar[2].sqldata =  prod_type;
   out_sqlda->sqlvar[2].sqltype = SQL_TEXT + 1;
   out_sqlda->sqlvar[2].sqllen = TYPELEN;
   out_sqlda->sqlvar[2].sqlind = &flag2;
```

Executing the statement

Once the query statement string is prepared, it can be executed:

```
isc_dsql_execute(
   status_vector,
   &trans,   /* set by previous isc_start_transaction() call */
   &stmt,    /* allocated above by isc_dsql_allocate_statement() */
   1,        /* XSQLDA version number */
   NULL      /* NULL since stmt doesn’t have input values */
   );
```

This statement creates a select list, the rows returned by execution of the statement.

Fetching selected rows

A looping construct is used to fetch (into the output XSQLDA) the column data for a single row at a time from the select-list and to process each row before the next row is fetched. Each execution of **isc_dsql_fetch()** fetches the column data into the corresponding XSQLVAR substructures of *out_sqlda*. For the Blob column, the Blob ID, not the actual Blob data, is fetched.

```
ISC_STATUS fetch_stat;
long SQLCODE;
. . .
while ((fetch_stat =
         isc_dsql_fetch(status_vector, &stmt, 1, out_sqlda))
      == 0)
{
   proj_name[PROJLEN] = ’\0’;
   prod_type[TYPELEN] = ’\0’;
   printf("\nPROJECT: %-20s TYPE: %-15s\n\n",
      proj_name, prod_type);
   /* Read and process the Blob data (see next section) */
}
if (fetch_stat != 100L)
```

```
{
   /* isc_dsql_fetch returns 100 if no more rows remain to be
      retrieved */
   SQLCODE = isc_sqlcode(status_vector);
   isc_print_sqlerror(SQLCODE, status_vector);
   return(1);
}
```

Reading and processing the Blob data

To read and process the Blob data:

1. Declare and initialize a Blob handle:

```
isc_blob_handle blob_handle; /* Declare a Blob handle. */
blob_handle = NULL; /* Set handle to NULL before using it */
```

2. Create a buffer for holding each Blob segment as it is read. Its size should be the maximum size segment your program expects to be read from the Blob.

```
char blob_segment[80];
```

3. Declare an unsigned short variable into which InterBase will store the actual length of each segment read:

```
unsigned short actual_seg_len;
```

4. Open the Blob with the fetched *blob_id*:

```
isc_open_blob2(
   status_vector,
   &db_handle,
   &trans,
   &blob_handle,/* set by this function to refer to the Blob */
   &blob_id, /* Blob ID put into out_sqlda by isc_dsql_fetch() */
   0,        /* BPB length = 0; no filter will be used */
   NULL      /* NULL BPB, since no filter will be used */
   );
```

5. Read all the Blob data by calling **isc_get_segment()** repeatedly to get each Blob segment and its length. Process each segment read. In the following example, “processing” consists of printing each Blob as it is read:

```
blob_stat = isc_get_segment(
   status_vector,
   &blob_handle,    /* set by isc_open_blob2()*/
   &actual_seg_len, /* length of segment read */
   sizeof(blob_segment), /* length of segment buffer */
```

```
      blob_segment     /* segment buffer */
      );
   while (blob_stat == 0 || status_vector[1] == isc_segment)
   {
      /* isc_get_segment returns 0 if a segment was successfully read.
*/
      /* status_vector[1] is set to isc_segment if only part of a */
      /* segment was read due to the buffer (blob_segment) not being */
      /* large enough. In that case, the following calls to */
      /* isc_get_segment() read the rest of the buffer. */
      printf("%*.*s", actual_seg_len, actual_seg_len, blob_segment);
      blob_stat = isc_get_segment(status_vector, &blob_handle,
         &actual_seg_len, sizeof(blob_segment), blob_segment);
      printf("\n");
   };
   printf("\n");
```

6. Close the Blob:

```
   isc_close_blob(status_vector, &blob_handle);
```

Writing data to a Blob

Before you can create a new Blob and write data to it, you must do at least one of the following:

- Include Blob data in a row to be inserted into a table.
- Replace the data referenced by a Blob column of a row.
- Update the data referenced by a Blob column of a row.

The entry in a Blob column of a row does not actually contain Blob data. Rather, it has a Blob ID referring to the data, which is stored elsewhere. So, to set or modify a Blob column, you need to set (or reset) the Blob ID stored in it. If a Blob column contains a Blob ID, and you modify the column to refer to a different Blob (or to contain NULL), the Blob referenced by the previously stored Blob ID will be deleted during the next garbage collection.

All these operations require the following steps:

1. Prepare an appropriate DSQL statement. This will be an INSERT statement if you are inserting a new row into a table, or an UPDATE statement for modifying a row. Each of these statements will need a corresponding input XSQLDA structure for supplying parameter values to the statement at run time. The Blob ID of a new Blob will be one of the values passed.
2. Create a new Blob, and write data into it.
3. Associate the Blob ID of the new Blob with the Blob column of the table row by executing the UPDATE or INSERT statement.

Note that you cannot update Blob data directly. If you want to modify Blob data, you must:

- Create a new Blob.
- Read the old Blob data into a buffer where you can edit or modify it.
- Write the modified data to the new Blob.
- Prepare and execute an UPDATE statement that will modify the Blob column to contain the Blob ID of the new Blob, replacing the old Blob's Blob ID.

The sections below describe the steps required to insert, replace, or update Blob data.

Preparing the UPDATE *or* INSERT *statement*

To prepare an UPDATE or INSERT statement for execution, follow these steps:

1. Elicit an UPDATE or INSERT statement string from the user or create one for inserting a row or updating the row containing the Blob column of interest. For example, the following statement is for updating the Blob column named PROJ_DESC in the row of the table, PROJECT, whose PROJ_ID field contains a value specified at run time:

   ```
   char *upd_str =
       "UPDATE PROJECT SET PROJ_DESC = ? WHERE PROJ_ID = ?";
   ```

   As an example of an INSERT statement, the following inserts a new row containing values in four columns:

   ```
   char *in_str = "INSERT INTO PROJECT (PROJ_NAME, PROJ_DESC, PRODUCT,
       PROJ_ID) VALUES (?, ?, ?, ?)";
   ```

   The remaining steps refer only to UPDATE statements, but the actions apply to INSERT statements as well.

2. Declare a variable to hold the input XSQLDA needed to supply parameter values to the UPDATE statement at run time. For example, the following declaration creates an XSQLDA called *in_sqlda*:

   ```
   XSQLDA *in_sqlda;
   ```

3. Allocate memory for the input XSQLDA using the XSQLDA_LENGTH macro. The XSQLDA must contain one XSQLVAR substructure for each parameter to be passed to the UPDATE statement. The following statement allocates storage for an input XSQLDA (*in_sqlda*) with two XSQLVAR substructures:

   ```
   in_sqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(2));
   ```

4. Set the *version* field of the XSQLDA to SQLDA_VERSION1, and set the *sqln* field to indicate the number of XSQLVAR structures allocated:

   ```
   in_sqlda->version = SQLDA_VERSION1;
   in_sqlda->sqln = 2;
   ```

5. Set up the XSQLVAR structure in the XSQLDA for each parameter to be passed. Setting up an XSQLVAR structure involves the following steps:

   - Specify the item's datatype
   - For parameters whose types are known at compile time: Point the *sqldata* field of the XSQLVAR to an appropriate local variable that will contain the data to be passed
   - For parameters whose types are not known until run time: Allocate local storage for the data pointed to by the *sqldata* field of the XSQLVAR
   - Specify the number of bytes of data

   Data storage for Blob (and array) columns is different from other types of columns, so the XSQLVAR fields must be set differently. For non-Blob and non-array columns, input parameter data comes from the space pointed to by *sqldata*. For Blob columns, you must set the type to SQL_Blob (or SQL_Blob + 1 if you want a NULL indicator). Your application must store space for the internal Blob identifier, not the Blob data, in the *sqldata* space. For more information about creating a Blob, storing its ID in the *sqldata* space, and associating the Blob with a column, see **"Creating a new Blob and storing data" on page 125**.

   The following code example illustrates the assignments for one text column and one Blob column, where the column types are known at compile time. For examples of handling datatypes that are unknown until run time, see **Chapter 6, "Working with Dynamic SQL."**

```
#define PROJLEN 5
char proj_id[PROJLEN + 1];
ISC_QUAD blob_id;
```

```
in_sqlda->sqlvar[0].sqldata = (char *) &blob_id;
in_sqlda->sqlvar[0].sqltype = SQL_Blob + 1;
in_sqlda->sqlvar[0].sqllen = sizeof(ISC_QUAD);
in_sqlda->sqlvar[1].sqldata = proj_id;
in_sqlda->sqlvar[1].sqltype = SQL_TEXT;
in_sqlda->sqlvar[1].sqllen = 5;
```

The *proj_id* variable should be assigned a value at run time (unless the value is known at compile time). The *blob_id* variable should be set to refer to the newly created Blob, as described in the following sections.

Creating a new Blob and storing data

To create a new Blob containing the data to be written:

1. Declare and initialize a Blob handle:

   ```
   isc_blob_handle blob_handle; /* Declare a Blob handle. */
   blob_handle = NULL; /* Set handle to NULL before using it */
   ```

2. Declare and initialize a Blob ID:

   ```
   ISC_QUAD blob_id;   /* Declare a Blob ID. */
   blob_id = NULL;     /* Set handle to NULL before using it */
   ```

3. Create a new Blob by calling **isc_create_blob2()**:

   ```
   isc_create_blob2(
       status_vector,
       &db_handle,
       &trans,
       &blob_handle, /* set by this function to refer to the new Blob */
       &blob_id, /* Blob ID set by this function */
       0,  /* Blob Parameter Buffer length = 0; no filter will be used
   */
       NULL /* NULL Blob Parameter Buffer, since no filter will be used
   */
       );
   ```

   This function creates a new Blob, opens it for write access, and sets *blob_handle* to point to the new Blob.

   isc_create_blob2() also assigns the Blob a Blob ID, and sets *blob_id* to point to the Blob ID. Note that *blob_id* is the variable pointed to by the *sqldata* field of the UPDATE statement input parameter that specifies the Blob column to be updated. Thus, when the UPDATE statement is executed, this new Blob will be used to update the Blob column.

4. Write all the data to be written to the Blob by making a series of calls to **isc_put_segment()**. The following example reads lines of data, and concatenates each to the Blob referenced by *blob_handle*. (**get_line()** reads the next line of data to be written.)

```
char *line;
unsigned short len;
. . .
line = get_line();
while (line)
{
   len = strlen(line);
   isc_put_segment(
         status_vector,
         &blob_handle,/* set by previous isc_create_blob2() */
         len,   /* length of buffer containing data to write */
         line   /* buffer containing data to write into Blob */
         );
   if (status_vector[0] == 1 && status_vector[1])
   {
      isc_print_status(status_vector);
      return(1);
   };
   line = get_line();
};
```

5. Close the Blob:

```
isc_close_blob(status_vector, &blob_handle);
```

Associating the new Blob with the Blob column

Execute the UPDATE statement to associate the new Blob with the Blob column in the row selected by the statement:

```
isc_dsql_execute_immediate(
   status_vector,
   &db_handle,
   &trans,
   0,        /* indicates string to execute is null-terminated */
   upd_str,  /* UPDATE statement string to be executed */
   1,        /* XSQLDA version number */
   in_sqlda  /* XSQLDA supplying parameters to UPDATE statement */
   );
```

Deleting a Blob

There are four ways to delete a Blob:

- Delete the row containing the Blob. You can use DSQL to execute a DELETE statement.
- Replace the Blob with a different one. If a Blob column contains a Blob ID, and you modify the column to refer to a different Blob, the Blob referenced by the previously stored Blob ID will be deleted during the next garbage collection.
- Reset to NULL the column referring to the Blob, for example, by using DSQL to execute a statement like the following:

  ```
  UPDATE PROJECT SET PROJ_DESC = NULL WHERE PROJ_ID = "VBASE"
  ```

  The Blob referenced by the previously stored Blob ID will be deleted during the next garbage collection.
- Discard a Blob after it has been created but before it has been associated with a particular column of a table row. Use the **isc_cancel_blob()** function, as in:

  ```
  isc_cancel_blob(status_vector, &blob_handle);
  ```

Requesting information about an open Blob

After an application opens a Blob, it can obtain information about the Blob. The **isc_blob_info()** call enables an application to query for Blob information such as the total number of segments in the Blob, or the length, in bytes, of the longest segment.

In addition to a pointer to the error status vector and a Blob handle, **isc_blob_info()** requires two application-provided buffers, an item-list buffer, where the application specifies the information it needs, and a result buffer, where InterBase returns the requested information. An application populates the item-list buffer with information requests prior to calling **isc_blob_info()**, and passes it both a pointer to the item-list buffer, and the size, in bytes, of that buffer.

The application must also create a result buffer large enough to hold the information returned by InterBase. It passes both a pointer to the result buffer, and the size, in bytes, of that buffer to **isc_blob_info()**. If InterBase attempts to pass back more information than can fit in the result buffer, it puts the value, *isc_info_truncated*, defined in *ibase.h*, in the final byte of the result buffer.

Item-list buffer items and result buffer values

The item-list buffer is a char array that holds a sequence of byte values, one per requested item of information. Each byte is an *item type*, specifying the kind of information desired. Compile-time constants for all item types are defined in *ibase.h*:

```
#define isc_info_blob_num_segments      4
#define isc_info_blob_max_segment       5
#define isc_info_blob_total_length      6
#define isc_info_blob_type              7
```

The result buffer returns a series of *clusters* of information, one per item requested. Each cluster consists of three parts:

1. A one-byte *item type*. Each is the same as one of the item types in the item-list buffer.
2. A 2-byte number specifying the number of bytes that follow in the remainder of the cluster.
3. A *value*, stored in a variable number of bytes, whose interpretation depends on the item type.

A calling program is responsible for interpreting the contents of the result buffer and for deciphering each cluster as appropriate.

The clusters returned to the result buffer are not aligned. Furthermore, all numbers are represented in a generic format, with the least significant byte first, and the most significant byte last. Signed numbers have the sign in the last byte. Convert the numbers to a datatype native to your system, if necessary, before interpreting them. The API call, **isc_vax_integer()**, can be used to perform the conversion.

The following table lists items about which information can be requested and returned, and the values reported:

Request and return item	**Return value**
isc_info_blob_num_segments	Total number of segments
isc_info_blob_max_segment	Length of the longest segment
isc_info_blob_total_length	Total size, in bytes, of Blob
isc_info_blob_type	Type of Blob (0: segmented, or 1: stream)

TABLE 7.2 Blob request and return items

In addition to the information InterBase returns in response to a request, InterBase can also return one or more of the following status messages to the result buffer. Each status message is one unsigned byte in length:

Item	Description
isc_info_end	End of the messages
isc_info_truncated	Result buffer is too small to hold any more requested information
isc_info_error	Requested information is unavailable. Check the status vector for an error code and message

TABLE 7.3 Status message return items

isc_blob_info( ) call example

The following code requests the number of segments and the maximum segment size for a Blob after it is opened, then examines the result buffer:

```
char blob_items[] = {
   isc_info_blob_max_segment, isc_info_blob_num_segments};
char res_buffer[20], *p, item;
short length;
SLONG max_size = 0L, num_segments = 0L;
ISC_STATUS status_vector[20];
isc_open_blob2(
   status_vector,
   &db_handle,  /* database handle, set by isc_attach_database() */
   &tr_handle, /* transaction handle, set by isc_start_transaction()
*/
   &blob_handle, /* set by this function to refer to the Blob */
   &blob_id,    /* Blob ID of the Blob to open */
   0,           /* BPB length = 0; no filter will be used */
   NULL         /* NULL BPB, since no filter will be used */
   );
if (status_vector[0] == 1 && status_vector[1])
{
   isc_print_status(status_vector);
   return(1);
}
isc_blob_info(
   status_vector,
   &blob_handle,    /* Set in isc_open_blob2() call above. */
   sizeof(blob_items),/* Length of item-list buffer. */
   blob_items,      /* Item-list buffer. */
   sizeof(res_buffer),/* Length of result buffer. */
   res_buffer       /* Result buffer */
   );
if (status_vector[0] == 1 && status_vector[1])
{
   /* An error occurred. */
   isc_print_status(status_vector);
   isc_close_blob(status_vector, &blob_handle);
   return(1);
};
/* Extract the values returned in the result buffer. */
for (p = res_buffer; *p != isc_info_end ;)
{
   item = *p++
   length = (short)isc_vax_integer(p, 2);
   p += 2;
   switch (item)
   {
       case isc_info_blob_max_segment:
           max_size = isc_vax_integer(p, length);
           break;
       case isc_info_blob_num_segments:
           num_segments = isc_vax_integer(p, length);
           break;
       case isc_info_truncated:
           /* handle error */
           break;
       default:
           break;
   }
   p += length;
};
```

Blob descriptors

A Blob descriptor is used to provide dynamic access to Blob information. For example, it can be used to store information about Blob data for filtering (conversion) purposes, such as character set information for text Blob data and subtype information for text and non-text Blob data. Two Blob descriptors are needed whenever a filter will be used when writing to or reading from a Blob: one to describe the filter source data, and the other to describe the target.

A Blob descriptor is a structure defined in the *ibase.h* header file as follows:

```
typedef struct {
    short blob_desc_subtype; /* type of Blob data */
    short blob_desc_charset; /* character set */
    short blob_desc_segment_size; /* segment size */
    unsigned char blob_desc_field_name [32]; /* Blob column name */
    unsigned char blob_desc_relation_name [32]; /* table name */
} ISC_Blob_DESC;
```

For more information about the character sets recognized by InterBase, see the *Language Reference*.

The segment size of a Blob is the maximum number of bytes that an application is expected to write to or read from the Blob. You can use this size to allocate your own buffers.

The *blob_desc_relation_name* and *blob_desc_field_name* fields contain null-terminated strings.

Populating a Blob descriptor

There are four possible ways to populate a Blob descriptor. You can do so by:

- Calling **isc_blob_default_desc()**. This stores default values into the descriptor fields. The default subtype is 1 (TEXT), segment size is 80 bytes, and charset is the default charset for your process.
- Calling **isc_blob_lookup_desc()**. This accesses the database system metadata tables to look up and copy information for the specified Blob column into the descriptor fields.
- Calling **isc_blob_set_desc()**. This initializes the descriptor from parameters you call it with, rather than accessing the database metadata.
- Setting the descriptor fields directly.

The following example calls **isc_blob_lookup_desc()** to look up the current subtype and character set information for a Blob column named PROJ_DESC in a table named PROJECT. It stores the information into the source descriptor, *from_desc*.

```
isc_blob_lookup_desc (
   status_vector,
   &db_handle;   /* Set by previous isc_attach_database() call. */
   &tr_handle,   /* Set by previous isc_start_transaction() call. */
   "PROJECT",    /* Table name. */
   "PROJ_DESC", /* Column name. */
   &from_desc,   /* Blob descriptor filled in by this function call. */
   &global       /* Global column name, returned by this function. */
   )
```

For more information about the usage of Blob descriptors in applications that request data filtering, and for further examples of populating Blob descriptors, see **"Writing an application that requests filtering" on page 139**.

Filtering Blob data

A Blob filter is a routine that translates Blob data from one subtype to another.

InterBase includes a set of special internal Blob filters that convert from subtype 0 (unstructured data) to subtype 1 (TEXT), and from subtype 1 to subtype 0.

In addition to using these standard filters, you can write your own external filters to provide special data translation. For example, you might develop a filter to convert one image format to another, for instance to display the same image on monitors with different resolutions. Or you might convert a binary Blob to plain text and back again to be able to move the file more easily from one system to another.

If you define filters, you can assign them subtype identifiers from –32,768 to –1.

The following sections provide an overview of how to write Blob filters, followed by details of how to write an application that requires filtering. For more information about writing Blob filters, see the *Programmer's Guide*.

Note Blob filters are available for databases residing on all InterBase server platforms except NetWare, where Blob filters cannot be created or used.

Using your own filters

Unlike the standard InterBase filters that convert between subtype 0 and subtype 1, an external Blob filter is generally part of a library of routines you create and link to an application.

You can write Blob filters in C or Pascal (or any language that can be called from C). To use your own filters, follow these steps:

1. Decide which filters you need to write.
2. Write the filters in a host language.
3. Build a shared filter library.
4. Make the filter library available.
5. Define the filters to the database.
6. Write an application that requests filtering.

Steps numbered 2, 5, and 6 are described in greater detail in the following sections.

Declaring an external Blob filter to the database

To declare an external filter to a database, use the DECLARE FILTER statement. For example, the following statement declares the filter, SAMPLE:

```
DECLARE FILTER SAMPLE
   INPUT TYPE –1 OUTPUT_TYPE –2
   ENTRY POINT "FilterFunction"
   MODULE_NAME "filter.dll";
```

In the example, the filter’s input subtype is defined as –1 and its output subtype as –2. If subtype –1 specifies lowercase text, and subtype –2 uppercase text, then the purpose of filter SAMPLE would be to translate Blob data from lowercase text to uppercase text.

The ENTRY_POINT and MODULE_NAME parameters specify the external routine that InterBase calls when the filter is invoked. The MODULE_NAME parameter specifies *filter.dll*, the dynamic link library containing the filter’s executable code. The ENTRY_POINT parameter specifies the entry point into the DLL. Although the example shows only a simple file name, it is good practice to specify a fully-qualified path name, since users of your application need to load the file.

Writing an external Blob filter

If you choose to write your own filters, you must have a detailed understanding of the datatypes you plan to translate. InterBase does not do strict datatype checking on Blob data; it is your responsibility.

Defining the filter function

When writing a filter, you must include an entry point, known as a *filter function*, in the declaration section of the program. InterBase calls the filter function when an application performs a Blob access operation on a Blob specified to use the filter. All communication between InterBase and the filter is through the filter function. The filter function itself may call other functions that comprise the filter executable.

You declare the name of the filter function and the name of the filter executable with the ENTRY_POINT and MODULE_NAME parameters of the DECLARE FILTER statement.

A filter function must have the following declaration *calling sequence*:

```
filter_function_name(short action, isc_blob_ctl control);
```

The parameter, *action*, is one of eight possible action macro definitions, and the parameter, *control*, is an instance of the *isc_blob_ctl* Blob control structure, defined in the InterBase header file, *ibase.h*. These parameters are discussed later in this chapter.

The following listing of a skeleton filter declares the filter function, *jpeg_filter*:

```
#include <ibase.h>
#define SUCCESS 0
#define FAILURE 1
ISC_STATUS jpeg_filter(short action, isc_blob_ctl control)
{
   ISC_STATUS status = SUCCESS;
   switch (action)
   {
   case isc_blob_filter_open:
      . . .
      break;
   case isc_blob_filter_get_segment:
      . . .
      break;
   case isc_blob_filter_create:
      . . .
      break;
   case isc_blob_filter_put_segment:
      . . .
      break;
   case isc_blob_filter_close:
      . . .
      break;
   case isc_blob_filter_alloc:
      . . .
      break;
   case isc_blob_filter_free:
      . . .
      break;
   case isc_blob_filter_seek:
      . . .
      break;
   default:
      . . .
      break;
   }
return status;
}
```

InterBase passes one of eight possible actions to the filter function, *jpeg_filter*, by way of the *action* parameter, and also passes an instance of the Blob control structure, *isc_blob_ctl*, by way of the parameter, *control*.

The ellipses (…) in the previous listing represent code that performs some operations based on each action, or event, that is listed in the case statement. Most of the actions correspond to API functions called by an application. For more information regarding the types of code to write for each action, see the *Programmer's Guide*.

Defining the Blob control structure

The *isc_blob_ctl* Blob control structure provides the fundamental method of data exchange between InterBase and a filter.

The Blob control structure is defined as a typedef, *isc_blob_ctl*, in *ibase.h*, as follows:

```
typedef struct isc_blob_ctl {
    ISC_STATUS (*ctl_source)();
        /* Internal InterBase Blob access routine. */
    struct isc_blob_ctl *ctl_source_handle;
        /* Instance of isc_blob_ctl to pass to
                internal InterBase Blob access routine. */
    short ctl_to_sub_type;/* Target subtype. */
    short ctl_from_sub_type;/* Source subtype. */
    unsigned short ctl_buffer_length; /* Length of ctl_buffer. */
    unsigned short ctl_segment_length; /* Length of current segment. */
    unsigned short ctl_bpb_length; /* Blob parameter buffer length. */
    char *ctl_bpb;   /* Pointer to Blob parameter buffer. */
    unsigned char *ctl_buffer; /* Pointer to segment buffer. */
    ISC_LONG ctl_max_segment; /* Length of longest Blob segment. */
    ISC_LONG ctl_number_segments; /* Total number of segments. */
    ISC_LONG ctl_total_length; /* Total length of Blob. */
    ISC_STATUS *ctl_status;/* Pointer to status vector. */
    long ctl_data[8];/* Application-specific data. */
} *ISC_Blob_CTL;
```

The purpose of certain *isc_blob_ctl* fields depend on the action being performed.

For example, when an application calls the **isc_put_segment()** API function, InterBase passes an *isc_blob_filter_put_segment* action to the filter function. The buffer pointed to by the *ctl_buffer* field of the control structure passed to the filter function contains the segment data to be written, as specified by the application in its call to **isc_put_segment()**. Because the buffer contains information passed into the filter function, it is called an IN field. The filter function should include instructions in the case statement under the *isc_blob_filter_put_segment* case for performing the filtering and then passing the data on for writing to the database. This can be done by calling the **ctl_source* internal InterBase Blob access routine. For more information about *ctl_source*, see the *Programmer's Guide*.

On the other hand, when an application calls the **isc_get_segment()** API function, the buffer pointed to by *ctl_buffer* in the control structure passed to a filter function is empty. In this situation, InterBase passes an *isc_blob_filter_get_segment* action to the filter function. The filter function *isc_blob_filter_get_segment* action handling should include instructions for filling *ctl_buffer* with segment data from the database to return to the application. This can be done by calling the **ctl_source* internal InterBase Blob access routine. In this case, because the buffer is used for filter function output, it is called an OUT field.

The following table describes each of the fields in the *isc_blob_ctl* Blob control structure, and whether they are used for filter function input (IN), or output (OUT).

Field name	**Description**
*(*ctl_source)()*	Pointer to the internal InterBase Blob access routine (IN)
**ctl_source_handle*	Pointer to an instance of *isc_blob_ctl* to be passed to the internal InterBase Blob access routine (IN)
ctl_to_sub_type	Target subtype: information field provided to support multi-purpose filters that can perform more than one kind of translation; this field and the next one enable such a filter to decide which translation to perform (IN)
ctl_from_sub_type	Source subtype: information field provided to support multi-purpose filters that can perform more than one kind of translation; this field and the previous one enable such a filter to decide which translation to perform (IN)
ctl_buffer_length	For *isc_blob_filter_put_segment*, field is an IN field that contains the length of the segment data contained in *ctl_buffer* For *isc_blob_filter_get_segment*, field is an IN field set to the size of the buffer pointed at by *ctl_buffer*, which is used to store the retrieved Blob data
ctl_segment_length	Length of current segment. For *isc_blob_filter_put_segment*, field is not used For *isc_blob_filter_get_segment*, field is an OUT field set to the size of the retrieved segment (or partial segment, in the case when the buffer length *ctl_buffer_length* is less than the actual segment length)
ctl_bpb_length	Length of the Blob parameter buffer
**ctl_bpb*	Pointer to the Blob parameter buffer
**ctl_buffer*	Pointer to segment buffer. For *isc_blob_filter_put_segment*, field is an IN field that contains the segment data For *isc_blob_filter_get_segment*, field is an OUT field the filter function fills with segment data for return to the application

TABLE 7.4 *isc_blob_ctl* structure field descriptions

Field name	Description
ctl_max_segment	Length, in bytes, of the longest segment in the Blob. Initial value is 0. The filter function sets this field. This field is information only.
ctl_number_segments	Total number of segments in the Blob. Initial value is 0. The filter function sets this field. This field is information only.
ctl_total_length	Total length, in bytes, of the Blob. Initial value is 0. The filter function sets this field. This field is information only.
**ctl_status*	Pointer to InterBase status vector. (OUT)
ctl_data [8]	8-element array of application-specific data. Use this field to store resource pointers, such as memory pointers and file handles created by the *isc_blob_filter_open* handler, for example. Then, the next time the filter function is called, the resource pointers will be available for use. (IN/OUT)

TABLE 7.4 *isc_blob_ctl* structure field descriptions (*continued*)

Programming filter function actions

When an application invokes a Blob API function on a Blob to be filtered, InterBase passes a corresponding action message to the filter function by way of the *action* parameter. There are eight possible actions. The following action macro definitions are declared in the *ibase.h* file:

```
#define isc_blob_filter_open 0
#define isc_blob_filter_get_segment 1
#define isc_blob_filter_close 2
#define isc_blob_filter_create 3
#define isc_blob_filter_put_segment 4
#define isc_blob_filter_alloc 5
#define isc_blob_filter_free 6
#define isc_blob_filter_seek 7
```

The following table lists the actions, and specifies when the filter function is invoked with each particular action. Most of the actions are the result of events that occur when an application invokes a Blob API function.

Action	When filter is invoked with corresponding action
isc_blob_filter_open	Invoked when an application calls **isc_open_blob2()**
isc_blob_filter_get_segment	Invoked when an application calls **isc_get_segment()**
isc_blob_filter_close	Invoked when an application calls **isc_close_blob()**
isc_blob_filter_create	Invoked when an application calls **isc_create_blob2()**
isc_blob_filter_put_segment	Invoked when an application calls **isc_put_segment()**
isc_blob_filter_alloc	Invoked when InterBase initializes filter processing; not a result of a particular application action
isc_blob_filter_free	Invoked when InterBase ends filter processing; not a result of a particular application action
isc_blob_filter_seek	Reserved for internal filter use; not used by external filters

TABLE 7.5 Action constants

This concludes the overview of writing Blob filters. For detailed information about filters and how to program filter function actions, as well as a reference to a filter application example, see the *Programmer's Guide*.

Writing an application that requests filtering

To request filtering of Blob data as it is read from or written to a Blob, follow these steps in your application:

1. Create a Blob parameter buffer (BPB) specifying the source and target subtypes, and optionally character sets (for TEXT subtypes).
2. Call either **isc_open_blob2()** or **isc_create_blob2()** to open a Blob for read or write access, respectively. In the call, pass the BPB, whose information InterBase will use to determine which filter should be called.

▸ *Understanding the Blob parameter buffer*

A *Blob parameter buffer* (BPB) is needed whenever a filter will be used when writing to or reading from a Blob.

The BPB is a char array variable, specifically declared in an application, that contains the source and target subtypes. When data is read from or written to the Blob associated with the BPB, InterBase will automatically invoke an appropriate filter, based on the source and target subtypes specified in the BPB.

If the source and target subtypes are both 1 (TEXT), and the BPB also specifies different source and target character sets, then when data is read from or written to the Blob associated with the BPB, InterBase will automatically convert each character from the source to the target character set.

A Blob parameter buffer can be generated in one of two ways:

1. Indirectly, through API calls to create source and target descriptors and then generate the BPB from the information in the descriptors.
2. Directly by populating the BPB array with appropriate values.

If you generate a BPB via API calls, you do not need to know the format of the BPB. But if you wish to directly generate a BPB, then you must know the format.

Both approaches are described in the following sections. The format of the BPB is documented in the section about directly populating the BPB.

GENERATING A BLOB PARAMETER BUFFER USING API CALLS

To generate a BPB indirectly, use API calls to create source and target Blob descriptors, and then call **isc_blob_gen_bpb()** to generate the BPB from the information in the descriptors. Follow these steps:

1. Declare two Blob descriptors, one for the source, and the other for the target. For example,

   ```
   #include "ibase.h"
   ISC_Blob_DESC from_desc, to_desc;
   ```
2. Store appropriate information in the Blob descriptors, by calling one of the functions **isc_blob_default_desc()**, **isc_blob_lookup_desc()**, or **isc_blob_set_desc()**, or by setting the descriptor fields directly. The following example looks up the current subtype and character set information for a Blob column named GUIDEBOOK in a table named TOURISM, and stores it into the source descriptor, *from_desc*. It then sets the target descriptor, *to_desc* to the default subtype (TEXT) and character set, so that the source data will be converted to plain text.

```
isc_blob_lookup_desc (
```

```
    status_vector,
    &db_handle;  /* set in previous isc_attach_database() call */
    &tr_handle,  /* set in previous isc_start_transaction() call */
    “TOURISM",   /* table name */
    "GUIDEBOOK", /* column name */
    &from_desc,  /* Blob descriptor filled in by this function call */
    &global);
if (status_vector[0] == 1 && status_vector[1])
{
    /* process error */
    isc_print_status(status_vector);
    return(1);
};
isc_blob_default_desc (
    &to_desc,    /* Blob descriptor filled in by this function call */
    "",          /* NULL table name; it's not needed in this case */
    "");         /* NULL column name; it's not needed in this case */
```

For more information about Blob descriptors, see **“Blob descriptors” on page 131**.

3. Declare a character array which will be used as the BPB. Make sure it is at least as large as all the information that will be stored in the buffer.

```
char bpb[20];
```

4. Declare an unsigned short variable into which InterBase will store the actual length of the BPB data:

```
unsigned short actual_bpb_length;
```

5. Call **isc_blob_gen_bpb()** to populate the BPB based on the source and target Blob descriptors passed to **isc_blob_gen_bpb()**. For example,

```
isc_blob_gen_bpb(
    status_vector,
    &to_desc,    /* target Blob descriptor */
    &from_desc,  /* source Blob descriptor */
    sizeof(bpb), /* length of BPB buffer */
    bpb,         /* buffer into which the generated BPB will be stored
*/
    &actual_bpb_length /* actual length of generated BPB */
    );
```

GENERATING A BLOB PARAMETER BUFFER DIRECTLY

It is possible to generate a BPB directly.

A BPB consists of the following parts:

1. A byte specifying the version of the parameter buffer, always the compile-time constant*, isc_bpb_version1*.
2. A contiguous series of one or more *clusters* of bytes, each describing a single parameter.

Each cluster consists of the following parts:

1. A one-byte parameter type. There are compile-time constants defined for all the parameter types (for example, *isc_bpb_target_type*).
2. A one-byte number specifying the number of bytes that follow in the remainder of the cluster.
3. A variable number of bytes, whose interpretation depends on the parameter type.

Note All numbers in the Blob parameter buffer must be represented in a generic format, with the least significant byte first, and the most significant byte last. Signed numbers should have the sign in the last byte. The API function **isc_vax_integer()** can be used to reverse the byte order of a number. For more information about **isc_vax_integer()**, see **"isc_vax_integer()" on page 332**.

The following table lists the parameter types and their meaning:

Parameter type	Description
isc_bpb_target_type	Target subtype
isc_bpb_source_type	Source subtype
isc_bpb_target_interp	Target character set
isc_bpb_source_interp	Source character set

TABLE 7.6 Blob parameter buffer parameter types

The BPB must contain *isc_bpb_version1* at the beginning, and must contain clusters specifying the source and target subtypes. Character set clusters are optional. If the source and target subtypes are both 1 (TEXT), and the BPB also specifies different source and target character sets, then when data is read from or written to the Blob associated with the BPB, InterBase will automatically convert each character from the source to the target character set.

The following is an example of directly creating a BPB for a filter whose source subtype is –4 and target subtype is 1 (TEXT):

```
char bpb[] = {
    isc_bpb_version1,
    isc_bpb_target_type,
    1,     /* # bytes that follow which specify target subtype */
    1,     /* target subtype (TEXT) */
    isc_bpb_source_type,
    1,     /* # bytes that follow which specify source subtype */
    -4,    /* source subtype*/
    };
```

Of course, if you do not know the source and target subtypes until run time, you can assign those values in the appropriate BPB locations at run time.

Requesting filter usage

You request usage of a filter when opening or creating a Blob for read or write access. In the call to **isc_open_blob2()** or **isc_create_blob2()**, pass the BPB, whose information InterBase will use to determine which filter should be called.

The following example illustrates creating and opening a Blob for write access. For further information about writing data to a Blob and updating a Blob column of a table row to refer to the new Blob, see **"Writing data to a Blob" on page 122**.

Opening a Blob for read access requires additional steps to select the appropriate Blob to be opened. For more information, see **"Reading data from a Blob" on page 117**.

```
isc_blob_handle blob_handle; /* declare at beginning */
ISC_QUAD blob_id; /* declare at beginning */
. . .
isc_create_blob2(
    status_vector,
    &db_handle,
    &tr_handle,
    &blob_handle,    /* to be filled in by this function */
    &blob_id,        /* to be filled in by this function */
    actual_bpb_length, /* length of BPB data */
    &bpb             /* Blob parameter buffer */
    )
if (status_vector[0] == 1 && status_vector[1])
{
    isc_print_status(status_vector);
    return(1);
}
```

CHAPTER

8

Working with Array Data

This chapter describes arrays of datatypes and how to work with them using API functions. It shows how to set up an array descriptor specifying the array or array subset to be retrieved or written to, and how to use the two API functions that control access to arrays.

The following table summarizes the API functions for working with arrays. First the functions that can be used to populate an array descriptor are listed, followed by the functions for accessing array data.

Function	Purpose
isc_array_lookup_desc()	Looks up and stores into an array descriptor the datatype, length, scale, and dimensions for all elements in the specified array column of the specified table
isc_array_lookup_bounds()	Performs the same actions as the function, **isc_array_lookup_desc()**, but also looks up and stores the upper and lower bounds of each dimension
isc_array_set_desc()	Initializes an array descriptor from parameters passed to it
isc_array_get_slice()	Retrieves data from an array
isc_array_put_slice()	Writes data to an array

TABLE 8.1 API array access functions

Introduction to arrays

InterBase supports arrays of most datatypes. Using an array enables multiple data items to be stored in a single column. InterBase can treat an array as a single unit, or as a series of separate units, called *slices*. Using an array is appropriate when:

- The data items naturally form a set of the same datatype.
- The entire set of data items in a single database column must be represented and controlled as a unit, as opposed to storing each item in a separate column.
- Each item must also be identified and accessed individually.

The data items in an array are called *array elements*. An array can contain elements of any InterBase datatype except Blob, and cannot be an array of arrays. All of the elements of a particular array are of the same datatype.

InterBase supports multi-dimensional arrays, arrays with 1 to 16 dimensions. Multi-dimensional arrays are stored in row-major order.

Array dimensions have a specific range of upper and lower boundaries, called *subscripts*. The array subscripts are defined when an array column is created. For information about creating an array, see the *Language Reference*.

Array database storage

InterBase does not store array data directly in the array field of a table record. Instead, it stores an *array ID* there. The array ID is a unique numeric value that references the array data, which is stored elsewhere in the database.

Array descriptors

An *array descriptor* describes an array or array subset to be retrieved or written to the ISC_ARRAY_DESC structure. ISC_ARRAY_DESC is defined in the InterBase *ibase.h* header file as follows:

```
typedef struct {
   unsigned char array_desc_dtype; /* Datatype */
   char array_desc_scale; /* Scale for numeric datatypes */
   unsigned short array_desc_length;
   /* Length in bytes of each array element */
   char array_desc_field_name [32]; /* Column name */
   char array_desc_relation_name [32]; /* Table name */
   short array_desc_dimensions; /* Number of array dimensions */
   short array_desc_flags;
   /* Specifies whether array is to be accessed in row-major or
         column-major order */
   ISC_ARRAY_BOUND array_desc_bounds [16];
      /* Lower and upper bounds for each dimension */
} ISC_ARRAY_DESC;
```

ISC_ARRAY_BOUND is defined as:

```
typedef struct {
   short array_bound_lower; /* lower bound */
   short array_bound_upper; /* upper bound */
} ISC_ARRAY_BOUND;
```

An array descriptor contains 16 ISC_ARRAY_BOUND structures, one for each possible dimension. An array with *n* dimensions has upper and lower bounds set for the first *n* ISC_ARRAY_BOUND structures. The number of actual array dimensions is specified in the **array_desc_dimensions** field of the array descriptor.

When you retrieve data from an array, you supply an array descriptor defining the array *slice* (entire array or subset of contiguous array elements) to be retrieved. Similarly, when you write data to an array, you supply an array descriptor defining the array slice to be written to.

Populating an array descriptor

There are four ways to populate an array descriptor:

- Call **isc_array_lookup_desc()**, which looks up (in the system metadata tables) and stores in an array descriptor the datatype, length, scale, and dimensions for a specified array column in a specified table. This function also stores the table and column name in the descriptor, and initializes its **array_desc_flags** field to indicate that the array is to be accessed in row-major order. For example,

```
isc_array_lookup_desc(
    status_vector,
    &db_handle,      /* Set by isc_attach_database() */
    &tr_handle,      /* Set by isc_start_transaction() */
    "PROJ_DEPT_BUDGET",/* table name */
    "QUART_HEAD_CNT",/* array column name */
    &desc            /* descriptor to be filled in */
    );
```

- Call **isc_array_lookup_bounds()**, which looks and functions the same as a call to **isc_array_lookup_desc()**, except that the function **isc_array_lookup_bounds()** also looks up and stores into the array descriptor the upper and lower bounds of each dimension.
- Call **isc_array_set_desc()**, which initializes the descriptor from parameters, rather than by accessing the database metadata. For example,

```
short dtype = SQL_TEXT;
short len = 8;
short numdims = 2;
isc_array_set_desc(
    status_vector,
    "TABLE1",        /* table name */
    "CHAR_ARRAY",    /* array column name */
    &dtype,          /* datatype of elements */
    &len,            /* length of each element */
    &numdims,        /* number of array dimensions */
    &desc            /* descriptor to be filled in */
    );
```

- Setting the descriptor fields directly. An example of setting the *array_desc_dimensions* field of the descriptor, **desc,** is:

```
desc.array_desc_dimensions = 2;
```

For complete syntax and information about **isc_array_lookup_bounds()**, **isc_array_lookup_desc()**, and **isc_array_set_desc()**, see **Chapter 12, "API Function Reference."**

Accessing array data

InterBase supports the following operations on array data:

- Reading from an array or array slice.
- Writing to an array:
  - Including a new array in a row to be inserted into a table.
  - Replacing the array referenced by an array column of a row with a new array.
  - Updating the array referenced by an array column of a row by modifying the array data or a slice of the data.
- Deleting an array.

Dynamic SQL (DSQL) API functions and the XSQLDA data structure are needed to execute SELECT, INSERT, and UPDATE statements required to select, insert, or update relevant array data. The following sections include descriptions of the DSQL programming methods required to execute the sample statements provided.

For more information about DSQL and the XSQLDA, see **Chapter 6, "Working with Dynamic SQL."**

Note The following array operations are *not* supported:

- Referencing array dimensions dynamically in DSQL.
- Setting individual array elements to NULL.
- Using aggregate functions, such as MIN() and MAX(), with arrays.
- Referencing arrays in the GROUP BY clause of a SELECT.
- Creating views that select from array slices.

Reading data from an array

There are seven steps required for reading data from an array or slice of an array:

1. Create a SELECT statement that specifies selection of the array column (and any other columns desired) in the rows of interest.
2. Prepare an output XSQLDA structure to hold the column data for each row that is fetched.
3. Prepare the SELECT statement for execution.
4. Execute the statement.
5. Populate an array descriptor with information describing the array or array slice to be retrieved.
6. Fetch the selected rows one by one.
7. Read and process the array data from each row.

Creating the SELECT *statement*

Elicit a statement string from the user or create one that consists of the SQL query that will select rows containing the array data of interest. In your query, specify the array column name and the names of any other columns containing data you are interested in. For example, the following creates an SQL query statement string that selects an array column named QUART_HEAD_CNT and another column named DEPT_NO from the table, PROJ_DEPT_BUDGET:

```
char *sel_str =
    "SELECT DEPT_NO, QUART_HEAD_CNT FROM PROJ_DEPT_BUDGET \
    WHERE year = 1994 AND PROJ_ID = 'VBASE'";
```

Preparing the output XSQLDA

Most queries return one or more rows of data, referred to as a select-list. An output XSQLDA must be created to store the column data for each row that is fetched. For an array column, the column data is an internal array identifier (*array ID*) that is needed to access the actual data. To prepare the XSQLDA, follow these steps:

1. Declare a variable to hold the XSQLDA. For example, the following declaration creates an XSQLDA called **out_sqlda**:

   ```
   XSQLDA *out_sqlda;
   ```

2. Allocate memory for the XSQLDA using the XSQLDA_LENGTH macro. The XSQLDA must contain one XSQLVAR substructure for each column to be fetched. The following statement allocates storage for an output XSQLDA (**out_sqlda**) with two XSQLVAR substructures:

   ```
   out_sqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(2));
   ```

3. Set the **version** field of the XSQLDA to SQLDA_VERSION1, and set the **sqln** field of the XSQLDA to indicate the number of XSQLVAR substructures allocated:

   ```
   out_sqlda->version = SQLDA_VERSION1;
   out_sqlda->sqln = 2;
   ```

Preparing the SELECT *statement for execution*

After an XSQLDA is created for holding the column data for each selected row, the query statement string can be prepared for execution. Follow these steps:

1. Declare and initialize an SQL statement handle, then allocate it with **isc_dsql_allocate_statement()**:

   ```
   isc_stmt_handle stmt; /* Declare a statement handle. */
   stmt = NULL;          /* Set handle to NULL before allocation. */
   isc_dsql_allocate_statement(status_vector, &db_handle, &stmt);
   ```

2. Ready the statement string for execution with **isc_dsql_prepare()**. This checks the string (*sel_str*) for syntax errors, parses it into a format that can be efficiently executed, and sets the statement handle (**stmt**) to refer to this parsed format. The statement handle is used in a later call to **isc_dsql_execute()**.

   If **isc_dsql_prepare()** is passed a pointer to the output XSQLDA, as in the following example, it will fill in most fields of the XSQLDA and all its XSQLVAR substructures with information such as the datatype, length, and name of the corresponding columns in the statement.

   A sample call to **isc_dsql_prepare()** is:

   ```
   isc_dsql_prepare(
       status_vector,
       &trans,   /* Set by previous isc_start_transaction() call. */
       &stmt,    /* Statement handle set by this function call. */
       0,        /* Specifies statement string is null-terminated. */
       sel_str,  /* Statement string. */
       1,        /* XSQLDA version number. */
       out_sqlda    /* XSQLDA for storing column data. */
       );
   ```

3. Set up an XSQLVAR structure for each column. Setting up an XSQLVAR structure involves the following steps:

   For columns whose types are known at compile time:

   - Specify the column’s datatype (if it was not set by **isc_dsql_prepare()**, as previously described).

- Point the **sqldata** field of the XSQLVAR to an appropriate local variable.

For columns whose types are not known until run time:

- Coerce the item's datatype (optional); for example, from SQL_VARYING to SQL_TEXT.
- Dynamically allocate local storage for the data pointed to by the *sqldata* field of the XSQLVAR.

For both:

Provide a NULL value indicator for the parameter.

· Data retrieval for array (and Blob) columns is different from other types of columns, so the XSQLVAR fields must be set differently. For non-array (and non-Blob) columns, **isc_dsql_prepare()** sets each XSQLVAR *sqltype* field to the appropriate field type, and the data retrieved when a select list row's data is fetched is placed into the *sqldata* space allocated for the column. For array columns, the type is set to SQL_ARRAY (or SQL_ARRAY + 1 if the array column is allowed to be NULL). InterBase stores the internal array identifier (array ID), not the array data, in the *sqldata* space when a row's data is fetched, so you must point *sqldata* to an area the size of an array ID. To see how to retrieve the actual array or array slice data once you have an array ID, see **"Reading and processing the array data" on page 154**.

· The following code example illustrates the assignments for array and non-array columns whose types are known at compile time. For more information about DSQL and the XSQLDA, and working with columns whose types are unknown until run time, see **Chapter 6, "Working with Dynamic SQL."**

```
ISC_QUAD array_id = 0L;
char dept_no[6];
short flag0, flag1;
out_sqlda->sqlvar[0].sqldata = (char *) dept_no;
out_sqlda->sqlvar[0].sqltype = SQL_TEXT + 1;
out_sqlda->sqlvar[0].sqlind = &flag0;
out_sqlda->sqlvar[1].sqldata = (char *) &array_id;
out_sqlda->sqlvar[1].sqltype = SQL_ARRAY + 1;
out_sqlda->sqlvar[1].sqlind = &flag1;
```

Executing the statement

Once the query statement string is prepared, it can be executed:

```
isc_dsql_execute(
   status_vector,
   &trans,   /* set by previous isc_start_transaction() call */
   &stmt,    /* set above by isc_dsql_prepare() */
   1,        /* XSQLDA version number */
   NULL      /* NULL since stmt doesn’t have input values */
   );
```

This statement creates a select-list, the rows returned by execution of the statement.

Populating the array descriptor

To prepare an array descriptor that describes the array or array slice to be read, follow these steps:

1. Create the array descriptor:

   ```
   ISC_ARRAY_DESC desc;
   ```

2. Fill in the descriptor with information regarding the array column from which data will be read. Do this either by calling one of the functions **isc_array_lookup_bounds()**, **isc_array_lookup_desc()**, or **isc_array_set_desc()**, or by directly filling in the descriptor. For information on the contents of array descriptors, and the usage of these functions, see **"Array descriptors" on page 147**.

   Ensure the descriptor boundaries are set to those of the slice to be read.

   If you want to retrieve all the array data (that is, not just a smaller slice), set the boundaries to the full boundaries as initially declared for the array column. This is guaranteed to be the case if you fill in the descriptor by calling **isc_array_lookup_bounds()**, as in:

   ```
   ISC_ARRAY_DESC desc;
   isc_array_lookup_bounds(
       status_vector,
       &db_handle,
       &trans,
       "PROJ_DEPT_BUDGET",/* table name */
       "QUART_HEAD_CNT",/* array column name */
       &desc);
   ```

   Suppose the array column, QUART_HEAD_CNT, is a one-dimensional array consisting of four elements, and it was declared to have a lower subscript bound of 1 and an upper bound of 4 when it was created. Then after the above call to **isc_array_lookup_bounds()**, the array descriptor fields for the boundaries contain the following information:

   ```
   desc.array_desc_bounds[0].array_bound_lower == 1
   desc.array_desc_bounds[0].array_bound_upper == 4
   ```

If you want to read just a slice of the array, then modify the upper and/or lower bounds appropriately. For example, if you just want to read the first two elements of the array, then modify the upper bound to the value 2, as in:

```
desc.array_desc_bounds[0].array_bound_upper = 2
```

Fetching selected rows

A looping construct is used to fetch (into the output XSQLDA) the column data for a single row at a time from the select-list and to process each row before the next row is fetched. Each execution of **isc_dsql_fetch()** fetches the column data for the next row into the corresponding XSQLVAR structures of *out_sqlda*. For the array column, the array ID, not the actual array data, is fetched.

```
ISC_STATUS fetch_stat;
long SQLCODE;
. . .
while ((fetch_stat = j
          isc_dsql_fetch(status_vector, &stmt, 1, out_sqlda))
      == 0)
{
   /* Read and process the array data */
}
if (fetch_stat != 100L)
{
   /* isc_dsql_fetch returns 100 if no more rows remain to be
      retrieved */
   SQLCODE = isc_sqlcode(status_vector);
   isc_print_sqlerror(SQLCODE, status_vector);
   return(1);
}
```

Reading and processing the array data

To read and process the array or array slice data:

1. Create a buffer for holding the array data to be read. Make it large enough to hold all the elements in the slice to be read (which could be the entire array). For example, the following declares an array buffer large enough to hold 4 long elements:

   ```
   long hcnt[4];
   ```

2. Declare a short variable for specifying the size of the array buffer:

   ```
   short len;
   ```

3. Set the variable to the buffer length:

   ```
   len = sizeof(hcnt);
   ```

4. Read the array or array slice data into the buffer by calling **isc_array_get_slice()**. Process the data read. In the following example, the array is read into the *hcnt* array buffer, and "processing" consists of printing the data:

   ```
   isc_array_get_slice(
      status_vector,
      &db_handle,/* set by isc_attach_database()*/
      &trans,   /* set by isc_start_transaction() */
      &array_id, /* array ID put into out_sqlda by isc_dsql_fetch()*/
      &desc,    /* array descriptor specifying slice to be read */
      (void *) hcnt,/* buffer into which data will be read */
      (long *) &len/* length of buffer */
      );
   if (status_vector[0] == 1 && status_vector[1])
   {
      isc_print_status(status_vector);
      return(1);
   }
   /* Make dept_no a null-terminated string */
   dept_no[out_sqlda->sqlvar[0].sqllen] = '\0';
   printf("Department #: %s\n\n", dept_no);
   printf("\tCurrent head counts: %ld %ld %ld %ld\n",
      hcnt[0], hcnt[1], hcnt[2], hcnt[3]);
   ```

Writing data to an array

isc_array_put_slice() is called to write data to an array or array slice. Use it to:

- Include a new array in a row to be inserted into a table.
- Replace the current contents of an array column of a row with a new array.
- Update the array referenced by an array column of a row by modifying the array data or a slice of the data.

The entry in an array column of a row does not actually contain array data. Rather, it has an array ID referring to the data, which is stored elsewhere. So, to set or modify an array column, you need to set or change the array ID stored in it. If an array column contains an array ID, and you modify the column to refer to a different array (or to contain NULL), the array referenced by the previously stored array ID will be deleted during the next garbage collection.

The following steps are required to insert, replace, or update array data:

1. Prepare an array descriptor with information describing the array (or slice) to be written to.
2. Prepare an array buffer with the data to be written.
3. Prepare an appropriate DSQL statement. This will be an INSERT statement if you are inserting a new row into a table, or an UPDATE statement for modifying an existing row.
4. Call **isc_array_put_slice()** to create a new array (possibly copying an existing one), and to write the data from the array buffer into the array or array slice.
5. Associate the new array with an array column of the table row being modified or inserted by executing the UPDATE or INSERT statement. This sets the array column to contain the array ID of the new array.

Preparing the array descriptor

To prepare an array descriptor that specifies the array or array slice to be written to, follow these steps:

1. Create the array descriptor:

   ```
   ISC_ARRAY_DESC desc;
   ```

2. Fill in the descriptor with information regarding the array column to which data will be written. Do this either by calling one of the functions **isc_array_lookup_bounds()**, **isc_array_lookup_desc()**, or **isc_array_set_desc()**, or by directly filling in the descriptor. For information on the contents of array descriptors, and the usage of these functions, see **"Array descriptors" on page 147**.

   Ensure the descriptor boundaries are set to those of the slice to be written to.

   If you want to write to the entire array (i.e., not just a smaller slice), set the boundaries to the full boundaries as initially declared for the array column. This is guaranteed to be the case if you fill in the descriptor by calling **isc_array_lookup_bounds()**, as in:

   ```
   isc_array_lookup_bounds(
       status_vector,
   ```

```
    db_handle,
    &trans,
    "PROJ_DEPT_BUDGET",/* table name */
    "QUART_HEAD_CNT",/* array column name */
    &desc);
```

Suppose the array column, QUART_HEAD_CNT, is a one-dimensional array consisting of four elements, and it was declared to have a lower subscript bound of 1 and an upper bound of 4 when it was created. Then after a call to **isc_array_lookup_bounds()**, the array descriptor fields for the boundaries contain the following information:

```
desc.array_desc_bounds[0].array_bound_lower == 1
desc.array_desc_bounds[0].array_bound_upper == 4
```

If you just want to write to (or modify) a slice of the array, then change the upper and lower bound appropriately. For example, if you just want to write to the first two elements of the array, then modify the upper bound to the value 2, as in:

```
desc.array_desc_bounds[0].array_bound_upper == 2
```

Preparing the array buffer with data

Create an array buffer to hold the data to be written to the array. Make it large enough to hold all the elements in the slice to be written (which could be the entire array). For example, the following declares an array buffer large enough to hold 4 long elements:

```
long hcnt[4];
```

1. Create a variable specifying the length of the array buffer:

   ```
   short len;
   len = sizeof(hcnt);
   ```

2. Fill the array buffer with the data to be written.

   If you are creating a new array, then fill the buffer with data. For example,

   ```
   hcnt[0] = 4;
   hcnt[1] = 5;
   hcnt[2] = 6;
   hcnt[3] = 6;
   ```

   To modify existing array data instead of creating a new one, then perform all the steps listed in **"Reading data from an array" on page 149** to read the existing array data into the array buffer. Modify the data in the buffer.

Preparing the UPDATE *or* INSERT *statement*

To prepare an UPDATE or INSERT statement for execution, follow these steps:

1. Elicit an UPDATE or INSERT statement string from the user or create one for inserting a new row or updating the row(s) containing the array column(s) of interest. For example, the following statement is for updating the array column named QUART_HEAD_CNT in the specified row of the table, PROJ_DEPT_BUDGET. The department number and quarterly headcounts are assumed to be supplied at run time:

   ```
   char *upd_str =
      "UPDATE PROJ_DEPT_BUDGET SET QUART_HEAD_CNT = ? WHERE \
      YEAR = 1994 AND PROJ_ID = "MKTPR" AND DEPT_NO = ?";
   ```

   As an example of an INSERT statement, the following is for inserting a new row into the PROJ_DEPT_BUDGET table, with column data supplied at run time:

   ```
   char *upd_str =
      "INSERT INTO PROJ_DEPT_BUDGET (YEAR, PROJ_ID, DEPT_NO, \
      QUART_HEAD_CNT) VALUES (?, ?, ?, ?)";
   ```

   The remaining steps refer only to UPDATE statements, but the actions apply to INSERT statements as well.

2. Declare a variable to hold the input XSQLDA needed to supply parameter values to the UPDATE statement at run time. For example, the following declaration creates an XSQLDA called *in_sqlda*:

   ```
   XSQLDA *in_sqlda;
   ```

3. Allocate memory for the input XSQLDA using the XSQLDA_LENGTH macro. The XSQLDA must contain one XSQLVAR substructure for each parameter to be passed to the UPDATE statement. The following statement allocates storage for an input XSQLDA (*in_sqlda*) with two XSQLVAR substructures:

   ```
   in_sqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(2));
   ```

4. Set the Version field of the XSQLDA to SQLDA_VERSION1, and set the Sqln field to indicate the number of XSQLVAR structures allocated:

   ```
   in_sqlda->version = SQLDA_VERSION1;
   in_sqlda->sqln = 2;
   ```

5. Set up the XSQLVAR structure in the XSQLDA for each parameter to be passed. Setting up an XSQLVAR structure involves the following steps:

   - Specify the item's datatype.
   - For parameters whose types are known at compile time, point the Ssqldata field of the XSQLVAR to an appropriate local variable that will contain the data to be passed.

- For parameters whose types are not known until run time, allocate local storage for the data pointed to by the Sqldata field of the XSQLVAR.
- Specify the number of bytes of data.

Data storage for array (and Blob) columns is different from other types of columns, so the XSQLVAR fields must be set differently. For non-array (and non-Blob) columns, input parameter data comes from the space pointed to by Sqldata. For array columns, set the type to SQL_ARRAY (or SQL_ARRAY + 1 if the array column is allowed to be NULL). The application must store space for the internal array identifier, not the array data, in the Sqldata space. See the following sections to create or modify an array, store its array ID in the Sqldata space, and then associate the actual array data with the column.

The following code example illustrates the assignments for one TEXT column and one array column, where the column types are known at compile time.

```
#define NUMLEN 4
char dept_no[NUMLEN + 1];
ISC_QUAD array_id;
in_sqlda->sqlvar[0].sqldata = &array_id;
in_sqlda->sqlvar[0].sqltype = SQL_ARRAY + 1;
in_sqlda->sqlvar[0].sqllen = sizeof(ISC_QUAD);
in_sqlda->sqlvar[1].sqldata = dept_no;
in_sqlda->sqlvar[1].sqltype = SQL_TEXT;
in_sqlda->sqlvar[1].sqllen = 4;
```

The *dept_no* variable should be assigned a value at run time (unless the value is known at compile time). The *array_id* variable should be set to refer to the newly created array, as described in the following sections.

For examples of handling data whose types are not known until run time, see **Chapter 6, "Working with Dynamic SQL."**

Calling isc_array_put_slice()

The following steps are required to store the data into an array or array slice:

1. Declare an array ID:

```
ISC_QUAD array_id;  /* Declare an array ID. */
```

2. Initialize the array ID. If you are creating a new array to be inserted into a new row, or to replace an existing array, then simply initialize the array ID to NULL:

```
array_id = NULL;/* Set handle to NULL before using it */
```

If you are modifying an existing array, then follow the steps listed under "Reading Data from an Array" to read the existing array ID into *array_id*.

3. Call **isc_array_put_slice()**. In your call you pass the array ID (either the array ID of an existing array, or NULL for a new array) in the *array_id* variable. You also pass the buffer of data to be written and a descriptor specifying the array slice to which the data belongs.

   When **isc_array_put_slice()** is called with an array ID of an existing array, it creates a new array with the same characteristics as the specified array, and copies the existing array data to the new array. Then **isc_array_put_slice()** writes the data from the array buffer to the new array (or slice of the array), per the bounds specified in the array descriptor, and returns in the same *array_id* variable the array ID of the new array.

   When **isc_array_put_slice()** is called with a NULL array ID, it creates a new empty array with characteristics as declared for the array column whose name and table name are specified in the array descriptor passed to **isc_array_put_slice()**. It then writes the data from the array buffer to the new array (or slice of the array), and returns in the *array_id* variable the array ID of the new array.

   Note that in both cases, a new array is created, and its array ID is returned in the *array_id* variable. The array is temporary until an UPDATE or INSERT statement is executed to associate the array with a particular column of a particular row.

   You can make a single call to **isc_array_put_slice()** to write all the data to the array. Or, you may call **isc_array_put_slice()** multiple times to store data into various slices of the array. In this case, each call to **isc_array_put_slice()** after the first call should pass the array ID of the temporary array. When **isc_array_put_slice()** is called with the array ID of a temporary array, it copies the specified data to the specified slice of the temporary array, but does not create a new array.

   The following is a sample call to **isc_array_put_slice()**:

```
isc_array_put_slice(
    status_vector,
    &db_handle,
    &trans,
    &array_id,/* array ID (NULL, or existing array’s array ID) */
    &desc,    /* array descriptor describing where to write data */
    hcnt,     /* array buffer containing data to write to array */
    &len      /* length of array buffer */
    );
```

   This call creates a new array, copies the data in *hcnt* to the new array (or slice of the array), assigns the array an array ID, and sets *array_id* to point to the array ID.

IMPORTANT *array_id* should be the variable pointed to by the Sqldata field of the UPDATE (or INSERT) statement input parameter that specifies the array column to be updated. Thus, when the INSERT or UPDATE statement is executed, this new array's array ID will be used to set or update the array column to refer to the new array.

Associating the new array with the array column

Execute the UPDATE statement to associate the new array with the array column in the row selected by the statement:

```
isc_dsql_execute_immediate(
   status_vector,
   &db_handle,
   &trans,
   0,        /* indicates string to execute is null-terminated */
   upd_str,  /* UPDATE statement string to be executed */
   1,        /* XSQLDA version number */
   in_sqlda  /* XSQLDA supplying parameters to UPDATE statement */
   );
```

This sets the array column in the row specified in the UPDATE statement to contain the array ID of the new array. The array ID comes from the *array_id* variable pointed to by the *in_sqlda* parameter corresponding to the array column.

If the array column in the specified row contains the array ID of a different array before the UPDATE statement is executed, then the column is modified to contain the new array ID, and the array referenced by the previously stored array ID will be deleted during the next garbage collection.

Deleting an array

There are three ways to delete an array:

1. Delete the row containing the array. You can use DSQL to execute a DELETE statement.
2. Replace the array with a different one, as described above. If an array column contains an array ID, and you modify the column to refer to a different array, the array referenced by the previously stored array ID will be deleted during the next garbage collection.
3. Reset to NULL the column referring to the array. For example, use DSQL to execute a statement like the following, where *LANGUAGE_REQ* is an array column:

```
"UPDATE JOB SET LANGUAGE_REQ = NULL \
WHERE JOB_CODE = "SA12" AND JOB_GRADE = 10"
```

The array referenced by the previously stored array ID will be deleted during the next garbage collection.

CHAPTER

9

Working with Conversions

InterBase uses a proprietary format for internal storage of DATE data, but provides the following API calls for translating to and from this format:

- **isc_decode_date()** to convert the InterBase internal date format to the C time structure
- **isc_encode_date()** to convert the C time structure to the internal InterBase date format

These calls merely translate DATE data between formats; they do not read or write DATE data directly. DATE data is read from and written to the database using standard DSQL syntax processed with the isc_dsql family of API calls.

InterBase also requires that numbers entered in database and transaction parameter buffers be in a generic format, with the least significant byte last. Signed numbers require the sign to be in the last byte. Systems that represent numbers with the most significant byte last must use the **isc_vax_integer()** API function to reverse the byte order of numbers entered in database parameter buffers (DPBs) and transaction parameter buffers (TPBs). When numeric information is returned by information calls on these systems, **isc_vax_integer()** must be used once again to reverse the byte ordering.

For more information about using DSQL to read and write data, see **Chapter 6, "Working with Dynamic SQL."**

Converting dates from InterBase to C format

To select a date from a table, and convert it to a form usable in a C language program, follow these steps:

1. Create a host variable for a C time structure. Most C and C++ compilers provide a typedef declaration, *tm*, for the C time structure in the *time.h* header file. The following C code includes that header file, and declares a variable of type *tm*:

   ```
   #include <time.h>
   #include "ibase.h"
   . . .
   struct tm hire_time;
   . . .
   ```

Note To create host-language time structures in languages other than C and C++, see the host-language reference manual.

2. Create a host variable of type ISC_QUAD. For example, the host-variable declaration might look like this:

   ```
   ISC_QUAD hire_date;
   ```

   The ISC_QUAD structure is declared in *ibase.h*, but the programmer must declare actual host-language variables of type ISC_QUAD.

3. Retrieve a date from a table into the ISC_QUAD variable.
4. Convert the ISC_QUAD variable into a numeric C format with the InterBase function, **isc_decode_date()**. This function is also declared in *ibase.h*. **isc_decode_date()** requires two parameters, the address of the ISC_QUAD host-language variable, and the address of the *tm* host-language variable. For example, the following code fragment coverts *hire_date* to *hire_time*:

   ```
   isc_decode_date(&hire_date, &hire_time);
   ```

Converting dates from C to InterBase format

To insert a date in a table, it must be converted from the host-language format into InterBase format, and then stored. To perform the conversion and insertion in a C program, follow these steps:

1. Create a host variable for a C time structure. Most C and C++ compilers provide a typedef declaration, *tm*, for the C time structure in the *time.h* header file. The following C code includes that header file, and declares a *tm* variable, *hire_time*:

   ```
   #include <time.h>;
   . . .
   struct tm hire_time;
   . . .
   ```

   To create host-language time structures in languages other than C and C++, see the host-language reference manual.
2. Create a host variable of type ISC_QUAD, for use by InterBase. For example, the host-variable declaration might look like this:

   ```
   ISC_QUAD mydate;
   ```

   The ISC_QUAD structure is declared in *ibase.h*, but the programmer must declare actual host-language variables of type ISC_QUAD.
3. Put date and time information into *hire_time.*
4. Use the InterBase **isc_encode_date()** function to convert the information in *hire_time* into InterBase internal format and store that formatted information in the ISC_QUAD host variable (*hire_date* in the example). This function is also declared in *ibase.h*.

   isc_encode_date() requires two parameters, the address of the C time structure, and the address of the ISC_QUAD host-language variable. For example, the following code converts *hire_time* to *hire_date*:

   ```
   isc_encode_date(&hire_time, &hire_date);
   ```
5. Insert the date into a table.

Reversing byte order of numbers with isc_vax_integer( )

InterBase expects that numbers entered in database and transaction parameter buffers be in a generic format, with the least significant byte last. Signed numbers require the sign to be in the last byte. Systems that represent numbers with the most significant byte last must use the **isc_vax_integer()** API function to reverse the byte order of numbers entered in DPBs and TPBs. When numeric information is returned by information calls on these systems, **isc_vax_integer()** must be used once again to reverse the byte ordering. The syntax for **isc_vax_integer()** is:

```
ISC_LONG isc_vax_integer( char *buffer, short length);
```

buffer is a char pointer to the integer to convert, and *length* is the size, in bytes, of the integer. Valid lengths are 1 (short), 2 (int), and 4(long). The following code reverses the 4-byte value in a result buffer.

```
#include "ibase.h"
. . .
for(p = res_buffer; *p != isc_info_end;)
{
   p++;
   length = isc_vax_integer(p, 2);
}
```

CHAPTER

10 Handling Error Conditions

This chapter describes how to set up an error status vector where InterBase can store run-time error information, and how to use API functions to handle and report errors.

The following table summarizes the API functions for handling errors:

Function	**Purpose**
isc_interprete()	Capture InterBase error messages to a buffer
isc_print_sqlerror()	Display an SQL error message
isc_print_status()	Display InterBase error messages
isc_sqlcode()	Set the value of SQLCODE
isc_sql_interprete()	Capture an SQL error message to a buffer

TABLE 10.1 Error-handling functions

Setting up an error status vector

Most API functions return status information that indicates success or failure. The information returned is derived from the second array element of the error status vector, where InterBase reports error conditions. The error status vector is declared in applications as an array of 20 long integers, using the following syntax:

```
#include <ibase.h>
. . .
ISC_STATUS status_vector[20];
```

ISC_STATUS is a #define in *ibase.h* provided for programing convenience and platform independence.

Using information in the status vector

Whether or not an error occurs during the execution of an API call, InterBase loads the error status vector with status information. Information consists of one or more InterBase error codes, and error information that can be used to build an error message honed to a specific error.

An application can check the status vector after the execution of most API calls to determine their success or failure. If an error condition is reported, applications can:

- Display InterBase error messages using **isc_print_status()**.
- Set an SQLCODE value corresponding to an InterBase error using **isc_sqlcode()**, and display the SQLCODE and an SQL error message using **isc_print_sqlerror()**.
- Build individual InterBase error messages in a buffer with **isc_interprete()**. The buffer must be provided by the application. Using a buffer enables an application to perform additional message processing (for example, storing messages in an error log file). This ability is especially useful on windowing systems that do not permit direct screen writes.
- Capture an SQL error message in a buffer with **isc_sql_interprete()**. The buffer must be provided by the application.
- Parse for and react to specific InterBase error codes in the status vector.

Checking the status vector for errors

API functions that return information in the status vector are declared in *ibase.h* as returning an ISC_STATUS pointer. For example, the function prototype for **isc_prepare_transaction()** is declared as:

```
ISC_STATUS ISC_EXPORT isc_prepare_transaction(
ISC_STATUS ISC_FAR *,
isc_tr_handle ISC_FAR *);
```

To check the status vector for error conditions after the execution of a function, examine the first element of the status vector to see if it is set to 1, and if so, examine the second element to see if it is not 0. A nonzero value in the second element indicates an error condition. The following C code fragment illustrates how to check the status vector for an error condition:

```
#include <ibase.h>
. . .
ISC_STATUS status_vector[20];
. . .
/* Assume an API call returning status information is called here. */
if (status_vector[0] == 1 && status_vector[1] > 0)
{
   /* Handle error condition here. */
   ;
}
```

If an error condition is detected, you can use API functions in an error-handling routine to display error messages, capture the error messages in a buffer, or parse the status vector for particular error codes.

Display or capture of error messages is only one part of an error-handling routine. Usually, these routines also roll back transactions, and sometimes they can retry failed operations.

Displaying InterBase error messages

Use **isc_print_status()** to display InterBase error messages on the screen. This function parses the status vector to build all available error messages, then uses the C **printf()** function to write the messages to the display. **isc_print_status()** requires one parameter, a pointer to a status vector containing error information. For example, the following code fragment calls **isc_print_status()** and rolls back a transaction on error:

```
#include <ibase.h>
. . .
ISC_STATUS status_vector[20];
isc_tr_handle trans;
. . .
trans = 0L;
. . .
/* Assume a transaction, trans, is started here. */
/* Assume an API call returning status information is called here. */
if (status_vector[0] == 1 && status_vector[1] > 0)
{
   isc_print_status(status_vector);
   isc_rollback_transaction(status_vector, &trans);
}
```

IMPORTANT On windowing systems that do not permit direct screen writes with **printf()**, use **isc_interprete()** to capture error messages to a buffer.

TIP For applications that use the dynamic SQL (DSQL) API functions, errors should be displayed using SQL conventions. Use **isc_sqlcode()** and **isc_print_sqlerror()** instead of **isc_print_status()**.

Capturing InterBase error messages

Use **isc_interprete()** to build an error message from information in the status vector and store it in an application-defined buffer where it can be further manipulated. Capturing messages in a buffer is useful when applications:

- Run under windowing systems that do not permit direct screen writes.
- Require more control over message display than is possible with **isc_print_status()**.
- Store a record of all error messages in a log file.
- Manipulate or format error messages for display or pass them to a windowing system's display routines.

isc_interprete() retrieves and formats a single error message each time it is called. When an error occurs, the status vector usually contains more than one error message. To retrieve all relevant error messages, you must make repeated calls to **isc_interprete()**.

Given both the location of a buffer, and the address of the status vector, **isc_interprete()** builds an error message from the information in the status vector, puts the formatted string in the buffer where an application can manipulate it, and advances the status vector pointer to the start of the next cluster of error information. **isc_interprete()** requires two parameters, the address of an application buffer to hold formatted message output, and a pointer to the status vector array.

IMPORTANT Never pass the status vector array directly to **isc_interprete()**. Each time it is called, **isc_interprete()** advances the pointer to the status vector to the next element containing new message information. Before calling **isc_interprete()**, be sure to set the pointer to the starting address of the status vector.

The following code demonstrates an error-handling routine that makes repeated calls to **isc_interprete()** to retrieve error messages from the status vector in a buffer, one at a time, so they can be written to a log file:

```
#include <ibase.h>
. . .
ISC_STATUS status_vector[20];
isc_tr_handle trans;
long *pvector;
char msg[512];
FILE *efile; /* Code fragment assumes pointer to an open file. */
trans = 0L;
. . .
/* Error-handling routine starts here. */
/* Always set pvector to point to start of status_vector. */
pvector = status_vector;
/* Retrieve first message. */
isc_interprete(msg, &pvector);
/* Write first message from buffer to log file. */
fprintf(efile, "%s\n", msg);
msg[0] = '-'; /* Append leading hyphen to secondary messages. */
/* Look for more messages and handle in a loop. */
while(isc_interprete(msg + 1, &pvector)) /* More? */
   fprintf(efile, "%s\n", msg); /* If so, write it to the log. */
fclose(efile); /* All done, so close the log file. */
isc_rollback(status_vector, &trans);
return(1);
. . .
```

Note This code fragment assumes that the log file is properly declared and opened elsewhere in the application *before* control is passed to this error handler.

TIP For applications that use the dynamic SQL (DSQL) API functions, errors should be buffered using SQL conventions. Use **isc_sqlcode()** and **isc_sql_interprete()** instead of **isc_interprete()**.

Setting an SQLCODE value on error

For DSQL applications, error conditions should be cast in terms of SQL conventions. SQL applications typically report errors through a variable, SQLCODE, declared by an application. To translate an InterBase error code into SQLCODE format, use **isc_sqlcode()**. This function searches the error status vector for an InterBase error code that can be translated into an SQL error code, and performs the translation. Once SQLCODE is set, the other API functions for handling SQL errors, **isc_print_sqlerror()**, and **isc_sql_interprete()**, can be called.

isc_sqlcode() requires one parameter, a pointer to the status vector. It returns a long value, containing an SQL error code. The following code illustrates the use of this function:

```
#include <ibase.h>;
. . .
long SQLCODE; /* Declare the SQL error code variable. */
ISC_STATUS status_vector[20];
. . .
if (status_vector[0] == 1 && status_vector[1] > 0)
{
   SQLCODE = isc_sqlcode(status_vector);
   isc_print_sqlerror(SQLCODE, status_vector)
   . . .
}
```

If successful, **isc_sqlcode()** returns the first valid SQL error code decoded from the status vector. If no valid SQL error code is found, **isc_sqlcode()** returns -999.

Displaying SQL error messages

API applications that provide a DSQL interface to end users should use **isc_print_sqlerror()** to display SQL error codes and corresponding error messages on the screen. When passed a variable, conventionally named *SQLCODE*, containing an SQL error code, and a pointer to the status vector, **isc_print_sqlerror()** parses the status vector to build an SQL error message, then uses the C **printf()** function to write the SQLCODE value and message to the display. For example, the following code fragment calls **isc_print_sqlerror()** and rolls back a transaction on error:

```
#include <ibase.h>
. . .
ISC_STATUS status_vector[20];
isc_tr_handle trans;
long SQLCODE;
. . .
trans = 0L;
. . .
/* Assume a transaction, trans, is started here. */
/* Assume an API call returning status information is called here. */
if (status_vector[0] == 1 && status_vector[1] > 0)
{
   SQLCODE = isc_sqlcode(status_vector);
   isc_print_sqlerror(SQLCODE, status_vector);
   isc_rollback_transaction(status_vector, &trans);
}
```

IMPORTANT On windowing systems that do not permit direct screen writes with **printf()**, use **isc_sql_interprete()** to capture error messages to a buffer.

Capturing SQL error messages

Use **isc_sql_interprete()** to build an SQL error message based on a specific SQL error code and store it in a buffer defined by an application. Capturing messages in a buffer is useful when applications:

- Run under windowing systems that do not permit direct screen writes.
- Store a record of all error messages in a log file.
- Manipulate or format error messages for display or pass them to a windowing system's display routines.

isc_sql_interprete() requires three parameters: a valid SQL error code, usually passed as a variable named *SQLCODE*, a buffer where the SQL message should be stored, and the size of the buffer. The following code illustrates how this function might be called to build a message string and store it in a log file:

```
#include <ibase.h>
. . .
ISC_STATUS status_vector[20];
isc_tr_handle trans;
long SQLCODE;
char msg[512];
```

```
FILE *efile; /* Code fragment assumes pointer to an open file. */
trans = 0L;
. . .
/* Assume a transaction, trans, is started here. */
/* Assume an API call returning status information is called here. */
. . .
/* Error-handling routine starts here. */
if (status_vector[0] == 1 && status_vector[1] > 0)
{
   SQLCODE = isc_sqlcode(status_vector);
   isc_sql_interprete(SQLCODE, msg, 512);
   fprintf(efile, "%s\n", msg);
   isc_rollback_transaction(status_vector, &trans);
   return(1);
}
```

Note This code fragment assumes that the log file is properly declared and opened elsewhere in the application *before* control is passed to this error handler.

Parsing the status vector

InterBase stores error information in the status vector in *clusters* of two or three longs. The first cluster in the status vector *always* indicates the primary cause of the error. Subsequent clusters may contain supporting information about the error, for example, strings or numbers for display in an associated error message. The actual number of clusters used to report supporting information varies from error to error.

In many cases, additional errors may be reported in the status vector. Additional errors are reported immediately following the first error and its supporting information, if any. The first cluster for each additional error message identifies the error. Subsequent clusters may contain supporting information about the error.

How the status vector is parsed

The InterBase error-handling routines, **isc_print_status()** and **isc_interprete()**, use routines which automatically parse error message information in the status vector without requiring you to know about its structure. If you plan to write your own routines to read and react to the contents of the status vector, you need to know how to interpret it.

The key to parsing the status vector is to decipher the meaning of the first long in each cluster, beginning with the first cluster in the vector.

Meaning of the first long in a cluster

The first long in any cluster is a *numeric descriptor*. By examining the numeric descriptor for any cluster, you can always determine the:

- Total number of longs in the cluster.
- Kind of information reported in the remainder of the cluster.
- Starting location of the next cluster in the status vector.

Interpretation of 1st long in a cluster

Value	Longs in cluster	Meaning
0	—	End of error information in the status vector
1	2	Second long is an InterBase error code
2	2	Second long is the address of string used as a replaceable parameter in a generic InterBase error message
3	3	Second long is the length, in bytes, of a variable-length string provided by the operating system (most often this string is a file name); third long is the address of the string
4	2	Second long is a number used as a replaceable parameter in a generic InterBase error message
5	2	Second long is the address of an error message string requiring no further processing before display
6	2	Second long is a VAX/VMS error code
7	2	Second long is a Unix error code
8	2	Second long is an Apollo Domain error code

TABLE 10.2 Interpretation of status vector clusters

Interpretation of 1st long in a cluster		
Value	**Longs in cluster**	**Meaning**
9	2	Second long is an MS-DOS or OS/2 error code.
10	2	Second long is an HP MPE/XL error code.
11	2	Second long is an HP MPE/XL IPC error code.
12	2	Second long is a NeXT/Mach error code.

Note: As InterBase is adapted to run on other hardware and software platforms, additional numeric descriptors for specific platform and operating system error codes may be added to the end of this list.

TABLE 10.2 Interpretation of status vector clusters (*continued*)

By including *ibase.h* at the start of your source code, you can use a series of #defines to substitute for hard-coded numeric descriptors in the status vector parsing routines you write. The advantages of using these #defines over hard-coding the descriptors are:

- Your code will be easier to read.
- Code maintenance will be easier should the numbering scheme for numeric descriptors change in a future release of InterBase.

The following table lists the #define equivalents of each numeric descriptor:

Value	**#define**	**Value**	**#define**
0	*isc_arg_end*	8	*isc_arg_domain*
1	*isc_arg_gds*	9	*isc_arg_dos*
2	*isc_arg_string*	10	*isc_arg_mpexl*
3	*isc_arg_cstring*	11	*isc_arg_mpexl_ipc*
4	*isc_arg_number*	15	*isc_arg_next_mach*
5	*isc_arg_interpreted*	16	*isc_arg_netware*
6	*isc_arg_vms*	17	*isc_arg_win32*
7	*isc_arg_unix*		

TABLE 10.3 #defines for status vector numeric descriptors

For an example of code that uses these defines, see **"Status vector parsing example" on page 178**.

Meaning of the second long in a cluster

The second long in a cluster is *always* one of five items:

- An InterBase error code (1st long = 1).
- A string address (1st long = 2 or 5).
- A string length (1st long = 3).
- A numeric value (1st long = 4).
- An operating system error code (1st long > 5).

INTERBASE ERROR CODES

InterBase error codes have two uses. First, they are used internally by InterBase functions to build and display descriptive error message strings. For example, **isc_interprete()** calls another function which uses the InterBase error code to retrieve a base error message from which it builds an error message string you can display or store in a log file.

Secondly, when you write your own error-handling routine, you can examine the status vector directly, trapping for and reacting to specific InterBase error codes.

When the second long of a cluster is an InterBase error code, then subsequent clusters may contain additional parameters for the error message string associated with the error code. For example, a generic InterBase error message may contain a replaceable string parameter for the name of the table where an error occurs, or it may contain a replaceable numeric parameter for the code of the trigger which trapped the error condition.

If you write your own parsing routines, you may need to examine and use these additional clusters of error information.

STRING ADDRESSES

String addresses point to error message text. When the first long in the cluster is 2 (*isc_arg_string*), the address pointed to often contains the name of the database, table, or column affected by the error. In these cases, InterBase functions which build error message strings replace a parameter in a generic InterBase error message with the string pointed to by this address. Other times the address points to an error message hard-coded in a database trigger.

When the first long in the cluster is 5 (*isc_arg_interpreted*), the address points to a text message which requires no further processing before retrieval. Sometimes this message may be hard-coded in InterBase itself, and other times it may be a system-level error message.

In either of these cases, InterBase functions such as **isc_print_status()** and **isc_interprete()** can format and display the resulting error message for you.

STRING LENGTH INDICATORS

When the first long in a cluster is 3 (*isc_arg_cstring*), the numeric value in the second long indicates the length, in bytes, of a message string whose address is stored in the third long in the cluster. This string requires translation into a standard, null-terminated C string before display.

NUMERIC VALUES

A numeric value has different meaning depending upon the value of the numeric descriptor in the first long of a cluster. If the first long is 4 (*isc_arg_number*), a numeric value is used by InterBase functions to replace numeric parameters in generic InterBase error messages during message building. For instance, when an integrity error occurs, InterBase stores the code of the trigger which detects the problem as a numeric value in the status vector. When an InterBase function like **isc_interprete()** builds the error message string for this error, it inserts the numeric value from the status vector into the generic InterBase integrity error message string to make it more specific.

OPERATING SYSTEM ERROR CODES

If the first long in a cluster is greater than 5, the numeric value in the second long is an error code specific to a particular platform or operating system. InterBase functions should not be used to retrieve and display the specific platform or operating system error message. Consult your operating system manual for information on how to handle such errors.

Meaning of the third long in a cluster

If the first long in a cluster is 3 (*isc_arg_cstring*), the cluster's total length is three longs. The third long *always* contains the address of a message string requiring translation into a standard, null-terminated C string before display. Such a string is often a file or path name. InterBase functions like **isc_interprete()** automatically handle this translation for you.

Status vector parsing example

The following C example illustrates a simple, brute force parsing of the status vector. The code forces an error condition. The error-handling block parses the status vector array cluster by cluster, printing the contents of each cluster and interpreting it for you.

```
#include <ibase.h>
. . .
ISC_STATUS status_vector[20];
main()
{
   int done, v; /* end of args?, index into vector */
   int c, extra; /* cluster count, 3 long cluster flag */
   static char *meaning[] = {"End of error information",
      "n InterBase error code"," string address"," string length",
      " numeric value"," system code"};
/* Assume database is connected and transaction started here. */
if (status_vector[0] == 1 && status_vector[1] > 0)
   error_exit();
. . .
}
void error_exit(void)
{
   done = v = 0;
   c = 1;
   while (!done)
   {
      extra = 0;
      printf("Cluster %d:\n", c);
      printf("Status vector %d: %ld: ", v, status_vector[v]);
      if (status_vector[v] != gds_arg_end)
         printf("Next long is a");
      switch (status_vector[v++])
      {
         case gds_arg_end:
            printf("%s\n", meaning[0]);
            done = 1;
            break;
         case gds_arg_gds:
            printf("%s\n", meaning[1]);
            break;
         case gds_arg_string:
         case gds_arg_interpreted:
            printf("%s\n", meaning[2]);
            break;
         case gds_arg_number:
            printf("%s\n", meaning[4]);
            break;
         case gds_arg_cstring:
            printf("%s\n", meaning[3]);
            extra = 1;
            break;
         default:
            printf("%s\n", meaning[5]);
            break;
      }
      if (!done)
      {
         printf("Status vector %d: %ld", v, status_vector[v]);
         v++;/* advance vector pointer */
         c++;/* advance cluster count */
         if (extra)
         {
            printf(": Next long is a %s\n", meaning[2]);
            printf("Status vector: %d: %ld\n\n", v,
               status_vector[v]);
            v++;
         }
         else
            printf("\n\n");
      }
   }
   isc_rollback_transaction(status_vector, &trans);
   isc_detach_database(&db1);
   return(1);
}
```

Here is a sample of the output from this program:

```
Cluster 1:
Status vector 0: 1: Next long is an InterBase error code
Status vector 1: 335544342
Cluster 2:
Status vector 2: 4: Next long is a numeric value
Status vector 3: 1
Cluster 3:
Status vector 4: 1: Next long is an InterBase error code
Status vector 5: 335544382
Cluster 4:
Status vector 6: 2: Next long is a string address
Status vector 7: 156740
```

```
Cluster 5:
Status vector 8: 0: End of error information
```

This output indicates that two InterBase errors occurred. The first error code is 335544342. The error printing routines, **isc_print_status()** and **isc_interprete(),** use the InterBase error code to retrieve a corresponding base error message. The base error message contains placeholders for replaceable parameters. For error code 335544342, the base error message string is:

```
"action cancelled by trigger (%ld) to preserve data integrity"
```

This error message uses a replaceable numeric parameter, *%ld*.

In this case, the numeric value to use for replacement, 1, is stored in the second long of the second cluster. When the error printing routine inserts the parameter into the message, it displays the message:

```
action cancelled by trigger (1) to preserve data integrity
```

The second error code is 335544382. The base message retrieved for this error code is:

```
"%s"
```

In this case, the entire message to be displayed consists of a replaceable string. The second long of the fourth cluster contains the address of the replacement string, 156740. This is an error message defined in the trigger that caused the error. When the error printing routine inserts the message from the trigger into the base message, it displays the resulting message:

```
-Department name is missing.
```

Note This sample program is only meant to illustrate the structure of the status vector and its contents. While the error-handling routine in this program might serve as a limited debugging tool for a program under development, it does not provide useful information for end users. Ordinarily, error-handling blocks in applications should interpret errors, display explanatory error messages, and take corrective action, if appropriate.

For example, if the error-handling routine in the sample program had called **isc_print_status()** to display the error messages associated with these codes, the following messages would have been displayed:

```
action cancelled by trigger (1) to preserve data integrity
-Department name is missing.
```

CHAPTER

11 Working with Events

This chapter describes how to work with *events*, a message passed from a trigger or stored procedure to an application to announce the occurrence of a specified condition or action, usually a database change such as an insertion, modification, or deletion of a record. It explains how to set up event buffers, and use the following API functions to make synchronous and asynchronous event calls. In the following table, functions are listed in the order they typically appear in an application:

Function	Purpose
isc_event_block()	Allocate event parameter buffers
isc_wait_for_event()	Wait for a synchronous event to be posted
isc_que_events()	Set up an asynchronous event and return to application processing
isc_event_counts()	Determine the change in values of event counters in the event parameter buffer
isc_cancel_events()	Cancel interest in an event

TABLE 11.1 API event functions

For asynchronous events, this chapter also describes how to create an asynchronous trap (AST), a function that responds to posted events.

Understanding the event mechanism

The InterBase event mechanism consists of four parts:

- The InterBase engine that maintains an event queue and notifies applications when an event occurs.
- Event parameter buffers set up by an application where it can receive notification of events.
- An application that registers interest in one or more specified, named events and either waits for notification to occur (synchronous event), or passes a pointer to an AST function that handles notifications so that application processing can continue in the meantime (asynchronous event).
- A trigger or stored procedure that notifies the engine that a specific, named event has occurred. Notification is called *posting*.

The InterBase event mechanism enables applications to respond to actions and database changes made by other, concurrently running applications without the need for those applications to communicate directly with one another, and without incurring the expense of CPU time required for periodic polling to determine if an event has occurred.

For information about creating triggers and stored procedures that post events, see the *Data Definition Guide*.

Event parameter buffers

If an application is to receive notification about events, it must set up two identically-sized event parameter buffers (EPBs) using **isc_event_block()**. The first buffer, *event_buffer*, is used to hold the count of event occurrences before the application registers an interest in the event. The second buffer, *result_buffer*, is subsequently filled in with an updated count of event occurrences when an event of interest to the application occurs. A second API function, **isc_event_counts()**, determines the differences between item counts in these buffers to determine which event or events occurred.

For more information about setting up and using EPBs, see **"Creating EPBs with isc_event_block( )" on page 186**.

Synchronous event notification

When an application depends on the occurrence of a specific event for processing, it should use synchronous event notification to suspend its own execution until the event occurs. For example, an automated stock trading application that buys or sells stock when specific price changes occur might start execution, set up EPBs, register interest in a set of stocks, then suspend its own execution until those price changes occur.

The **isc_wait_for_event()** function provides synchronous event handling for an application. For more information about synchronous event handling, see **"Waiting on events with isc_wait_for_event( )" on page 187**.

Asynchronous event notification

When an application needs to react to possible database events, but also needs to continue processing whether or not those events occur, it should set up an asynchronous trap (AST) function, and use asynchronous event notification to register interest in events while continuing its own processing. For example, a stock brokering application requires constant access to a database of stocks to allow a broker to buy and sell stock, but, at the same time, may want to use events to alert the broker to particularly significant or volatile stock price changes.

The **isc_que_events()** function and the AST function provide asynchronous event handling for an application. For more information about asynchronous event handling, see **"Continuous processing with isc_que_events( )" on page 188**.

Transaction control of events

Events occur under transaction control, and can therefore be committed or rolled back. Interested applications do not receive notification of an event until the transaction from which the event is posted is committed. If a posted event is rolled back, notification does not occur.

A transaction can post the same event more than once before committing, but regardless of how many times an event is posted, it is regarded as a single event occurrence for purposes of event notification.

Creating EPBs with isc_event_block( )

Before an application can register interest in an event, it must establish and populate two event parameter buffers (EPBs), one for holding the initial occurrence count values for each event of interest, and another for holding the changed occurrence count values. These buffers are passed as parameters to several API event functions.

In C, each EPB is declared as a char pointer, as follows:

```
char *event_buffer, *result_buffer;
```

Once the buffers are declared, **isc_event_block()** is called to allocate space for them, and to populate them with starting values.

isc_event_block() also requires at least two additional parameters: the number of events in which an application is registering interest, and, for each event, a string naming the event. A single call to **isc_event_block()** can pass up to 15 event name strings. Event names must match the names of events posted by stored procedures or triggers.

isc_event_block() allocates the same amount of space for each EPB, enough to handle each named event. It returns a single value, indicating the size, in bytes, of each buffer.

The syntax for **isc_event_block()** is:

```
ISC_LONG isc_event_block(
char **event_buffer,
char **result_buffer,
unsigned short id_count,
. . . );
```

For example, the following code sets up EPBs for three events:

```
#include <ibase.h>;
. . .
char *event_buffer, *result_buffer;
long blength;
. . .
blength = isc_event_block(&event_buffer, &result_buffer, 3, "BORL",
"INTEL", "SUN");
. . .
```

This code assumes that there are triggers or stored procedures defined for the database that post events named “BORL”, “INTEL”, and “SUN”.

TIP Applications that need to respond to more than 15 events can make multiple calls to **isc_event_block()**, specifying different EPBs and event lists for each call.

For the complete syntax of **isc_event_block()**, see **“isc_event_block()” on page 295**.

Waiting on events with isc_wait_for_event( )

After setting up EPBs and specifying events of interest with **isc_event_block()**, an application can use **isc_wait_for_event()** to register interest in those events and pause its execution until one of the events occurs.

IMPORTANT **isc_wait_for_event()** cannot be used in Microsoft Windows applications or under any other operating system that does not permit processes to pause. Applications on these platforms must use asynchronous event handling.

The syntax for **isc_wait_for_event()** is:

```
ISC_STATUS isc_wait_for_event(
ISC_STATUS *status_vector,
isc_db_handle *db_handle,
short length,
char *event_buffer,
char *result_buffer);
```

For example, the following code sets up EPBs for three events, then calls **isc_wait_for_event()** to suspend its execution until one of the events occurs:

```
#include <ibase.h>;
. . .
char *event_buffer, *result_buffer;
long blength;
ISC_STATUS status_vector[20];
isc_db_handle db1;
. . .
/* Assume database db1 is attached here and a transaction started. */
blength = isc_event_block(&event_buffer, &result_buffer, 3, "BORL",
"INTEL", "SUN");
isc_wait_for_event(status_vector, &db1, (short)blength,
   event_buffer, result_buffer);
/* Application processing is suspended here until an event occurs. */
. . .
```

Once **isc_wait_for_event()** is called, application processing stops until one of the requested events is posted. When the event occurs, application processing resumes at the next executable statement following the call to **isc_wait_for_event()**. If an application is waiting on more than one event, it must use **isc_event_counts()** to determine which event was posted.

Note A single call to **isc_wait_for_event()** can wait on a maximum of 15 events. Applications that need to wait on more than 15 events must wait on one set of 15, then make another call to **isc_wait_for_event()** to wait on additional events.

For the complete syntax of **isc_wait_for_event()**, see **"isc_wait_for_event()" on page 336**.

Continuous processing with isc_que_events( )

isc_que_events() is called to request asynchronous notification of events listed in an event buffer passed as an argument. Upon completion of the call, but before any events are posted, control is returned to the calling application so that it can continue processing.

When a requested event is posted, InterBase calls an asynchronous trap (AST) function, also passed as a parameter to **isc_que_events()**, to handle the posting. The AST is a function or subroutine in the calling application, the sole purpose of which is to process the event posting for the application.

The syntax for **isc_que_events()** is:

```
ISC_STATUS isc_que_events(
ISC_STATUS *status_vector,
isc_db_handle *db_handle,
ISC_LONG *event_id,
short length,
char *event_buffer,
isc_callback event_function,
void *event_function_arg);
```

event_id is a long pointer that is used as a handle in subsequent calls to **isc_cancel_events()** to terminate event notification. It need not be initialized when passed. The *length* parameter is the size of *event_buffer*, which contains the current count of events to be waited upon. *event_function* is a pointer to the AST function that InterBase should call when an event of interest is posted. It is up to the AST function to notify the application that it has been called, perhaps by setting a global flag of some kind. *event_function_arg* is a pointer to the first parameter to pass to the AST.

For a complete example of a call to **isc_que_events()** and a call to an AST, see **"A complete isc_que_events( ) example" on page 189**.

Creating an AST

The event function, *event_function,* should be written to take three arguments:

1. The *event_function_arg* specified in the call to **isc_que_events()**. This is usually a pointer to the event parameter buffer that should be filled in with updated event counts.
2. The length of the following *events_list* buffer.
3. A pointer to the *events_list* buffer, a temporary event parameter buffer just like that passed to **isc_que_events()**, except for having updated event counts.

A result buffer is not automatically updated by the event occurrence; it is up to the *event_function* to copy the temporary *events_list* buffer to the more permanent buffer that the application utilizes.

event_function also needs to let the application know that it has been called, for example, by setting a global flag.

A sample *event_function* appears in the following example:

```
isc_callback event_function
   (char *result, short length, char *updated)
{
   /* Set the global event flag. */
   event_flag++
   /* Copy the temporary updated buffer to the result buffer. */
   while (length--)
      *result++ = *updated++;
   return(0);
};
```

A complete isc_que_events( ) example

The following program fragment illustrates calling **isc_que_events()** to wait asynchronously for event occurrences. Within a loop, it performs other processing, and checks the event flag (presumably set by the specified event function) to determine when an event has been posted. If one has, the program resets the event flag, calls **isc_event_counts()** to determine which events have been posted since the last call to **isc_que_events()**, and calls **isc_que_events()** to initiate another asynchronous wait.

```
#include <ibase.h>
#define number_of_stocks 3;
#define MAX_LOOP 10
char *event_names[] = {"DEC", "HP", "SUN"};
char *event_buffer, *result_buffer;
ISC_STATUS status_vector[20];
short length;
```

```
ISC_LONG event_id;
int i, counter;
int event_flag = 0;
length = (short)isc_event_block(
   &event_buffer,
   &result_buffer,
   number_of_stocks,
   "DEC", "HP", "SUN");
isc_que_events(
   status_vector,
   &database_handle, /* Set in previous isc_attach_database(). */
   &event_id,
   length, /* Returned from isc_event_block(). */
   event_buffer,
   (isc_callback)event_function,
   result_buffer);
if (status_vector[0] == 1 && status_vector[1])
{
   isc_print_status(status_vector); /* Display error message. */
   return(1);
};
counter = 0;
while (counter < MAX_LOOP)
{
   counter++;
   if (!event_flag)
   {
      /* Do whatever other processing you want. */
      ;
   }
   else
   {  event_flag = 0;
      isc_event_counts(
         status_vector,
         length,
         event_buffer,
         result_buffer);
      if (status_vector[0] == 1 && status_vector[1])
      {
         isc_print_status(status_vector); /* Display error message.
*/
         return(1);
```

```
            };
            for (i=0; i<number_of_stocks; i++)
               if (status_vector[i])
               {
                  /* The event has been posted. Do whatever is appropriate,
                     e.g., initiating a buy or sell order.
                     Note: event_names[i] tells the name of the event
                     corresponding to status_vector[i]. */
                  ;
               }
            isc_que_events(
               status_vector,
               &database_handle,
               &event_id,
               length,
               event_buffer,
               (isc_callback)event_function,
               result_buffer);
            if (status_vector[0] == 1 && status_vector[1])
            {
               isc_print_status(status_vector); /* Display error message.
*/
               return(1);
            }
         }  /* End of else. */
      }  /* End of while. */
      /* Let InterBase know you no longer want to wait asynchronously. */
      isc_cancel_events(
         status_vector,
         &database_handle,
         &event_id);
      if (status_vector[0] == 1 && status_vector[1])
      {
         isc_print_status(status_vector); /* Display error message. */
         return(1);
      }
```

Determining which events occurred with isc_event_counts( )

When an application registers interest in multiple events and receives notification that an event occurred, the application must use **isc_event_counts()** to determine which event or events occurred. **isc_event_counts()** subtracts values in the *event_buffer* array from the values in the *result_buffer* array to determine the number of times each event has occurred since an application registered interest in a set of events. *event_buffer* and *result_buffer* are variables declared within an application, and allocated and initialized by **isc_event_block()**.

The difference of each element is returned in the error status array that is passed to **isc_event_counts()**. To determine which events occurred, an application must examine each element of the array for nonzero values. A nonzero count indicates the number of times an event is posted between the time **isc_event_block()** is called and the first time an event is posted after **isc_wait_for_event()** or **isc_que_events()** are called. Where multiple applications are accessing the same database, therefore, a particular event count may be 1 or more, and more than one event count element may be nonzero.

Note When first setting up an AST to trap events with **isc_que_events()**, InterBase initializes all count values in the status vector to 1, rather than 0. To clear the values, call **isc_event_counts()** to reset the values.

In addition to determining which event occurred, **isc_event_counts()** reinitializes the *event_buffer* array in anticipation of another call to **isc_wait_for_event()** or **isc_que_events()**. Values in *event_buffer* are set to the same values as corresponding values in *result_buffer.*

The complete syntax for **isc_event_counts()** is:

```
void isc_event_counts(
ISC_STATUS status_vector,
short buffer_length,
char *event_buffer,
char *result_buffer);
```

For example, the following code declares interest in three events, waits on them, then uses **isc_event_counts()** to determine which events occurred:

```
#include <ibase.h>;
. . .
char *event_buffer, *result_buffer;
long blength;
ISC_STATUS status_vector[20];
isc_db_handle db1;
long count_array[3];
int i;
```

```
. . .
/* Assume database db1 is attached here and a transaction started. */
blength = isc_event_block(&event_buffer, &result_buffer, 3, "BORL",
"INTEL", "SUN");
isc_wait_for_event(status_vector, &db1, (short)blength,
   event_buffer, result_buffer);
/* Application processing is suspended here until an event occurs. */
isc_event_counts(status_vector, (short)blength, event_buffer,
   result_buffer);
for (i = 0; i < 3; i++)
{
   if (status_vector[i])
   {
      /* Process the event here. */
      ;
   }
}
```

For more information about **isc_event_counts()**, see **"isc_event_counts()" on page 296** of **Chapter 12, "API Function Reference."**

Canceling interest in asynchronous events with isc_cancel_events( )

An application that requested asynchronous event notification with **isc_que_events()** can subsequently cancel the notification request at any time with **isc_cancel_events()** using the following syntax:

```
ISC_STATUS isc_cancel_events(
ISC_STATUS *status_vector,
isc_db_handle *db_handle,
ISC_LONG *event_id);
```

event_id is an event handle set in a previous call to **isc_que_events()**. For example, the following code cancels interest in the event or events identified by *event_id*:

```
include <ibase.h>;
. . .
/* For example code leading up to this call, see the code example
in "Continuous Processing with isc_event_que(), earlier in this
chapter. */
isc_cancel_events(status_vector, &db_handle, &event_id);
```

PART II

API Reference Guide

CHAPTER

12 API Function Reference

This chapter is an alphabetical reference for the InterBase API function calls. It provides tables that categorize calls by the tasks they perform, and then provides an alphabetical and detailed description of each call, including its syntax, arguments, examples of use, and cross references to related calls.

Function categories

There are ten classes of InterBase API function calls:

- Array functions for handling arrays of data
- Blob functions for handling the InterBase Blob datatype
- Database functions for handling database requests
- Conversion functions for translating dates between InterBase format and Unix *tm* format, and for reversing the byte-order of integers
- DSQL functions for handling SQL statements entered by users at run time
- Error-handling functions
- Event functions for registering interest in events posted by triggers and stored procedures in applications and for processing the event queue

- Information functions for retrieving information about databases, transactions, Blob data, and events
- Security functions for adding, deleting, and modifying user records in the password database
- Transaction functions for handling transactions in an application

Some functions, such as information calls, occur in more than one class.

Array functions

The following table summarizes the InterBase API functions available for handling array data in an application:

Function name	Purpose
isc_array_get_slice()	Retrieve a specified part of an array field
isc_array_lookup_bounds()	Determine the dimensions of an array field
isc_array_lookup_desc()	Retrieve an array description
isc_array_put_slice()	Write a specified part of an array field
isc_array_set_desc()	Set an array description

TABLE 12.1 Array functions

Blob functions

The following table summarizes the InterBase API functions available for handling Blob data in an application:

Function name	Purpose
isc_blob_default_desc()	Set a default Blob description for dynamic access
isc_blob_gen_bpb()	Generate a Blob parameter buffer (BPB) for dynamic access
isc_blob_info()	Request information about a Blob field
isc_blob_lookup_desc()	Retrieve a Blob description
isc_blob_set_desc()	Set a Blob description
isc_cancel_blob()	Discard a Blob
isc_close_blob()	Close a Blob
isc_create_blob2()	Create a new Blob
isc_get_segment()	Retrieve a segment of Blob data
isc_open_blob2()	Open a Blob for read access
isc_put_segment()	Write a segment of Blob data

TABLE 12.2 Blob functions

Database functions

The following table summarizes the InterBase API functions available for handling database requests in an application:

Function name	Purpose
isc_attach_database()	Connect to an existing database
isc_database_info()	Request information about an attached database
isc_detach_database()	Disconnect from a database
isc_drop_database()	Delete an attached database and its associated files
isc_expand_dpb()	Build a database parameter buffer (DPB) dynamically
isc_version()	Retrieve database implementation number and on-disk structure (ODS) major and minor version numbers

TABLE 12.3 Database functions

Conversion functions

The following table summarizes the InterBase API functions available for translating between InterBase DATE format and the Unix *tm* date format, and for reversing the byte-order of an integer:

Function name	Purpose
isc_decode_date()	Translate a date from InterBase format to C *tm* format
isc_encode_date()	Translate a date from C *tm* format to InterBase format
isc_vax_integer()	Reverse the byte-order of an integer

TABLE 12.4 Date and conversion functions

DSQL functions

The following table summarizes the InterBase API functions available for handling DSQL statements built or entered by users at run time:

Function name	Purpose
isc_dsql_allocate_statement()	Allocate a statement handle
isc_dsql_alloc_statement2()	Allocate a statement handle that is automatically freed on database detachment
isc_dsql_describe()	Fill in an XSQLDA with information about values returned by a statement
isc_dsql_describe_bind()	Fill in an XSQLDA with information about a statement's input parameters
isc_dsql_execute()	Execute a prepared statement
isc_dsql_execute2()	Execute a prepared statement returning a single set of values
isc_dsql_execute_immediate()	Prepare and execute a statement without return values for one-time use
isc_dsql_exec_immed2()	Prepare and execute a statement with a single set of return values for one-time use
isc_dsql_fetch()	Retrieve data returned by a previously prepared and executed statement
isc_dsql_free_statement()	Free a statement handle, or close a cursor associated with a statement handle
isc_dsql_prepare()	Prepare a statement for execution
isc_dsql_set_cursor_name()	Define a cursor name and associate it with a statement handle
isc_dsql_sql_info()	Request information about a prepared statement

TABLE 12.5 DSQL functions

Error-handling functions

The following table summarizes the InterBase API functions available for handling database error conditions an application:

Function name	Purpose
isc_interprete()	Capture InterBase error messages to a buffer
isc_print_sqlerror()	Display an SQL error message
isc_print_status()	Display InterBase error messages
isc_sqlcode()	Set the value of SQLCODE
isc_sql_interprete()	Capture an SQL error message to a buffer

TABLE 12.6 Error-handling functions

Event functions

The following table summarizes the InterBase API functions available for handling events in an application:

Function name	Purpose
isc_cancel_events()	Cancel interest in an event
isc_event_block()	Allocate event parameter buffers
isc_event_counts()	Get the change in values of event counters in the event array
isc_que_events()	Wait asynchronously until an event is posted
isc_wait_for_event()	Wait synchronously until an event is posted

TABLE 12.7 Event functions

Information functions

The following table summarizes the InterBase API functions available for reporting information about databases, transactions, and Blob data to a client application that requests it:

Function name	Purpose
isc_blob_info()	Request information about a Blob field
isc_database_info()	Request information about an attached database
isc_dsql_sql_info()	Request information about a prepared DSQL statement
isc_transaction_info()	Request information about a specified transaction
isc_version()	Retrieve database implementation number and on-disk structure (ODS) major and minor version numbers

TABLE 12.8 Information functions

Security functions

The following table summarizes the InterBase API functions available for adding, deleting, and modifying user records in the password database:

Function name	Purpose
isc_add_user()	Adds a user to the password database
isc_delete_user()	Deletes a user from the password database
isc_modify_user()	Modifies user information in the password database

TABLE 12.9 Security functions

Transaction control functions

The following table summarizes the InterBase API functions available for controlling transactions in an application:

Function name	Purpose
isc_commit_retaining()	Commit a transaction, and start a new one using the original transaction's context
isc_commit_transaction()	Save a transaction's database changes, and end the transaction
isc_prepare_transaction()	Execute the first phase of a two-phase commit
isc_prepare_transaction2()	Execute the second phase of a two-phase commit
isc_rollback_transaction()	Undo a transaction's database changes, and end the transaction
isc_start_multiple()	Begin new transactions (used on systems that do not support a variable number of input arguments)
isc_start_transaction()	Begin new transactions
isc_transaction_info()	Request information about a specified transaction

TABLE 12.10 Transaction control functions

Using function definitions

Each function definition in this chapter includes the elements in the following table:

Element	Description
Title	Function name
Definition	Main purpose of function
Syntax	Diagram of the function and parameters
Parameters	Table describing each parameter
Description	Detailed information about using the function
Example	Example of using the function in a program
Return value	Description of possible values returned in the status vector, if any
See also	Cross references to other related functions

TABLE 12.11 Function description format

isc_add_user( )

Adds a user record to the password database, *isc4.gdb*.

Syntax

```
ISC_STATUS isc_add_user(
ISC_STATUS *status
USER_SEC_DATA *user_sec_data);
```

Parameter	Type	Description
status vector	ISC_STATUS *	Pointer to the error status vector
user_sec_data	USER_SEC_DATA *	Pointer to a struct that is defined in *ibase.h*

Description The three security functions, **isc_add_user()**, **isc_delete_user()**, and **isc_modify_user()** mirror functionality that is available in the **gsec** command-line utility. **isc_add_user()** adds a record to *isc4.gdb*, InterBase's password database.

At a minimum, you must provide the user name and password. If the server is not local, you must also provide a server name and protocol. Valid choices for the protocol field are *sec_protocol_tcpip*, *sec_protocol_netbeui*, *sec_protocol_spx*, and *sec_protocol_local*.

InterBase reads the settings for the ISC_USER and ISC_PASSWORD environment variables if you do not provide a DBA user name and password.

The definition for the USER_SEC_DATA struct in *ibase.h* is as follows:

```
typedef struct {
   short  sec_flags;      /* which fields are specified */
   int    uid;            /* the user’s id */
   int    gid;            /* the user’s group id */
   int    protocol;       /* protocol to use for connection */
   char   *server;        /* server to administer */
   char   *user_name;     /* the user’s name */
   char   *password;      /* the user’s password */
   char   *group_name;    /* the group name */
   char   *first_name;    /* the user’s first name */
   char   *middle_name;   /* the user’s middle name */
   char   *last_name;     /* the user’s last name */
   char   *dba_user_name; /* the dba user name */
   char   *dba_password;  /* the dba password */
} USER_SEC_DATA;
```

When you pass this struct to one of the three security functions, you can tell it which fields you have specified by doing a bitwise OR of the following values, which are defined in *ibase.h*:

```
sec_uid_spec                0x01
sec_gid_spec                0x02
sec_server_spec             0x04
sec_password_spec           0x08
sec_group_name_spec         0x10
sec_first_name_spec         0x20
sec_middle_name_spec        0x40
sec_last_name_spec          0x80
sec_dba_user_name_spec      0x100
sec_dba_password_spec        0x200
```

No bit values are available for user name and password, since they are required.

The following error messages exist for this function:

Code	Value	Description
isc_usrname_too_long	335544747	The user name passed in is greater than 31 bytes
isc_password_too_long	335544748	The password passed in is longer than 8 bytes
isc_usrname_required	335544749	The operation requires a user name
isc_password_required	335544750	The operation requires a password
isc_bad_protocol	335544751	The protocol specified is invalid
isc_dup_usrname_found	335544752	The user name being added already exists in the security database
isc_usrname_not_found	335544753	The user name was not found in the security database
isc_error_adding_sec_record	335544754	An unknown error occurred while adding a user
isc_error_deleting_sec_record	335544755	An unknown error occurred while deleting a user
isc_error_modifying_sec_record	335544756	An unknown error occurred while modifying a user
isc_error_updating_sec_db	335544757	An unknown error occurred while updating the security database

TABLE 12.12 Error messages for user security functions

Example The following example adds a user (“Socks”) to the password database, using the bitwise OR technique for passing values from the USER_SEC_DATA struct.

```
{
   ISC_STATUS status[20];
   USER_SEC_DATA sec;

   sec.server           = "kennel";
   sec.dba_user_name    = "sysdba";
   sec.dba_password     = "masterkey";
   sec.protocol         = sec_protocol_tcpip;
   sec.first_name       = "Socks";
   sec.last_name        = "Clinton";
   sec.user_name        = "socks";
   sec.password         = "2meow!"; /* Note: do not hardcode passwords
*/
   sec.sec_flags        = sec_server_spec
                        | sec_password_spec
                        | sec_dba_user_name_spec
                        | sec_dba_password_spec
                        | sec_first_name_spec
                        | sec_last_name_spec;
   isc_add_user(status, &sec);
   /* check status for errors */
   if (status[0] == 1 && status[1])
   {
      switch (status[1]) {
      case isc_usrname_too_long:
         printf("Security database cannot accept long user names\n");
         break;
      ...
      }
   }
}
```

Return Value **isc_add_user()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. See the “Description” section for this function for a list of error codes. For more information about examining the status vector, see **Chapter 10, “Handling Error Conditions.”**

See Also **isc_delete_user( )**, **isc_modify_user( )**

isc_array_get_slice()

Retrieves data from an array column in a row returned by a SELECT.

Syntax

```
ISC_STATUS isc_array_get_slice(
ISC_STATUS *status_vector,
isc_db_handle *db_handle,
isc_tr_handle *trans_handle,
ISC_QUAD *array_id,
ISC_ARRAY_DESC *desc,
void *dest_array,
ISC_LONG *slice_length);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
db_handle	*isc_db_handle* *	Pointer to a database handle set by a previous call to **isc_attach_database()**; the handle identifies the database containing the array column *db_handle* returns an error in *status_vector* if it is NULL
trans_handle	*isc_tr_handle* *	Pointer to a transaction handle whose value has been set by a previous **isc_start_transaction()** call; *trans_handle* returns an error if NULL
array_id	ISC_QUAD *	Internal identifier for the array; the array ID must be previously retrieved through API DSQL functions
desc	ISC_ARRAY_DESC *	Descriptor defining the array slice (entire array or subset) to be retrieved
dest_array	void *	Pointer to a buffer of length *slice_length* into which the array slice will be copied by this function
slice_length	ISC_LONG *	Length, in bytes, of the *dest_array* buffer

isc_array_get_slice() retrieves data from an array column of a table row using an array ID. You can either retrieve all the array elements in that column, or a subset of contiguous array elements, called a *slice*. The upper and lower boundaries in the *desc* structure specify which elements are to be retrieved.

InterBase copies the elements into the buffer, *dest_array*, whose size is specified by *slice_length*. This should be at least the expected length required for the elements retrieved. Before returning from **isc_array_get_slice()**, InterBase sets *slice_length* to the actual number of bytes copied.

Before calling **isc_array_get_slice()**, there are many operations you must do in order to fill in the array descriptor, *desc*, determine the appropriate internal array identifier, *array_id*, and fetch the rows whose array columns you want to access. For complete step-by-step instructions for setting up an array descriptor and retrieving array information, see **Chapter 8, "Working with Array Data."**

Note *Never* execute a DSQL statement that tries to access array column data directly unless you are fetching only a single element. The way to access slices of array column data is to call **isc_array_get_slice()** or **isc_array_put_slice()**. The only supported array references in DSQL statements are ones that specify an entire array column (that is, just the column name) in order to get the internal identifier for the array, which is required by **isc_array_get_slice()** and **isc_array_put_slice()**, or single element references.

Example The following program operates on a table named PROJ_DEPT_BUDGET. This table contains the quarterly head counts allocated for each project in each department of an organization. Each row of the table applies to a particular department and project. The quarterly head counts are contained in an array column named QUARTERLY_HEAD_CNT. Each row has four elements in this column, one per quarter. Each element of the array is a number of type *long*.

The example below selects the rows containing 1994 information for the project named VBASE. For each such row, it retrieves and prints the department number and the data in the array column (that is, the quarterly head counts).

In addition to illustrating the usage of **isc_array_lookup_bounds()** and **isc_array_get_slice()**, the program shows data structure initializations and calls to the DSQL functions required to prepare and execute the SELECT statement, to obtain the *array_id* needed by **isc_array_get_slice()**, and to fetch the selected rows one by one.

```
#include <ibase.h>
#define Return_if_Error(stat) if (stat[0] == 1 && stat[1]) \
                    { \
                    isc_print_status(stat); \
                    return(1); \
                    }
char *sel_str =
   "SELECT dept_no, quarterly_head_cnt FROM proj_dept_budget \
      WHERE year = 1994 AND proj_id = 'VBASE'";
char dept_no[6];
long hcnt[4], tr_handle, database_handle, SQLCODE;
```

```
short len, i, flag0, flag1;
ISC_QUAD array_id;
ISC_ARRAY_DESC desc;
ISC_STATUS status_vector[20], fetch_stat;
isc_stmt_handle stmt = NULL;
XSQLDA *osqlda;
tr_handle = database_handle = 0L;
/* Attach to a database here--this code omitted for brevity */
/* Start a transaction here--this code omitted for brevity */
/* Set up the SELECT statement. */
/* Allocate the output XSQLDA for holding the array data. */
osqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(2));
osqlda->sqln = 2;
osqlda->version = 1;
/* Allocate a statement handle. */
isc_dsql_allocate_statement(
   status_vector,
   &database_handle,
   &stmt);
Return_if_Error(status_vector);
/* Prepare the query for execution. */
isc_dsql_prepare(
   status_vector,
   &tr_handle,
   &stmt,
   0,
   sel_str,
   1,
   osqlda);
Return_if_Error(status_vector);
/* Set up an XSQLVAR structure to allocate space for each item
   to be retrieved. */
osqlda->sqlvar[0].sqldata = (char *) dept_no;
osqlda->sqlvar[0].sqltype = SQL_TEXT + 1;
osqlda->sqlvar[0].sqlind = &flag0;
osqlda->sqlvar[1].sqldata = (char *) &array_id;
osqlda->sqlvar[1].sqltype = SQL_ARRAY + 1;
osqlda->sqlvar[1].sqlind = &flag1;
/* Execute the SELECT statement. */
isc_dsql_execute(
   status_vector,
   &tr_handle,
   &stmt,
   1,
   NULL);
Return_if_Error(status_vector);
/* Set up the array descriptor. */
isc_array_lookup_bounds(
   status_vector,
   &database_handle, /* Set by previous isc_attach_database() call. */
   &tr_handle, /* Set by previous isc_start_transaction() call. */
   "PROJ_DEPT_BUDGET", /* Table name. */
   "QUARTERLY_HEAD_CNT", /* Array column name. */
   &desc);
Return_if_Error(status_vector);

/* Fetch the head count for each department’s four quarters. */
while ((fetch_stat = isc_dsql_fetch(
   status_vector,
   &stmt,
   1,
   osqlda)) == 0)
{
   if (!flag1)
   {
       /* There is array data; get the current values. */
       len = sizeof(hcnt);
       /* Fetch the data from the array column into hcnt array. */
       isc_array_get_slice(
           status_vector,
           &database_handle,
           &tr_handle,
           &array_id,
           &desc,
           hcnt,
           &len);
       Return_if_Error(status_vector);

       /* Print department number and head counts. */
       dept_no[osqlda->sqlvar[0].sqllen] = ’\0’;
       printf("Department #: %s\n\n", dept_no);
       printf("\tCurrent counts: %d %d %d %d\n",
           hcnt[0], hcnt[1], hcnt[2], hcnt[3]);
   };
```

```
}
if (fetch_stat != 100L)
{
   SQLCODE = isc_sqlcode(status_vector);
   isc_print_sqlerror(SQLCODE, status_vector);
   return(1);
}
```

Return Value **isc_array_get_slice()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to *isc_bad_stmt_handle*, *isc_bad_trans_handle*, or another InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, “Handling Error Conditions.”**

See Also **isc_array_lookup_bounds()**, **isc_array_lookup_desc()**, **isc_array_put_slice()**, **isc_array_set_desc()**, **isc_dsql_fetch()**, **isc_dsql_prepare()**

isc_array_lookup_bounds()

Determines the datatype, length, scale, dimensions, and array boundaries for the specified array column in the specified table.

Syntax

```
ISC_STATUS isc_array_lookup_bounds(
ISC_STATUS *status_vector,
isc_db_handle *db_handle,
isc_tr_handle *trans_handle,
char *table_name,
char *column_name,
ISC_ARRAY_DESC *desc);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
db_handle	*isc_db_handle* *	Pointer to a database handle set by a previous call to **isc_attach_database()**; the handle identifies the database containing the array column *db_handle* returns an error in *status_vector* if it is NULL
trans_handle	*isc_tr_handle* *	Pointer to a transaction handle whose value has been set by a previous **isc_start_transaction()** call; *trans_handle* returns an error if NULL
table_name	char *	Name of the table containing the array column, column_name; can be either null-terminated or blank-terminated
column_name	char *	Name of the array column; can be either null-terminated or blank-terminated
desc	ISC_ARRAY_DESC *	Pointer to a descriptor for the arrays that will be filled in by this function

Description **isc_array_lookup_bounds()** determines the datatype, length, scale, dimensions, and array boundaries for the elements in an array column, *column_name* in the table, *table_name*. It stores this information in the array descriptor, *desc*.

isc_array_lookup_bounds() also sets to 0 a flag in the descriptor. This specifies that the array should be accessed in future function calls in row-major order, the default. If an application requires column-major access, reset this flag to 1.

The array descriptor is used in subsequent calls to **isc_array_get_slice()** or **isc_array_put_slice()**.

For a detailed description of the array descriptor, see **Chapter 8, "Working with Array Data."**

Note There are ways to fill in an array descriptor other than by calling **isc_array_lookup_bounds()**. You can also:

- Call **isc_array_lookup_desc()**. This is exactly the same as calling **isc_array_lookup_bounds()**, except that the former does not fill in information about the upper and lower bounds of each dimension.
- Call **isc_array_set_desc()** to initialize the descriptor from parameters you call it with, rather than accessing the database metadata.
- Set the descriptor fields directly. Note that *array_desc_dtype* must be expressed as one of the datatypes in the following table, and the parameters, *array_desc_field_name*, and *array_desc_relation_name*, must be null-terminated:

array_desc_dtype	Corresponding InterBase datatype
blr_text	CHAR
blr_text2	CHAR
blr_short	SMALLINT
blr_long	INTEGER
blr_quad	ISC_QUAD structure
blr_float	FLOAT
blr_double	DOUBLE PRECISION
blr_date	DATE
blr_varying	VARCHAR
blr_varying2	VARCHAR
blr_blob_id	ISC_QUAD structure
blr_cstring	NULL-terminated string
blr_cstring2	NULL-terminated string

TABLE 12.13 Datatypes for array descriptor fields

Example The following illustrates a sample call to **isc_array_lookup_bounds()**. More complete examples of accessing arrays are found in the example programs for **isc_array_get_slice()** and **isc_array_put_slice()**.

```
#include <ibase.h>

ISC_STATUS status_vector[20];
ISC_ARRAY_DESC desc;
char *str1 = "PROJ_DEPT_BUDGET";
char *str2 = "QUARTERLY_HEAD_CNT";

isc_array_lookup_bounds(
   status_vector,
   &database_handle, /* Set in previous isc_attach_database() call. */
   &tr_handle, /* Set in previous isc_start_transaction() call. */
   str1,
   str2,
   &desc);
if (status_vector[0] == 1 && status_vector[1])
{
   /* Process error. */
   isc_print_status(status_vector);
   return(1);
}
```

Return Value **isc_array_lookup_bounds()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to *isc_bad_stmt_handle*, *isc_bad_trans_handle*, *isc_fld_not_def,* or another InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_array_get_slice()**, **isc_array_lookup_desc()**, **isc_array_put_slice()**, **isc_array_set_desc()**

isc_array_lookup_desc()

Determines the datatype, length, scale, and dimensions for all elements in the specified array column in the specified table.

Syntax

```
ISC_STATUS isc_array_lookup_desc(
ISC_STATUS *status_vector,
isc_db_handle *db_handle,
isc_tr_handle *trans_handle,
char *table_name,
char *column_name,
ISC_ARRAY_DESC *desc);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
db_handle	*isc_db_handle **	Pointer to a database handle set by a previous call to **isc_attach_database()**; the handle identifies the database containing the array column *db_handle* returns an error in *status_vector* if it is NULL
trans_handle	*isc_tr_handle **	Pointer to a transaction handle whose value has been set by a previous **isc_start_transaction()** call; *trans_handle* returns an error if NULL
table_name	char *	Name of the table containing the array column column_name; can be either null-terminated or blank-terminated
column_name	char *	Name of the array column; can be either null-terminated or blank-terminated
desc	ISC_ARRAY_DESC *	Pointer to an array descriptor that will be filled in by this function

Description

isc_array_lookup_desc() determines the datatype, length, scale, and dimensions for the array column, *column_name,* in the table, *table_name*. It stores this information in the array descriptor, *desc*.

It also sets to 0 a flag in the descriptor. This specifies that the array is accessed in future function calls in row-major order, the default. If an application requires column-major access, reset this flag to 1.

The array descriptor is used in subsequent calls to **isc_array_get_slice()** or **isc_array_put_slice()**.

For a detailed description of the array descriptor, see **Chapter 8, "Working with Array Data."**

Note There are ways to fill in an array descriptor other than by calling **isc_array_lookup_desc()**. You can also:

- Call **isc_array_lookup_bounds()**. This is like **isc_array_lookup_desc()**, except that **isc_array_lookup_bounds()** also fills in information about the upper and lower bounds of each dimension.
- Call **isc_array_set_desc()**, to initialize the descriptor from parameters you call it with, rather than accessing the database metadata.
- Set the descriptor fields directly. Note that *array_desc_dtype* must be expressed as one of the datatypes in the following table, and the parameters, *array_desc_field_name*, and *array_desc_relation_name*, must be null-terminated:

array_desc_dtype	**Corresponding InterBase datatype**
blr_text	CHAR
blr_text2	CHAR
blr_short	SMALLINT
blr_long	INTEGER
blr_quad	ISC_QUAD structure
blr_float	FLOAT
blr_double	DOUBLE PRECISION
blr_date	DATE
blr_varying	VARCHAR
blr_varying2	VARCHAR
blr_blob_id	ISC_QUAD structure
blr_cstring	NULL-terminated string
blr_cstring2	NULL-terminated string

TABLE 12.14 Datatypes for array descriptor fields

Example The following illustrates a sample call to **isc_array_lookup_desc()**. More complete examples of accessing arrays are found in the example programs for **isc_array_get_slice()** and **isc_array_put_slice()**.

```
#include <ibase.h>
ISC_STATUS status_vector[20];
ISC_ARRAY_DESC desc;
char str1 = "PROJ_DEPT_BUDGET";
char str2 = "QUARTERLY_HEAD_CNT";

isc_array_lookup_desc(
   status_vector,
   &database_handle, /* Set in previous isc_attach_database() call. */
   &tr_handle, /* Set in previous isc_start_transaction() call. */
   str1,
   str2,
   &desc);
if (status_vector[0] == 1 && status_vector[1])
{
   /* Process error. */
   isc_print_status(status_vector);
   return(1);
};
```

Return Value **isc_array_lookup_desc()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to *isc_bad_stmt_handle*, *isc_bad_trans_handle*, *isc_fld_not_def,* or another InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_array_get_slice()**, **isc_array_lookup_bounds()**, **isc_array_put_slice()**, **isc_array_set_desc()**

isc_array_put_slice()

Writes data into an array column.

Syntax

```
ISC_STATUS isc_array_put_slice(
ISC_STATUS *status_vector,
isc_db_handle *db_handle,
isc_tr_handle *trans_handle,
ISC_QUAD *array_id,
ISC_ARRAY_DESC *desc,
void *source_array,
ISC_LONG *slice_length);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
db_handle	*isc_db_handle* *	Pointer to a database handle set by a previous call to **isc_attach_database()**; the handle identifies the database containing the array column *db_handle* returns an error in *status_vector* if it is NULL
trans_handle	*isc_tr_handle* *	Pointer to a transaction handle whose value has been set by a previous **isc_start_transaction()** call; *trans_handle* returns an error if NULL
array_id	ISC_QUAD *	On input, NULL (if you are creating a new array), or the internal identifier for an array to be modified, as assigned by the InterBase engine. This internal identifier must have been determined by previous calls to DSQL functions. This function changes *array_id* to be the identifier for the array it creates or modifies (see below).
desc	ISC_ARRAY_DESC *	Descriptor defining the array slice (entire array or subset) to be written to
source_array	void *	Pointer to a buffer of length slice_length, that contains the slice of data that will be copied to the array by this function
slice_length	ISC_LONG *	Length, in bytes, of the *source_array* buffer

Description **isc_array_put_slice()** writes data into an array column. You can either store into all the array elements in that column, or into an array *slice*, a subset of contiguous array elements. The boundaries passed to the function in the array descriptor, *desc*, specify which elements are to be stored into.

InterBase copies the elements from the buffer, *source_array*, whose size is specified by *slice_length*.

The array identifier (array ID), *array_id*, should be passed as NULL if you are calling **isc_array_put_slice()** to create a new array. If you are calling it to modify an existing array, then *array_id* should be the identifier of the array to be modified. This must have been determined by previous calls to DSQL functions.

When **isc_array_put_slice()** is called with an array ID of an existing array, it:

1. Creates a new array with the same dimensions, bounds, etc., as the specified array, and copies the existing array data to the new array.
2. Writes the data from the array buffer, *source_array*, to the new array (or slice of the array), per the bounds specified in the array descriptor, *desc*.
3. Returns in the same *array_id* variable the array ID of the new array.

When **isc_array_put_slice()** is called with a NULL array ID, it:

1. Creates a new empty array with dimensions, bounds, etc., as declared for the array column whose name and table name are specified in the array descriptor, *desc*.
2. Writes the data from the array buffer, *source_array*, to the new array (or slice of the array)
3. Returns in the *array_id* variable the array ID of the new array.

Note that in both cases, a new array is created, and its array ID is returned in the *array_id* variable. The array is temporary until an UPDATE or INSERT statement is executed to associate the array with a particular column of a particular row.

You can make a single call to **isc_array_put_slice()** to write all the data you wish to the array. Or, you can call **isc_array_put_slice()** multiple times
to store data into various slices of the array. In this case, each call to **isc_array_put_slice()** after the first call should pass the array ID of the temporary array. When **isc_array_put_slice()** is called with the array ID of a temporary array, it copies the specified data to the specified slice of the temporary array (it will not create a new array), and it doesn't modify *array_id*.

Before calling **isc_array_put_slice()**, there are many operations you must do in order to fill in the array descriptor, *desc*, determine the appropriate internal array identifier, *array_id*, and fetch the rows whose array columns you want to access.

For complete step-by-step instructions for setting up an array descriptor and writing array information, see **Chapter 8, "Working with Array Data."**

Note Never execute a DSQL statement that tries to directly store data into an array column. The *only* way to access array column data is by calling **isc_array_get_slice()** or **isc_array_put_slice()**. The only supported array references in DSQL statements are ones that specify an entire array column (that is, just the column name) in order to get the internal identifier for the array, which is required by **isc_array_get_slice()** and **isc_array_put_slice()**.

Example The following program operates on a table named PROJ_DEPT_BUDGET. This table contains the quarterly head counts allocated for each project in each department of an organization. Each row of the table applies to a particular department and project. The quarterly head counts are contained in an array column named QUARTERLY_HEAD_CNT. Each table row has four elements in this column, one per quarter. Each element is a number of type *long*.

This program selects the rows containing 1994 information for the project named VBASE. For each such row, it calls **isc_array_get_slice()** to retrieve a slice of the array, the quarterly head counts for the last two quarters. It then increments each, and calls **isc_array_put_slice()** to store the updated values.

In addition to illustrating the usage of **isc_array_lookup_desc()**, **isc_array_get_slice(),** and **isc_array_put_slice()**, the program shows data structure initializations and calls to the DSQL functions required to prepare and execute the SELECT and UPDATE statements, to obtain the *array_id* needed by **isc_array_get_slice()** and **isc_array_put_slice()**, to fetch the selected rows one by one, and to update the array ID.

```
#include <ibase.h>

#define Return_if_Error(stat) if (stat[0] == 1 && stat[1]) \
                    { \
                    isc_print_status(stat); \
                    return(1); \
                    }

char *sel_str =
   "SELECT dept_no, quarterly_head_cnt FROM proj_dept_budget \
      WHERE year = 1994 AND proj_id = ’VBASE’";
char *upd_str =
   "UPDATE proj_dept_budget SET quarterly_head_count = ? \
   WHERE CURRENT OF S";

char dept_no[6];
long fetch_stat, SQLCODE, hcnt[2];
```

```
short len, i, flag0, flag1, flag2;
ISC_QUAD array_id;
ISC_ARRAY_DESC desc;
ISC_STATUS status_vector[20];
isc_stmt_handle stmt = NULL;
isc_stmt_handle ustmt = NULL;
char *cursor = "S";
XSQLDA *osqlda, *isqlda;

/* Set up the SELECT statement. */

/* Allocate the output XSQLDA for holding the array data. */
osqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(2));
osqlda->sqln = 2;
osqlda->version = SQLDA_VERSION1;

/* Allocate a statement handle for the SELECT statement. */
isc_dsql_allocate_statement(
   status_vector, &database_handle, &stmt);
Return_if_Error(status_vector);

/* Prepare the query for execution. */
isc_dsql_prepare(
   status_vector,
   &tr_handle,
   &stmt,
   0,
   sel_str,
   1,
   osqlda);
Return_if_Error(status_vector);

/* Set up an XSQLVAR structure to allocate space for each item
   to be retrieved. */

osqlda->sqlvar[0].sqldata = (char *) dept_no;
osqlda->sqlvar[0].sqltype = SQL_TEXT + 1;
osqlda->sqlvar[0].sqlind = &flag0;

osqlda->sqlvar[1].sqldata = (char *) &array_id;
osqlda->sqlvar[1].sqltype = SQL_ARRAY + 1;
osqlda->sqlvar[1].sqlind = &flag1;
```

```
/* Execute the SELECT statement. */
isc_dsql_execute(
   status_vector,
   &tr_handle,
   &stmt,
   1,
   NULL);
Return_if_Error(status_vector);

/* Declare a cursor. */
isc_dsql_set_cursor_name(
   status_vector, &stmt, cursor, 0);
Return_if_Error(status_vector);

/* Set up the UPDATE statement. */

/* Allocate a statement handle for the UPDATE statement. */
isc_dsql_allocate_statement(
   status_vector, &database_handle, &ustmt);
Return_if_Error(status_vector);

/* Allocate the input XSQLDA. */
isqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(2));
isqlda->sqln = 1;
1sqlda->version = SQLDA_VERSION1;

/* Prepare the UPDATE statement for execution. */
isc_dsql_prepare(
   status_vector,
   &tr_handle,
   &ustmt,
   0,
   upd_str,
   1,
   NULL);
Return_if_Error(status_vector);

/* Initialize the input XSQLDA. */
isc_dsql_describe_bind(
   status_vector, &ustmt, 1, isqlda);
Return_if_Error(status_vector);
```

```
/* Set up the input sqldata and sqlind fields. */
isqlda->sqlvar[0].sqldata = (char *) &array_id;
isqlda->sqlvar[0].sqlind = &flag2;

/* Set up the array descriptor. */
isc_array_lookup_desc(
   status_vector,
   &database_handle, /* Set by previous isc_attach_database() call. */
   &tr_handle, /* Set by previous isc_start_transaction() call. */
   "PROJ_DEPT_BUDGET", /* Table name. */
   "QUARTERLY_HEAD_CNT", /* Array column name. */
   &desc);
Return_if_Error(status_vector);

/* Set the descriptor bounds to those of the slice to be updated, that
is, to those of the last two elements. Assuming the array column was
defined to contain 4 elements, with a lower bound (subscript) of 1 and
an upper bound of 4, the last two elements are at subscripts 3 and 4. */
desc->array_desc_bounds[0].array_bound_lower = 3;
desc->array_desc_bounds[0].array_bound_upper = 4;

/* Fetch and process the rows of interest. */
while ((fetch_stat = isc_dsql_fetch(
   status_vector, &stmt, 1, osqlda)) == 0)
{
   if (!flag1)
   {
      /* There is array data; get values for last two quarters. */
      len = sizeof(hcnt);
      /* Fetch the data from the array slice into hcnt array. */
      isc_array_get_slice(
         status_vector,
         &database_handle,
         &tr_handle,
         &array_id,
         &desc,
         hcnt,
         &len);
      Return_if_Error(status_vector);

      /* Add 1 to each count. */
```

```
            for (i = 0; i < 2; i++)
               hcnt[i] = hcnt[i] + 1;

            /* Save new values. */
            isc_array_put_slice(
               status_vector,
               &database_handle,
               &tr_handle,
               &array_id,
               &desc,
               hcnt,
               &len);
            Return_if_Error(status_vector);

            /* Update the array ID. */
            isc_dsql_execute(
                  status_vector, &tr_handle, &ustmt, 1, isqlda);
            Return_if_Error(status_vector);

         };
      };
      if (fetch_stat != 100L)
      {
         SQLCODE = isc_sqlcode(status_vector);
         isc_print_sqlerror(SQLCODE, status_vector);
         return(1);
      }
```

Return Value **isc_array_put_slice()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to *isc_bad_stmt_handle*, *isc_bad_trans_handle*, or another InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_array_get_slice()**, **isc_array_lookup_bounds()**, **isc_array_lookup_desc()**, **isc_array_set_desc()**, **isc_dsql_allocate_statement()**, **isc_dsql_describe_bind()**, **isc_dsql_execute()**, **isc_dsql_fetch()**, **isc_dsql_prepare()**, **isc_dsql_set_cursor_name()**

isc_array_set_desc()

Initializes an array descriptor.

Syntax

```
ISC_STATUS isc_array_get_slice(
ISC_STATUS *status_vector,
char *table_name,
char *column_name,
short *sql_dtype,
short *sql_length,
short *dimensions,
ISC_ARRAY_DESC *desc);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
table_name	char *	Blank- or null-terminated name of the table containing the array column, *column_name*
column_name	char *	Name of the array column; this may be either null-terminated or blank-terminated
sql_dtype	short *	Pointer to SQL datatype of the array elements
sql_length	short *	Pointer to length of each array element
dimensions	short *	Pointer to number of array dimensions
desc	ISC_ARRAY_DESC *	Array descriptor to be filled in by this function

Description **isc_array_set_desc()** initializes the array descriptor, *desc*, from the function parameters, *table_name*, *column_name*, *sql_dtype*, *sql_length*, and *dimensions*.

isc_array_set_desc() also sets to 0 a flag in the descriptor. This specifies that the array is accessed in future function calls in row-major order, the default. If an application requires column-major access, reset this flag to 1.

table_name and *column_name* can be either null-terminated or blank-terminated. The names stored in the descriptor will be null-terminated.

sql_dtype must be given as an SQL macro constant.

The array descriptor is used in subsequent calls to **isc_array_get_slice()** or **isc_array_put_slice()**.

For a detailed description of the array descriptor, see **Chapter 8, “Working with Array Data.”**

Note There are ways to fill in an array descriptor other than by calling **isc_array_set_desc()**. You can also:

- Call **isc_array_lookup_bounds()**. This function is similar to **isc_array_lookup_desc()**, except that **isc_array_lookup_bounds()** also fills in information about the upper and lower bounds of each dimension.
- Call **isc_array_lookup_desc()**. This function is similar to **isc_array_lookup_bounds()**, except that **isc_array_lookup_desc()** does not fill in information about the upper and lower bounds of each dimension.
- Set the descriptor fields directly. Note that *array_desc_dtype* must be expressed as one of the datatypes in the following table, and the parameters, *array_desc_field_name*, and *array_desc_relation_name*, must be null-terminated:

array_desc_dtype	Corresponding InterBase datatype
blr_text	CHAR
blr_text2	CHAR
blr_short	SMALLINT
blr_long	INTEGER
blr_quad	ISC_QUAD structure
blr_float	FLOAT
blr_double	DOUBLE PRECISION
blr_date	DATE
blr_varying	VARCHAR
blr_varying2	VARCHAR
blr_blob_id	ISC_QUAD structure
blr_cstring	NULL-terminated string
blr_cstring2	NULL-terminated string

TABLE 12.15 Datatypes for array descriptor fields

Example The following illustrates a sample call to **isc_array_set_desc()**. More complete examples of accessing arrays are found in the example programs for **isc_array_get_slice()** and **isc_array_put_slice()**.

```
#include <ibase.h>
ISC_STATUS status_vector[20];
ISC_ARRAY_DESC desc;
short dtype = SQL_TEXT;
short len = 8;
short dims = 1;

isc_array_set_desc(
   status_vector,
   "TABLE1",
   "CHAR_ARRAY",
   &dtype,
   &len,
   &dims,
   &desc);
if (status_vector[0] == 1 && status_vector[1])
{
   /* Process error. */
   isc_print_status(status_vector);
   return(1);
}
```

Return Value **isc_array_set_desc()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_array_get_slice(), isc_array_lookup_bounds(), isc_array_lookup_desc(), isc_array_put_slice()**

isc_attach_database()

Attaches to an existing database.

Syntax

```
ISC_STATUS isc_attach_database(
ISC_STATUS *status_vector,
short db_name_length,
char *db_name,
isc_db_handle *db_handle,
short parm_buffer_length,
char *parm_buffer);
```

Parameter	**Type**	**Description**
status_vector	ISC_STATUS *	Pointer to the error status vector
db_name_length	short	Number of bytes in db_name string; if 0, the string is assumed to be null-terminated
db_name	char *	Database name
db_handle	*isc_db_handle* *	Pointer to a database handle set by this function; It is recommended that you set db_handle to NULL before passing it to **isc_attach_database()**
parm_buffer_length	short	Number of bytes in the database parameter buffer (DPB)
parm_buffer	char *	Address of the DPB

Description The **isc_attach_database()** function connects to an existing database to enable subsequent program access. It also optionally specifies various operational characteristics, such as a user name and password combination for access to a database on a remote server, or the number of database cache buffers to use. These optional characteristics are passed in a database parameter buffer (DPB) supplied and populated by the calling program, either through direct program construction, and by calling **isc_expand_dpb()** to build the DPB.

A program passes the name of the database file to which to attach in *db_name*. For programs not written in C, the program must also pass the length, in bytes, of *db_name* in the *db_name_length* parameter. C programs should pass a 0 length in this parameter.

If successful, **isc_attach_database()** assigns a unique ID to *db_handle*. Subsequent API calls use this handle to identify the database against which they operate.

When finished accessing a database, disconnect from the database with **isc_detach_database()**.

Example The following program fragment attaches to a database named *employee.db*. In the parameter buffer, it specifies a user name and password. These come from the contents of char * variables named *user_name* and *user_password*, respectively.

```
char dpb_buffer[256], *dpb, *p;
ISC_STATUS status_vector[20];
isc_db_handle handle = NULL;
short dpb_length;

/* Construct the database parameter buffer. */
dpb = dpb_buffer;
*dpb++ = isc_dpb_version1;

*dpb++ = isc_dpb_user_name;
*dpb++ = strlen(user_name);
for (p = user_name; *p;)
   *dpb++ = *p++;

*dpb++ = isc_dpb_password;
*dpb++ = strlen(user_password);
for (p = user_password; *p;)
   *dpb++ = *p++;
/* An alternate choice for the above construction is to call:
isc_expand_dpb(). */

dpb_length = dpb - dpb_buffer;

isc_attach_database(
   status_vector,
   0,
   "employee.db",
   &handle,
   dpb_length,
   dpb_buffer);
if (status_vector[0] == 1 && status_vector[1])
{
   /* An error occurred. */
   isc_print_status (status_vector);
   return(1);
}
```

Return Value **isc_attach_database()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_detach_database(), isc_expand_dpb()**

For more information about creating and populating a DPB, see **"Creating and populating a DPB" on page 42**. For more information about attaching to a database, see **"Connecting to databases" on page 40**.

isc_blob_default_desc()

Loads a data structure with default information about a Blob, including its subtype, character set, and segment size.

```
void isc_blob_default_desc(
     ISC_BLOB_DESC *desc,
     unsigned char *table_name,
     unsigned char *column_name);
```

Parameter	Type	Description
desc	ISC_BLOB_DESC *	Pointer to a Blob descriptor
table_name	unsigned char *	Table name
column_name	unsigned char *	Blob column name

Description **isc_blob_default_desc()** loads a Blob descriptor, *desc*, with the specified *table_name* and *column_name*, and the following default values prior to calling **isc_blob_gen_bpb()** to generate a Blob parameter buffer (BPB) for the Blob column being accessed:

- Subtype is set to TEXT.
- Character set is set to the default character set for the process or database.
- Segment size is set to 80 bytes.

isc_blob_default_desc() and three related functions, **isc_blob_gen_bpb()**, **isc_blob_lookup_desc()**, and **isc_blob_set_desc()**, provide dynamic access to Blob information. In particular, these functions can define and access information about a Blob for filtering purposes, such as character set information for text Blob data, and subtype information for text and non-text Blob data.

The following table lists the fields in the *desc* structure:

Parameter	**Type**	**Description**
blob_desc_subtype	short	Subtype of the Blob filter
blob_desc_charset	short	Character set being used
blob_desc_segment_size	short	Blob segment size
blob_desc_field_name [32]	char	Array containing the name of the Blob column
blob_desc_relation_name [32]	char	Array containing the name of the table in which the Blob is stored

TABLE 12.16 Blob descriptor fields

Example The following fragment loads the Blob descriptor with default information:

```
typedef struct
{
    short           blob_desc_subtype;
    short           blob_desc_charset;
    short           blob_desc_segment_size;
    unsigned char   blob_desc_field_name[32];
    unsigned char   blob_desc_relation_name[32];
ISC_BLOB_DESC;
isc_blob_default_desc(&desc, &relation, &field);
```

Return Value None.

See Also **isc_blob_gen_bpb()**, **isc_blob_lookup_desc()**, **isc_blob_set_desc()**

For more information about Blob descriptors, see **Chapter 7, “Working with Blob Data.”**

isc_blob_gen_bpb()

Generates a Blob parameter buffer (BPB) to allow dynamic access to Blob subtype and character set information.

Syntax

```
ISC_STATUS isc_blob_gen_bpb(
ISC_STATUS *status_vector,
ISC_BLOB_DESC *to_desc,
ISC_BLOB_DESC *from_desc,
unsigned short bpb_buffer_length,
unsigned char *bpb_buffer,
unsigned short *bpb_length);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
to_desc	ISC_BLOB_DESC *	Pointer to the target Blob descriptor
from_desc	ISC_BLOB_DESC *	Pointer to the source Blob descriptor
bpb_buffer_length	unsigned short	Length of the BPB bpb_buffer
bpb_buffer	unsigned char *	Pointer to the BPB
bpb_length	unsigned short *	Pointer to the length of the data stored into the BPB

Description **isc_blob_gen_bpb()** generates a Blob parameter buffer (BPB) from subtype and character set information stored in the source Blob descriptor *from_desc* and the target (destination) Blob descriptor *to_desc*.

A BPB is needed whenever a filter will be used when writing to or reading from a Blob column. Two Blob descriptors are needed for filtering: one (*from_desc*) to describe the filter source data, and the other (*to_desc*) to describe the destination. The descriptors must have been previously created either directly, or via a call to **isc_blob_default_desc()**, **isc_blob_lookup_desc()**, or **isc_blob_set_desc()**.

The BPB generated by **isc_blob_gen_bpb()** is subsequently needed in calls to **isc_open_blob2()** or **isc_create_blob2()** if filtering will be utilized. For more information about the BPB, see **Chapter 7, "Working with Blob Data."**

Example The following fragment generates the Blob descriptor:

```
isc_blob_gen_bpb(status, &to_desc, &from_desc, bpb_length, &buffer,
&buf_length);
```

Return Value **isc_blob_gen_bpb()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_blob_default_desc()**, **isc_blob_lookup_desc()**, **isc_blob_set_desc()**, **isc_create_blob2()**, **isc_open_blob2()**

isc_blob_info()

Returns information about an open Blob.

Syntax

```
ISC_STATUS isc_blob_info(
ISC_STATUS *status_vector,
isc_blob_handle *blob_handle,
short item_list_buffer_length,
char *item_list_buffer,
short result_buffer_length,
char *result_buffer);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
blob_handle	*isc_blob_handle* *	Pointer to the Blob
item_list_buffer_length	short	Length of the item-list buffer in which you specify the items for which you want information
item_list_buffer	char *	Pointer to the item-list buffer
result_buffer_length	short	Length of the result buffer into which InterBase returns the requested information
result_buffer	char *	Pointer to the result buffer

Description **isc_blob_info()** returns information about an existing Blob specified by *blob_handle*. The item-list buffer is an unstructured byte vector. An application lists the items about which it wants information in the item-list buffer.

InterBase returns the requested information to the result buffer as a series of *clusters* of information, one per item requested. Each cluster consists of three parts:

1. A one-byte *item type*. Each is the same as one of the item types in the item-list buffer.
2. A 2-byte number specifying the number of bytes that follow in the remainder of the cluster.
3. A *value*, stored in a variable number of bytes, whose interpretation depends on the item type.

A calling program is responsible for interpreting the contents of the result buffer and for deciphering each cluster as appropriate.

For a list of items that can be requested and returned, see **Chapter 7, "Working with Blob Data."**

Example The following example retrieves information about the current open Blob:

```
static char blob_items[] = {
   isc_info_blob_max_segment,
   isc_info_blob_num_segments,
   isc_info_blob_type};

CHAR blob_info[32];

isc_open_blob2(status_vector, &db, &tr_handle, &blob_handle,
&blob_id, blength, baddr)
if (status_vector[0] == 1 && status_vector[1])
{
   isc_print_status(status_vector);
   return(1);
}
isc_blob_info(status_vector, &blob_handle, sizeof(blob_items),
      blob_items, sizeof(blob_info), blob_info));
```

Return Value **isc_blob_info()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_create_blob2()**, **isc_open_blob2()**

isc_blob_lookup_desc()

Determines the subtype, character set, and segment size of a Blob, given a table name and Blob column name.

Syntax

```
ISC_STATUS isc_blob_lookup_desc(
ISC_STATUS *status_vector,
isc_db_handle **db_handle,
isc_tr_handle **trans_handle,
unsigned char *table_name,
unsigned char *column_name,
ISC_BLOB_DESC *desc,
unsigned char *global);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
db_handle	*isc_db_handle* **	Pointer to a database handle set by a previous call to **isc_attach_database**() *db_handle* returns an error in *status_vector* if it is NULL
trans_handle	*isc_tr_handle* **	Pointer to a transaction handle whose value has been set by a previous **isc_start_transaction()** call; *trans_handle* returns an error if NULL
table_name	unsigned char *	Name of the table containing the Blob column
column_name	unsigned char *	Name of the Blob column
desc	ISC_BLOB_DESC *	Pointer to the Blob descriptor to which the function returns information
global	unsigned char *	Global column name, returned by this function

Description **isc_blob_lookup_desc()** uses the system tables of a database to determine the subtype, character set, and segment size of a Blob given a table name and Blob column name.

isc_blob_lookup_desc() and three related functions, **isc_blob_default_desc()**, **isc_blob_gen_bpb()**, and **isc_blob_set_desc()** provide dynamic access to Blob information. In particular, you can use these functions to define and access information about Blob data for filtering purposes, such as character set information for text Blob data, and subtype information for text and non-text Blob data.

isc_blob_lookup_desc() stores the requested information about the Blob into the *desc* Blob descriptor structure. The following table describes the *desc* structure:

Parameter	Type	Description
blob_desc_subtype	short	Subtype of the Blob filter
blob_desc_charset	short	Character set being used
blob_desc_segment_size	short	Blob segment size
blob_desc_field_name [32]	char	Array containing the name of the Blob column
blob_desc_relation_name [32]	char	Array containing the name of the table in which the Blob is stored

TABLE 12.17 Blob descriptor fields

Example The following fragment retrieves information into a Blob descriptor:

```
isc_blob_lookup_desc(status, &db_handle, &tr_handle, &relation_name,
    &field_name, desc, &global);
```

Return Value **isc_blob_lookup_desc()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_blob_default_desc()**, **isc_blob_gen_bpb()**, **isc_blob_set_desc()**

For more information about Blob descriptors, see **Chapter 7, "Working with Blob Data."**

isc_blob_set_desc()

Sets the subtype and character set for a Blob.

Syntax

```
ISC_STATUS isc_blob_set_desc(
ISC_STATUS *status_vector,
unsigned char *table_name,
unsigned char *column_name,
short subtype,
short charset,
short segment_size,
ISC_BLOB_DESC *desc);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
table_name	unsigned char *	Name of the table containing the Blob column
column_name	unsigned char *	Name of the Blob column in the table
subtype	short	Specifies the subtype of the Blob; value are: • InterBase-defined subtype values, 0 or 1 (TEXT) • User-defined subtypes, –1 to –32768
charset	short	Specifies the character set for the Blob
segment_size	short	Specifies the segment size for the Blob
desc	ISC_BLOB_DESC *	Pointer to a Blob descriptor to populate.

Description **isc_blob_set_desc()** sets the Blob column name, table name, subtype, segment size, and character set for a Blob column to values specified by the application. To set these values to InterBase defaults, use **isc_blob_default_desc()**.

isc_blob_set_desc() and three related functions, **isc_blob_default_desc()**, **isc_blob_gen_bpb()**, and **isc_blob_lookup_desc()** provide dynamic access to Blob data. In particular, you can use these functions to define and access information about Blob data for filtering purposes, such as character set information for text Blob data, and subtype information for text and non-text Blob data.

You can manually set the subtype and character set information (for a TEXT subtype) in a Blob descriptor, by way of a call to **isc_blob_set_desc()**. Pass the subtype, character set, and segment size to the Blob descriptor in your application.

isc_blob_set_desc() is useful for setting the contents of the Blob descriptor without querying the system tables for the information. Calls to this function also let an application specify character set and subtype for custom filtering operations.

Note Do not call this function while running against a V3.x database.

Example The following example sets the default values for a tour guide application, including subtype, character set, and segment size:

```
isc_blob_set_desc(status, "TOURISM", "GUIDEBOOK", 1, 2, 80, &desc);
```

Return Value **isc_blob_set_desc()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_blob_default_desc()**, **isc_blob_gen_bpb()**, **isc_blob_lookup_desc()**

For more information about Blob descriptors, see **Chapter 7, "Working with Blob Data."**

isc_cancel_blob()

Discards a Blob, frees internal storage used by the Blob, and sets the Blob handle to NULL.

Syntax

```
ISC_STATUS isc_cancel_blob(
ISC_STATUS *status_vector,
isc_blob_handle *blob_handle);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
blob_handle	*isc_blob_handle* *	Pointer to the handle for the Blob you want to cancel; sets the handle to zero and returns a successful result even if the handle is NULL.

Description InterBase temporarily stores Blob data in the database during create operations. If, for some reason, you do not, or cannot, close a Blob, the storage space remains allocated in the database and InterBase does not set the handle to NULL. Call **isc_cancel_blob()** to release the temporary storage in the database, and to set *blob_handle* to NULL. If you close the Blob in the normal course of your application processing logic, this step is unnecessary as InterBase releases system resources on a call to **isc_close_blob()**.

Note A call to this function does not produce an error when the handle is NULL. Therefore, it is good practice to call **isc_cancel_blob()** before creating or opening a Blob to clean up existing Blob operations.

Example The following fragment cancels any open Blob before creating a new one:

```
isc_cancel_blob(status_vector, &blob_handle);
if (status_vector[0] == 1 && status_vector[1])
{
    /* process error */
    isc_print_status(status_vector);
    return(1);
}
isc_create_blob(status_vector, &DB, &trans, &blob_handle, &blob_id)
```

Return Value **isc_cancel_blob()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_close_blob()**

isc_cancel_events()

Cancels an application's interest in asynchronous notification of any of a specified group of events.

Syntax

```
ISC_STATUS isc_cancel_events(
ISC_STATUS *status_vector,
isc_db_handle *db_handle,
ISC_LONG *event_id);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
db_handle	*isc_db_handle* *	Pointer to a database handle set by a previous call to **isc_attach_database()**; the handle identifies the database for which the event watch is to be canceled. *db_handle* returns an error in *status_vector* if it is NULL
event_id	ISC_LONG *	Pointer to the event or events to cancel; set by a previous call to **isc_que_events()**

Description **isc_cancel_events()** cancels an application program's asynchronous wait for any of a specified list of events. The events are the ones that were associated with *event_id* as a result of a previous call to **isc_que_events()**.

Example The following call cancels a program's wait for events associated with *event_id*, where *event_id* was previously returned from a call to **isc_que_events()**:

```
isc_cancel_events(status_vector, &database_handle, &event_id);
```

A more complete example is provided in the section on **isc_que_events()**.

Return Value **isc_cancel_events()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_que_events()**

isc_close_blob()

Closes an open Blob, which involves flushing any remaining segments, releasing system resources associated with Blob update or retrieval, and setting the Blob handle to zero.

Syntax

```
ISC_STATUS isc_close_blob(
ISC_STATUS *status_vector,
isc_blob_handle *blob_handle);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
blob_handle	*isc_blob_handle* *	Pointer to the handle of the Blob to close

Description **isc_close_blob()** is used to store a Blob in the database and clean up after Blob operations. Close any Blob after reading from or writing to it. If, for some reason, your application does not close a Blob, you can lose data. If your application might open a Blob without closing it then you should call **isc_cancel_blob()** to make sure that the application does not try to open a
Blob that is already open.

blob_handle is set by a call to **isc_create_blob2()** or to **isc_open_blob2()**.

Example The following example closes a Blob and frees system resources:

```
if (status_vector[1] == isc_segstr_eof)
   isc_close_blob(status_vector, &blob_handle)
```

Return Value **isc_close_blob()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_cancel_blob()**, **isc_create_blob2()**, **isc_open_blob2()**

isc_commit_retaining()

Commits an active transaction and retains the transaction context after a commit.

Syntax

```
ISC_STATUS isc_commit_retaining(
ISC_STATUS *status_vector,
isc_tr_handle *trans_handle);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
trans_handle	*isc_tr_handle* *	Pointer to a transaction handle whose value has been set by a previous **isc_start_transaction()** call; *trans_handle* returns an error if NULL

Description **isc_commit_retaining()** commits an active transaction and immediately clones itself. This means that the function retains the transaction name, system resources associated with the transaction, and the current state of any open cursors in the transaction. Although the function is actually initiating a new transaction, by assigning the new transaction the active transaction handle it is, in effect, keeping the transaction open across commits. This results in improved performance by allowing an application to minimize the overhead of initiating additional transactions. **isc_commit_retaining()** allows you to commit updates while keeping a cursor open.

You can initiate a rollback within the active transaction but the rollback only affects uncommitted updates. In other words, a rollback is legal, even after the transaction context has been passed to the cloned transaction, but, in that case, the rollback will only affect the updates your application has made to the database since the last commit.

To audit the commits made by your calls to this function, check the first element in the status vector to see if the call was successful. If this element contains a zero, the call was successful.

The transaction ends when you commit without using the retention feature, such as with a call to **isc_commit_transaction()**, or when you roll back with **isc_rollback_transaction()**.

Example The following call commits a transaction, prints a message, and starts a new transaction with the same handle within the same request:

```
if (!isc_commit_retaining(status, &retained_trans))
{
   fprintf("Committed and retained\\n");
```

```
    isc_print_status(status);
}
```

The following call commits a transaction, prints a confirmation message, starts a new transaction with the same handle within the same request, or, if the commit fails, prints an error message and rolls back.

```
isc_commit_retaining(status, &retained_trans);
if (status[0] == 1 && status[1])
{
    fprintf("An error occurred during commit, rolling back.");
    rb_status = isc_rollback_transaction(status, &retained_status);
}
else
{
    fprintf("Commit successful.");
    tr_count++; /*Increments the number of commits. */
}
```

Return Value **isc_commit_retaining()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_commit_transaction()**, **isc_rollback_transaction()**, **isc_start_transaction()**

isc_commit_transaction()

Commits a specified active transaction.

Syntax

```
ISC_STATUS isc_commit_transaction(
ISC_STATUS *status_vector,
isc_tr_handle *trans_handle);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
trans_handle	*isc_tr_handle* *	Pointer to a transaction handle whose value has been set by a previous **isc_start_transaction()** call; *trans_handle* returns an error if NULL

Description **isc_commit_transaction()** closes record streams, frees system resources, and sets the transaction handle to zero for the specified transaction.

When you call this function to execute a commit operation against multiple databases, InterBase first initiates a call to the **isc_prepare_transaction()** function. **isc_prepare_transaction()** executes the first phase of a two-phase commit. This puts the transaction into limbo and signals your intention to commit, so that InterBase can poll all target databases to verify that they are ready to accept the commit. Also, **isc_commit_transaction()** writes a Blob message to the RDB$TRANSACTION_DESCRIPTION column of the RDB$TRANSACTIONS system table, detailing information required by InterBase to perform a reconnect in case of system failure during the commit process.

The **isc_commit_transaction()** function also performs the second phase of a two-phase commit upon receiving verification that all databases are ready to accept the commit. Also, **isc_commit_transaction()** cleans up RDB$TRANSACTIONS.

Example The following call commits a transaction and prints a message:

```
isc_commit_transaction(status, &trans);
if (status[0] == 1 && status[1])
{
   fprintf("Error on write\\n");
   isc_print_status(status);
}
```

Return Value **isc_commit_transaction()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_commit_retaining()**, **isc_prepare_transaction()**

isc_create_blob2()

Creates and opens the Blob for write access, and optionally specifies the filters to be used to translate the Blob from one subtype to another.

Syntax

```
ISC_STATUS isc_create_blob2(
ISC_STATUS *status_vector,
isc_db_handle *db_handle,
isc_tr_handle *trans_handle,
isc_blob_handle *blob_handle,
ISC_QUAD *blob_id,
short bpb_length,
char *bpb_address);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
db_handle	*isc_db_handle* *	Pointer to a database handle set by a previous call to **isc_attach_database()** *db_handle* returns an error in *status_vector* if it is NULL
trans_handle	*isc_tr_handle* *	Pointer to the handle of the transaction in which you want the Blob to be created
blob_handle	*isc_blob_handle* *	Pointer to the Blob handle
blob_id	ISC_QUAD *	Pointer to the 64-bit system-defined Blob ID, which is stored in a field in the table and points to the first segment of the Blob or to a page of pointers to Blob fragments
bpb_length	short	Length of the Blob parameter buffer (BPB)
bpb_address	char *	Pointer to the BPB

Description **isc_create_blob2()** creates a context for storing a Blob, opens a Blob for write access, and optionally specifies the filters used to translate from one Blob format to another. Subsequent calls to **isc_put_segment()** write data from an application buffer to the Blob.

If a Blob filter is used, it is called for each segment written to the Blob. InterBase selects the filter to be used based on the source and target subtypes specified in a previously populated Blob parameter buffer (BPB), pointed to by *bpb_address*.

Note Blob filters are not supported on Netware.

If a Blob filter is not needed or cannot be used, a BPB is not needed; pass 0 for *bpb_length* and NULL for *bpb_address*.

The Blob handle pointed to by *blob_handle* must be zero when **isc_create_blob2()** is called. To reuse *blob_handle*, close the Blob with a call to **isc_close_blob()** to zero out the handle before calling **isc_create_blob2()**.

On success, **isc_create_blob2()** assigns a unique ID to *blob_handle*, and a Blob identifier to *blob_id*. Subsequent API calls require one or both of these to identify the Blob against which they operate.

After a blob is created, data can be written to it by a sequence of calls to **isc_put_segment()**. When finished writing to the Blob, close it with **isc_close_blob()**.

When you create a Blob, it is essentially an "orphan" until you assign its *blob_id* to a particular Blob column of a particular row of a table. You do this, after closing the Blob, by using DSQL to execute either an INSERT statement to insert a new row containing the Blob (and any other columns desired), or an UPDATE statement to replace an existing Blob with the new one.

For more information about BPBs and Blob filters, see **Chapter 7, "Working with Blob Data."**

Example The following fragment declares a BPB, populates it with filter information, then creates a Blob and passes the BPB:

```
isc_blob_handle blob_handle; /* declare at beginning */
ISC_QUAD blob_id; /* declare at beginning */
char bpb[] = {
   isc_bpb_version1,
   isc_bpb_target_type,
   1,     /* # bytes that follow which specify target subtype */
   1,     /* target subtype (TEXT) */
   isc_bpb_source_type,
   1,     /* # bytes that follow which specify source subtype */
   -4,    /* source subtype*/
   };
```

```
. . .

isc_create_blob2(
    status_vector,
    &db_handle,
    &tr_handle,
    &blob_handle,     /* to be filled in by this function */
    &blob_id,         /* to be filled in by this function */
    actual_bpb_length, /* length of BPB data */
    &bpb              /* Blob parameter buffer */
    )
```

Return Value **isc_create_blob2()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_blob_gen_bpb()**, **isc_open_blob2()**, **isc_put_segment()**

isc_database_info()

Reports requested information about a previously attached database.

Syntax

```
ISC_STATUS isc_database_info(
ISC_STATUS *status_vector,
isc_db_handle *db_handle,
short item_list_buffer_length,
char *item_list_buffer,
short result_buffer_length,
char *result_buffer);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector.
db_handle	*isc_db_handle* *	Pointer to a database handle set by a previous call to **isc_attach_database()** *db_handle* returns an error in *status_vector* if it is NULL
item_list_buffer_length	short	Number of bytes in the item-list buffer.
item_list_buffer	char *	Address of the item-list buffer.
result_buffer_length	short	Number of bytes in the result buffer.
result_buffer	char *	Address of the result buffer.

Description isc_database_info() returns information about an attached database. Typically, **isc_database_info()** is called to:

- Determine how much space is used for page caches. The space is the product of the number of buffers and the page size, which are determined by calling **isc_database_info()** with the *isc_info_num_buffers* and *isc_info_page_size* item-list options.
- Monitor performance. For example, to compare the efficiency of two update strategies, such as updating a sorted or unsorted stream.

The calling program passes its request for information through the item-list buffer supplied by the program, and InterBase returns the information to a program-supplied result buffer.

Example The following program fragment requests the page size and the number of buffers, then examines the result buffer to retrieve the values supplied by the InterBase engine:

```
char db_items[] = {
   isc_info_page_size, isc_info_num_buffers,
   isc_info_end};
char res_buffer[40], *p, item;
int length;
SLONG page_size = 0L, num_buffers = 0L;
ISC_STATUS status_vector[20];

isc_database_info(
   status_vector,
   &handle,  /* Set in previous isc_attach_database() call. */
   sizeof(db_items),
   db_items,
   sizeof(res_buffer),
   res_buffer);
if (status_vector[0] == 1 && status_vector[1])
{
   /* An error occurred. */
   isc_print_status(status_vector);
   return(1);
};
/* Extract the values returned in the result buffer. */
for (p = res_buffer; *p != isc_info_end ;)
{
   item = *p++;
   length = isc_vax_integer (p, 2);
   p += 2;
   switch (item)
   {
      case isc_info_page_size:
         page_size = isc_vax_integer (p, length);
         break;
      case isc_info_num_buffers:
         num_buffers = isc_vax_integer (p, length);
         break;
      default:
         break;
   }
   p += length;
};
```

Return Value **isc_database_info()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_attach_database()**, **isc_detach_database()**

For more information about requesting database attachment information, see **"Requesting information about an attachment" on page 49**.

isc_decode_date()

Translates a date from InterBase ISC_QUAD format into the C *tm* format.

Syntax

```
void isc_decode_date(
ISC_QUAD *ib_date,
void *tm_date);
```

Parameter	Type	Description
ib_date	ISC_QUAD *	Pointer to an eight-byte ISC_QUAD structure containing a date in InterBase format.
tm_date	void *	Pointer to a C *tm* structure.

Description **isc_decode_date()** translates a date retrieved from a table and stored in an ISC_QUAD variable, *ib_date*, into a C time structure for program manipulation. Both *ib_date* and *tm_date* must be declared and initialized before use.

Use the **isc_dsql** family of API calls to retrieve InterBase DATE data from a table into the ISC_QUAD structure prior to translation.

Example The following code fragment illustrates declaring time structures and calling **isc_decode_date()** to translate an InterBase date format into a C time format:

```
#include <time.h>
#include <ibase.h>
. . .
struct tm hire_time;
ISC_QUAD hire_date;
. . .
```

```
/* Retrieve DATE data from a table here. */
. . .
isc_decode_date(&hire_date, &hire_time);
```

Return Value None.

See Also **isc_encode_date()**

isc_delete_user( )

Deletes a user record from the password database, *isc4.gdb*.

Syntax

```
ISC_STATUS isc_delete_user(
ISC_STATUS *status
USER_SEC_DATA *user_sec_data);
```

Parameter	Type	Description
status vector	ISC_STATUS *	Pointer to the error status vector.
user_sec_data	USER_SEC_DATA *	Pointer to a struct that is defined in *ibase.h*.

Description The three security functions, **isc_add_user()**, **isc_delete_user()**, and **isc_modify_user()** mirror functionality that is available in the **gsec** command-line utility. **isc_delete_user()** deletes a record from *isc4.gdb*, InterBase's password database.

At a minimum, you must provide the user name. If the server is not local, you must provide both a server name and a protocol. Valid choices for the protocol field are *sec_protocol_tcpip*, *sec_protocol_netbeui*, *sec_protocol_spx*, and *sec_protocol_local*.

InterBase reads the settings for the ISC_USER and ISC_PASSWORD environment variables if you do not provide a DBA user name and password.

The definition for the USER_SEC_DATA struct in *ibase.h* is as follows:

```
typedef struct {
   short  sec_flags;      /* which fields are specified */
   int    uid;            /* the user's id */
   int    gid;            /* the user's group id */
   int    protocol;       /* protocol to use for connection */
   char   *server;        /* server to administer */
   char   *user_name;     /* the user's name */
   char   *password;      /* the user's password */
   char   *group_name;     /* the group name */
   char   *first_name;     /* the user’s first name */
   char   *middle_name;    /* the user’s middle name */
   char   *last_name;      /* the user’s last name */
   char   *dba_user_name;  /* the dba user name */
   char   *dba_password;   /* the dba password */
} USER_SEC_DATA;
```

When you pass this struct to one of the three security functions, you can tell it which fields you have specified by doing a bitwise OR of the following values, which are defined in *ibase.h*:

```
sec_uid_spec               0x01
sec_gid_spec               0x02
sec_server_spec            0x04
sec_password_spec          0x08
sec_group_name_spec        0x10
sec_first_name_spec        0x20
sec_middle_name_spec       0x40
sec_last_name_spec         0x80
sec_dba_user_name_spec     0x100
sec_dba_password_spec      0x200
```

No bit values are available for user name and password, since they are required.

The following error messages exist for this function:

Code	Value	Description
isc_usrname_too_long	335544747	The user name passed in is greater than 31 bytes
isc_password_too_long	335544748	The password passed in is longer than 8 bytes
isc_usrname_required	335544749	The operation requires a user name
isc_password_required	335544750	The operation requires a password
isc_bad_protocol	335544751	The protocol specified is invalid
isc_dup_usrname_found	335544752	The user name being added already exists in the security database.

TABLE 12.18 Error messages for user security functions

Code	Value	Description
isc_usrname_not_found	335544753	The user name was not found in the security database
isc_error_adding_sec_record	335544754	An unknown error occurred while adding a user
isc_error_deleting_sec_record	335544755	An unknown error occurred while deleting a user
isc_error_modifying_sec_record	335544756	An unknown error occurred while modifying a user
isc_error_updating_sec_db	335544757	An unknown error occurred while updating the security database

TABLE 12.18 Error messages for user security functions

Example The following example deletes a user ("Socks") from the password database, using the bitwise OR technique for passing values from the USER_SEC_DATA struct.

```
{
   ISC_STATUS status[20];
   USER_SEC_DATA sec;

   sec.server          = "kennel";
   sec.dba_user_name   = "sysdba";
   sec.dba_password    = "masterkey";
   sec.protocol        = sec_protocol_tcpip;
   sec.user_name       = "socks";
   sec.sec_flags       = sec_server_spec
                       | sec_dba_user_name_spec
                       | sec_dba_password_name_spec;

   isc_delete_user(status, &sec);
   /* check status for errors */
   if (status[0] == 1 && status[1])
   {
      switch (status[1]) {
      case isc_usrname_too_long:
         printf("Security database cannot accept long user names\n");
         break;
      ...
      }
   }
}
```

Return Value **isc_delete_user()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. See the "Description" section for this function for a list of error codes. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_add_user( )**, **isc_modify_user( )**

isc_detach_database()

Detaches from a database previously connected with **isc_attach_database()**.

Syntax

```
ISC_STATUS isc_detach_database(
ISC_STATUS *status_vector,
isc_db_handle *db_handle);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
db_handle	*isc_db_handle* *	Pointer to a database handle set by a previous call to **isc_attach_database()** *db_handle* returns an error in *status_vector* if it is NULL

Description **isc_detach_database()** detaches an attached database. Call this function to release system resources when you are done using a database or before re-attaching the database with different attach parameters. **isc_detach_database()** also releases the buffers and structures that control the remote interface on the client and the remote server where the database is stored.

Before calling **isc_detach_database()** commit or roll back transactions affecting the database from which you want to detach.

Example The following conditional statement detaches a database:

```
if (handle)
   isc_detach_database(status_vector, &handle);
```

Assuming that *handle* is valid and identifies an attached database, the specified database is detached when this statement executes.

Return Value **isc_detach_database()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, “Handling Error Conditions.”**

See Also **isc_attach_database()**

isc_drop_database()

Deletes a currently attached database and all of its supporting files, such as secondary database files, write-ahead log files, and shadow files.

Syntax

```
ISC_STATUS isc_drop_database(
ISC_STATUS *status_vector,
isc_db_handle *db_handle);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
db_handle	*isc_db_handle* *	Pointer to a database handle set by a previous call to **isc_attach_database()**; the handle identifies the database containing the array column *db_handle* returns an error in *status_vector* if it is NULL

Description **isc_drop_database()** deletes an attached database and all of its supporting files. Call this routine when you no longer have a use for the database (for example, if you moved all the data into another database, or if the database was just temporary and is no longer needed). To succeed, **isc_drop_database()** must be issued when no other processes are attached to the database.

Example The following conditional statement drops a database:

```
if (handle)
   isc_drop_database(status_vector, &handle);
```

Assuming that *handle* is valid and identifies an attached database, the specified database is dropped when this statement executes.

Return Value **isc_drop_database()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, “Handling Error Conditions.”**

See Also **isc_attach_database()**

isc_dsql_allocate_statement()

Allocates a statement handle for subsequent use with other API dynamic SQL (DSQL) calls.

Syntax

```
ISC_STATUS isc_dsql_allocate_statement(
ISC_STATUS *status_vector,
isc_db_handle *db_handle,
isc_stmt_handle *stmt_handle);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
db_handle	*isc_db_handle* *	Pointer to a database handle set by a previous call to **isc_attach_database()** *db_handle* returns an error in *status_vector* if it is NULL
stmt_handle	*isc_stmt_handle* *	Pointer to the statement handle to be allocated by this function; the handle must be NULL when this function is called, or an error is returned in status_vector

Description **isc_dsql_allocate_statement()** allocates a statement handle and returns a pointer to it in *stmt_handle*. This pointer is passed to **isc_dsql_prepare()** to associate the statement handle with a particular DSQL statement for processing.

If a DSQL statement is to be executed multiple times, or if it returns output (other than the results from a stored procedure), **isc_dsql_allocate_statement()** or **isc_dsql_alloc_statement2()** should be called to allocate a statement handle prior to preparing and executing the statement with **isc_dsql_prepare()** and **isc_dsql_execute()**.

Note The function, **isc_dsql_allocate_statement()**, is very similar to the function, **isc_dsql_alloc_statement2()** except that statement handles allocated using **isc_dsql_allocate_statement()** are *not* automatically reset to NULL when the database under which they are allocated is detached. To reset statement handles automatically, use **isc_dsql_alloc_statement2()**.

When you are done processing a statement, the statement handle can be freed with the **isc_dsql_free_statement()** or by calling **isc_detach_database()**.

Example The following program fragment allocates a statement handle for an SQL statement that will access the database referenced by the database handle, *database_handle*:

```
ISC_STATUS status_vector[20];
isc_stmt_handle statement_handle;

statement_handle = NULL; /* Set handle to NULL before allocating it. */
isc_dsql_allocate_statement(
   status_vector,
   &database_handle, /* Set in previous isc_attach_database() call. */
   &statement_handle);
if (status_vector[0] == 1 && status_vector[1])
{
   isc_print_status(status_vector); /* Display error message. */
   return(1); /* Return now. */
}
   /*  Call other functions to associate a particular SQL statement
with the statement handle, and to do other operations necessary to
prepare and execute the DSQL statement. Free the statement handle when
it is no longer needed. */
```

Return Value **isc_dsql_allocate_statement()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to *isc_bad_stmt_handle*, *isc_bad_db_handle*, or another InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_dsql_alloc_statement2()**, **isc_dsql_execute()**, **isc_dsql_free_statement()**, **isc_dsql_prepare()**

isc_dsql_alloc_statement2()

Allocates a statement handle for subsequent use with other API dynamic SQL (DSQL) calls.

Syntax

```
ISC_STATUS isc_dsql_alloc_statement2(
ISC_STATUS *status_vector,
isc_db_handle *db_handle,
isc_stmt_handle *stmt_handle);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
db_handle	*isc_db_handle* *	Pointer to a database handle set by a previous call to **isc_attach_database()**; the handle identifies the database containing the array column *db_handle* returns an error in *status_vector* if it is NULL
stmt_handle	*isc_stmt_handle* *	Pointer to the statement handle to be allocated by this function; the handle must be NULL when this function is called, or an error is returned in status_vector

Description **isc_dsql_alloc_statement2()** allocates a statement handle and returns a pointer to it in *stmt_handle*. This pointer is passed to **isc_dsql_prepare()** to associate the statement handle with a particular DSQL statement for processing.

If a DSQL statement is to be executed multiple times, or if it returns output (other than the results from a stored procedure), **isc_dsql_alloc_statement2()** or **isc_dsql_allocate_statement()** should be called to allocate a statement handle prior to preparing and executing the statement with **isc_dsql_prepare()** and **isc_dsql_execute()**.

Note The function, **isc_dsql_allocate_statement2()**, is very similar to the function**,** **isc_dsql_alloc_statement()** except that statement handles allocated using **isc_dsql_allocate_statement2()** are automatically reset to NULL when the database under which they are allocated is detached.

Example The following program fragment allocates a statement handle for an SQL statement that will access the database referenced by the database handle, *database_handle*:

```
ISC_STATUS status_vector[20];
isc_stmt_handle statement_handle;

isc_dsql_alloc_statement2(
    status_vector,
    &database_handle, /* Set in previous isc_attach_database() call. */
    &statement_handle);
if (status_vector[0] == 1 && status_vector[1])
{
    isc_print_status(status_vector); /* Display an error message. */
    return(1); /* Return now. */
}
    /*  Call other functions to associate a particular SQL statement
with the statement handle, and to do other operations necessary to
prepare and execute the DSQL statement. */
    ;
```

Return Value **isc_dsql_alloc_statement2()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to *isc_bad_stmt_handle*, *isc_bad_db_handle*, or another InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_dsql_allocate_statement()**, **isc_dsql_execute()**, **isc_dsql_free_statement()**, **isc_dsql_prepare()**

isc_dsql_describe()

Provides information about columns retrieved by the execution of a DSQL SELECT or EXECUTE PROCEDURE statement.

Syntax

```
ISC_STATUS isc_dsql_describe(
ISC_STATUS *status_vector,
isc_stmt_handle *stmt_handle,
unsigned short dialect,
XSQLDA *xsqlda);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
stmt_handle	*isc_stmt_handle* *	Pointer to a statement handle previously allocated with **isc_dsql_allocate_statement()** or **isc_dsql_alloc_statement2()**; the handle returns an error in status_vector if it is NULL
dialect	unsigned short	Indicates the version of the SQL descriptor area passed to the function; set this value to 1
xsqlda	XSQLDA *	Pointer to a previously allocated XSQLDA used for output

Description **isc_dsql_describe()** stores into *xsqlda* a description of the columns that make up the rows returned for a SELECT statement, or a description of the result values returned by an EXECUTE PROCEDURE statement. These statements must have been previously prepared for execution with **isc_dsql_prepare()**, before **isc_dsql_describe()** can be called.

Note Using **isc_dsql_describe()** is not necessary unless a previously issued **isc_dsql_prepare()** function indicates that there is insufficient room in the output XSQLDA for the return values of the DSQL statement to be executed.

Example The following program fragment illustrates a sequence of calls which allocates an XSQLDA, prepares a statement, checks whether or not the appropriate number of XSQLVARs was allocated, and corrects the situation if needed.

```
#include <ibase.h>
ISC_STATUS status_vector[20];
XSQLDA *osqlda;
int n;
char *query =
   "SELECT * FROM CITIES WHERE STATE = "NY" ORDER BY CITY DESCENDING";
```

```
osqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(3);
osqlda->version = SQLDA_VERSION1;
osqlda->sqln = 3;

isc_dsql_prepare(
   status_vector,
   &tr_handle, /* Set in previous isc_start_transaction() call. */
   &stmt_handle,
    /* Allocated previously by isc_dsql_allocate_statement()
      or isc_dsql_alloc_statement2() call. */
   0,
   query,
   1,
   osqlda);
if (status_vector[0] == 1 && status_vector[1])
{
   /* Process error. */
   isc_print_status(status_vector);
   return(1);
}

if (osqlda->sqld > osqlda->sqln) /* Need more XSQLVARS. */
{
   n = osqlda->sqld;
   free(osqlda);
   osqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(n);
   osqlda->sqln = n;
   osqlda->version = SQLDA_VERSION1;
   isc_dsql_describe(
      status_vector,
      &stmt_handle,
      1,
      osqlda);
   if (status_vector[0] == 1 && status_vector[1])
   {
      /* Process error. */
      isc_print_status(status_vector);
      return(1);
   }
}
```

Return Value **isc_dsql_describe()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to *isc_bad_stmt_handle*, or another InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_dsql_describe_bind()**, **isc_dsql_execute()**, **isc_dsql_execute2()**, **isc_dsql_prepare()**

For more information about preparing a DSQL statement with return values, see **"DSQL programming methods" on page 94**. For more information about creating and populating the XSQLDA, see **"Understanding the XSQLDA" on page 83**.

isc_dsql_describe_bind()

Provides information about dynamic input parameters required by a previously prepared DSQL statement.

Syntax

```
ISC_STATUS isc_dsql_describe_bind(
ISC_STATUS *status_vector,
isc_stmt_handle *stmt_handle,
unsigned short dialect,
XSQLDA *xsqlda);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
stmt_handle	*isc_stmt_handle* *	Pointer to a statement handle previously allocated with **isc_dsql_allocate_statement()** or **isc_dsql_alloc_statement2()**; the handle returns an error in status_vector if it is NULL
dialect	unsigned short	Indicates the version of the SQL descriptor area passed to the function; set this value to 1
xsqlda	XSQLDA *	Pointer to a previously allocated XSQLDA used for input

Description **isc_dsql_describe_bind()** stores into the input XSQLDA *xsqlda* information about the dynamic input parameters required by a DSQL statement previously prepared with **isc_dsql_prepare()**.

Before an application can execute a statement with input parameters, it must supply values for them in an input XSQLDA structure. If you know exactly how many parameters are required, and their datatypes, you can set up the XSQLDA directly without calling **isc_dsql_describe_bind()**. But if you need InterBase to analyze the statement and provide information such as the number of parameters and their datatypes, you must call **isc_dsql_describe_bind()** to supply the information.

Example The following program fragment illustrates a sequence of calls that allocates an input XSQLDA, prepares a DSQL UPDATE statement, calls the function **isc_dsql_describe_bind()**, checks whether or not the appropriate number of XSQLVARs was allocated, and corrects the situation if necessary.

```
#include <ibase.h>
ISC_STATUS status_vector[20];
XSQLDA *isqlda
int n;
char *str = "UPDATE DEPARTMENT SET BUDGET = ?, LOCATION = ?";

isc_dsql_prepare(
   status_vector,
   &tr_handle, /* Set in previous isc_start_transaction() call. */
   &stmt_handle,
   /* Allocated previously by isc_dsql_allocate_statement()
      or isc_dsql_alloc_statement2() call. */
   0,
   str,
   1,
   NULL);
if (status_vector[0] == 1 && status_vector[1])
{
    /* Process error. */
   isc_print_status(status_vector);
   return(1);
}
/* Allocate an input XSQLDA. */
isqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(1);
isqlda->version = SQLDA_VERSION1;
isqlda->sqln = 1;
isc_dsql_describe_bind(
   status_vector,
   &stmt_handle,
   /* Allocated previously by isc_dsql_allocate_statement()
      or isc_dsql_alloc_statement2() call. */
   1,
   isqlda);
if (status_vector[0] == 1 && status_vector[1])
{
   /* Process error. */
   isc_print_status(status_vector);
   return(1);
}
if (isqlda->sqld > isqlda->sqln) /* Need more XSQLVARs. */
{
   n = isqlda->sqld;
   free(isqlda);
   isqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(n);
   isqlda->sqln = n;
   isqlda->version = SQLDA_VERSION1;
   isc_dsql_describe_bind(
           status_vector,
           &stmt_handle,
           1,
           isqlda);
   if (status_vector[0] == 1 && status_vector[1])
   {
        /* Process error. */
      isc_print_status(status_vector);
      return(1);
   }
}
```

Return Value **isc_dsql_describe_bind()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to *isc_bad_stmt_handle*, or another InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_dsql_describe()**, **isc_dsql_execute()**, **isc_dsql_execute2()**, **isc_dsql_prepare()**

For more information about preparing a DSQL statement with input parameters, see **"DSQL programming methods" on page 94**. For more information about creating and populating the XSQLDA, see **"Understanding the XSQLDA" on page 83**.

isc_dsql_execute()

Executes a previously prepared DSQL statement.

Syntax

```
ISC_STATUS isc_dsql_execute(
ISC_STATUS *status_vector,
isc_tr_handle *trans_handle,
isc_stmt_handle *stmt_handle,
unsigned short dialect,
XSQLDA *xsqlda);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
trans_handle	*isc_tr_handle* *	Pointer to a transaction handle whose value has been set by a previous **isc_start_transaction()** call; *trans_handle* returns an error if NULL
stmt_handle	*isc_stmt_handle* *	Pointer to a statement handle previously allocated with **isc_dsql_allocate_statement()** or **isc_dsql_alloc_statement2()**; returns an error in status_vector if NULL
dialect	*unsigned short*	Indicates the version of the extended SQL descriptor area (XSQLDA) passed to the function; set this value to 1
xsqlda	XSQLDA *	Pointer to a previously allocated XSQLDA used for input

Description **isc_dsql_execute()** executes a DSQL statement previously prepared with **isc_dsql_prepare()**. **isc_dsql_execute()** can be used to execute two types of statements:

- Statements that may return more than one row of data.
- Statements that need to be executed more than once.

If a statement to execute has input parameters, then **isc_dsql_execute()** requires an input XSQLDA to describe those parameters. It does not provide for an output XSQLDA. A call to **isc_dsql_execute()** that executes a SELECT statement results in the creation of a *list* containing all the rows of data that are the result of execution of the statement. To access these rows, call **isc_dsql_fetch()** in a loop. Each call to **isc_dsql_fetch()** fetches the next row from the select-list.

If the statement to be executed requires input parameter values (that is, if it contains parameter markers), these values must be supplied in the input XSQLDA *xsqlda* before calling **isc_dsql_execute()**.

Note To execute a statement repeatedly when it both has input parameters and return values, such as EXECUTE PROCEDURE, use **isc_dsql_execute2()** which requires both an input and an output XSQLDA.

If you only need to execute a statement once, and it does not return any data, call **isc_dsql_execute_immediate()** instead of **isc_dsql_prepare()** and **isc_dsql_execute()**. To execute a statement with both input and output parameters a single time, use **isc_dsql_exec_immed2()**.

Note CREATE DATABASE and SET TRANSACTION cannot be executed with **isc_dsql_execute()** or **isc_dsql_execute2()**. To execute these statements, use **isc_dsql_execute_immediate()**.

Example The following program fragment illustrates calls to **isc_dsql_execute()** and **isc_dsql_fetch()**. It allocates input and output XSQLDAs, prepares a SELECT statement, executes it, and fetches and processes each row one-by-one.

```
#include <ibase.h>
ISC_STATUS status_vector[20], fetch_stat;
XSQLDA *isqlda, *osqlda;
XSQLVAR *ivar, *ovar;
char *str = "SELECT CITY, POPULATION FROM CITIES WHERE STATE = ?";
char *state = "CA";
/* Allocate an output XSQLDA osqlda. */
osqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(2);
osqlda->version = SQLDA_VERSION1;
osqlda->sqln = 2;

/* Prepare the statement, including filling in osqlda with information
about the select-list items to be returned by the statement. */
isc_dsql_prepare(
   status_vector,
   &tr_handle, /* Set in previous isc_start_transaction() call. */
   &stmt_handle,
      /* Allocated previously by isc_dsql_allocate_statement()
         or isc_dsql_alloc_statement2() call. */
```

```
    0,
    str,
    1,
    osqlda);
if (status_vector[0] == 1 && status_vector[1])
{
    /* Process error. */
    isc_print_status(status_vector);
    return(1);
}

/* Check to see whether or not the output XSQLDA had enough XSQLVARS
allocated. If not, correct it -- see isc_dsql_describe(). */

/* Allocate and fill in the input XSQLDA. This example assumes you know
how many input parameters there are (1), and all other information
necessary to supply a value. If this is not true, you will need to call
isc_dsql_describe_bind(). */
isqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(1));
isqlda->version = SQLDA_VERSION1;
isqlda->sqln = 1;
isqlda->sqld = 1;
ivar = isqlda->sqlvar[0];
ivar->sqltype = SQL_TEXT;
ivar->sqllen = sizeof(state);
ivar->sqldata = state;

/* Execute the statement. */
isc_dsql_execute(
    status_vector,
    &tr_handle, /* Set in previous isc_start_transaction() call. */
    &stmt_handle,
        /* Allocated previously by isc_dsql_allocate_statement()
            or isc_dsql_alloc_statement2() call. */
    1,
    isqlda);
if (status_vector[0] == 1 && status_vector[1])
{
    /* Process error. */
    isc_print_status(status_vector);
    return(1);
}
```

```
/* Set up an output XSQLVAR structure to allocate space for each item
to be returned. */
for (i=0, ovar = osqlda->sqlvar; i < osqlda->sqld; i++, ovar++)
{
   dtype = (ovar->sqltype & ~1) /* Drop NULL bit for now. */
   switch(dtype)
      {
      case SQL_TEXT:
         ovar->sqldata = (char *)malloc(sizeof(char) * ovar->sqllen);
         break;
      case SQL_LONG:
         ovar->sqldata = (char *)malloc(sizeof(long));
      /* Process remaining types. */
         . . .
      }
   if (ovar->sqltype & 1)
   {
      /* Assign a variable to hold NULL status. */
      ovar->sqlind = (short *)malloc(sizeof(short));
   }
} /* end of for loop */

/* Fetch and process the rows in the select list one by one. */
while ((fetch_stat = isc_dsql_fetch(
   status_vector,
   &stmt_handle,
   1,
   osqlda)) == 0)
{
   for (i=0; i < osqlda->sqld; i++)
   {
      /*  Call a function you've written to process each returned
      select-list item. */
      process_column(osqlda->sqlvar[i]);
   }
}
```

Return Value **isc_dsql_execute()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to *isc_bad_stmt_handle*, *isc_bad_trans_handle*, or another InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_dsql_describe_bind()**, **isc_dsql_exec_immed2()**, **isc_dsql_execute_immediate()**, **isc_dsql_execute2()**, **isc_dsql_fetch()**, **isc_dsql_prepare()**

For more information about creating and populating the XSQLDA, see **"Understanding the XSQLDA" on page 83**.

isc_dsql_execute2()

Executes a previously prepared DSQL statement.

Syntax

```
ISC_STATUS isc_dsql_execute2(
ISC_STATUS *status_vector,
isc_tr_handle *trans_handle,
isc_stmt_handle *stmt_handle,
unsigned short dialect,
XSQLDA *in_xsqlda,
XSQLDA *out_xsqlda);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
trans_handle	*isc_tr_handle* *	Pointer to a transaction handle whose value has been set by a previous **isc_start_transaction()** call; *trans_handle* returns an error if NULL
stmt_handle	*isc_stmt_handle* *	Pointer to a statement handle previously allocated with **isc_dsql_allocate_statement()** or **isc_dsql_alloc_statement2()**; the handle returns an error in status_vector if it is NULL
dialect	unsigned short	Indicates the version of the extended SQL descriptor area (XSQLDA) passed to the function; set this value to 1

Parameter	Type	Description
in_xsqlda	XSQLDA *	Pointer to an optional, previously allocated XSQLDA used for input; if input parameters are not supplied, set this value to NULL
out_xsqlda	XSQLDA *	Pointer to an optional, previously allocated XSQLDA used for results of statement execution; if not required, set this value to NULL

Description **isc_dsql_execute2()** executes a previously prepared DSQL statement that has input parameters and returns results, such as EXECUTE PROCEDURE and SELECT.

If the statement to execute requires input parameter values (that is, if it contains parameter markers), these values must be supplied in the input XSQLDA, *in_xsqlda* before calling **isc_dsql_execute2()**.

If the statement to execute returns values, they are placed in the specified output XSQLDA, *out_xsqlda*. If a NULL value is supplied for the output XSQLDA and the statement returns values, they are stored in an *result set*. To access the returned data, use **isc_dsql_fetch()** in a loop.

TIP If you just want to execute once a statement returning just one group of data, call **isc_dsql_exec_immed2()** instead of **isc_dsql_prepare()** and **isc_dsql_execute2()**.

To execute a statement that does not return any data a single time, call **isc_dsql_execute_immediate()** instead of **isc_dsql_prepare()** and **isc_dsql_execute2()**.

Note CREATE DATABASE and SET TRANSACTION cannot be executed with **isc_dsql_execute()** or **isc_dsql_execute2()**. To execute these statements, use **isc_dsql_execute_immediate()**.

Example The following program fragment illustrates a sequence of calls that allocates an input XSQLDA and loads values into it, allocates an output XSQLDA, prepares an EXECUTE PROCEDURE statement, allocates space in the output XSQLDA for each column returned for each row retrieved by the call, and executes the prepared statement, placing return values in the output XSQLDA.

```
#include <ibase.h>
ISC_STATUS status_vector[20];
XSQLDA *isqlda, *osqlda;
XSQLVAR *ivar, *ovar;
short null_flag;
char *str = "EXECUTE PROCEDURE P1";
char *state = "CA";
/* Allocate an output XSQLDA osqlda. This example assumes you know that
P1 will return one value. */
```

```
osqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(1);
osqlda->version = SQLDA_VERSION1;
osqlda->sqln = 1;

/* Prepare the statement, including filling in osqlda with information
about the item to be returned by the statement (procedure). */
isc_dsql_prepare(
   status_vector,
   &tr_handle, /* Set in previous isc_start_transaction() call. */
   &stmt_handle,
   /* Allocated previously by isc_dsql_allocate_statement()
         or isc_dsql_alloc_statement2() call. */
   0,
   str,
   1,
   osqlda);
if (status_vector[0] == 1 && status_vector[1])
{
   /* Process error. */
   isc_print_status(status_vector);
   return(1);
}
/* Set up the output XSQLVAR structure to allocate space for the return
value. Again, this example assumes you know that P1 returns just one
value. For an example of what to do if you’re not sure, see
isc_dsql_describe(). For an example of setting up an output XSQLVAR
structure to allocate space for multiple return items, see the
isc_dsql_execute() example program. */
ovar = osqlda->sqlvar[0];
dtype = (ovar->sqltype & ~1); /* Drop NULL bit for now. */
switch(dtype)
{
   case SQL_TEXT:
      ovar->sqldata = (char *)malloc(sizeof(char) * ovar->sqllen);
      break;
   case SQL_LONG:
      ovar->sqldata = (char *)malloc(sizeof(long));
   /* Process remaining types. */
   . . .
}
if (ovar->sqltype & 1)
{
    /* Assign a variable to hold NULL status. */
    ovar->sqlind = &null_flag;
}
/* Allocate and fill in the input XSQLDA. This example assumes you know
how many input parameters there are (1), and all other information
necessary to supply a value. If this is not true, you will need to call
isc_dsql_describe_bind(). */
isqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(1);
isqlda->version = SQLDA_VERSION1;
isqlda->sqln = 1;
isqlda->sqld = 1;
ivar = isqlda->sqlvar[0];
ivar->sqltype = SQL_TEXT;
ivar->sqllen = sizeof(state);
ivar->sqldata = state;

/* Execute the statement. */
isc_dsql_execute2(
    status_vector,
    &tr_handle, /* Set in previous isc_start_transaction() call. */
    &stmt_handle,
    /* Allocated previously by isc_dsql_allocate_statement()
            or isc_dsql_alloc_statement2() call. */
    1,
    isqlda,
    osqlda);
if (status_vector[0] == 1 && status_vector[1])
{
    /* Process error. */
    isc_print_status(status_vector);
    return(1);
}
/* Now process the value returned in osqlda->sqlvar[0]. */
. . .
```

Return Value **isc_dsql_execute2()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to *isc_bad_stmt_handle*, *isc_bad_trans_handle*, or another InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_dsql_exec_immed2()**, **isc_dsql_execute_immediate()**, **isc_dsql_execute()**, **isc_dsql_fetch()**, **isc_dsql_prepare()**

For more information about creating and populating the XSQLDA, see **"Understanding the XSQLDA" on page 83**.

isc_dsql_execute_immediate()

Prepares and executes just once a DSQL statement that does not return data.

Syntax

```
ISC_STATUS isc_dsql_execute_immediate(
ISC_STATUS *status_vector,
isc_db_handle *db_handle,
isc_tr_handle *trans_handle,
unsigned short length,
char *statement,
unsigned short dialect,
XSQLDA *xsqlda);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
db_handle	*isc_db_handle* *	Pointer to a database handle set by a previous call to **isc_attach_database()** *db_handle* returns an error in *status_vector* if it is NULL
trans_handle	*isc_tr_handle* *	Pointer to a transaction handle whose value has been set by a previous **isc_start_transaction()** call; *trans_handle* returns an error if NULL
length	unsigned short	Length of the DSQL statement in bytes; set to 0 in C programs to indicate a null-terminated string

Parameter	Type	Description
statement	char *	DSQL string to be executed
dialect	unsigned short	Indicates the version of the extended SQL descriptor area (XSQLDA) passed to the function; set this value to 1
xsqlda	XSQLDA *	Pointer to an optional, previously allocated XSQLDA used for input; if input parameters are not supplied, set this value to NULL

Description **isc_dsql_execute_immediate()** prepares the DSQL statement specified in *statement*, executes it once, and discards it. The statement must not be one that returns data (that is, it must not be a SELECT or EXECUTE PROCEDURE statement).

If *statement* requires input parameter values (that is, if it contains parameter markers), these values must be supplied in the input XSQLDA, *xsqlda*.

TIP If *statement* returns data, or if it needs to be executed more than once, use **isc_dsql_prepare()** and **isc_dsql_execute()** (or **isc_dsql_execute2()**) instead of **isc_dsql_execute_immediate()**.

Note You *must* call **isc_dsql_execute_immediate()** for CREATE DATABASE and SET TRANSACTION; you cannot prepare and execute such statements by calling **isc_dsql_prepare()** and **isc_dsql_execute()**.

Example The following program fragment calls **isc_dsql_execute_immediate()**:

```
#include <ibase.h>
ISC_STATUS status_vector[20];
char *insert_stmt =
   "INSERT INTO CUSTOMER(CUSTNAME, BAL, CUSTNO)
   VALUES("John Smith", 299.0, 5050)";

isc_dsql_execute_immediate(
   status_vector,
   &database_handle, /* Set in previous isc_attach_database() call. */
   &tr_handle, /* Set in previous isc_start_transaction() call. */
   0,
   insert_stmt,
   1,
   NULL);
if (status_vector[0] == 1 && status_vector[1])
{
   /* Process error. */
   isc_print_status(status_vector);
   return(1);
}
```

Return Value **isc_dsql_execute_immediate()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to *isc_bad_db_handle*, *isc_bad_trans_handle*, or another InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_dsql_exec_immed2()**, **isc_dsql_execute()**, **isc_dsql_prepare()**

For more information about creating and populating the XSQLDA, see **"Understanding the XSQLDA" on page 83**.

isc_dsql_exec_immed2()

Prepares and executes just once, a DSQL statement that returns no more than one row of data.

Syntax
```
ISC_STATUS isc_dsql_exec_immed2(
ISC_STATUS *status_vector,
isc_db_handle *db_handle,
isc_tr_handle *trans_handle,
unsigned short length,
char *statement,
unsigned short dialect,
XSQLDA *in_xsqlda,
XSQLDA *out_xsqlda);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
db_handle	*isc_db_handle* *	Pointer to a database handle set by a previous call to **isc_attach_database()** *db_handle* returns an error in *status_vector* if it is NULL
trans_handle	*isc_tr_handle* *	Pointer to a transaction handle whose value has been set by a previous **isc_start_transaction()** call; *trans_handle* returns an error if NULL
length	unsigned short	Length of the DSQL statement, in bytes; set to 0 in C programs to indicate a null-terminated string
statement	char *	DSQL string to be executed
dialect	unsigned short	Indicates the version of the extended SQL descriptor area (XSQLDA) passed to the function; set this value to 1
in_xsqlda	XSQLDA *	Pointer to an optional, previously allocated XSQLDA used for input; if input parameters are not supplied, set this value to NULL
out_xsqlda	XSQLDA *	Pointer to an optional, previously allocated XSQLDA used for results of statement execution. If not required, set this value to NULL.

Description **isc_dsql_exec_immed2()** prepares the DSQL statement specified in *statement*, executes it once, and discards it. *statement* can return a single set of values (i.e, it can be an EXECUTE PROCEDURE or singleton SELECT) in the output XSQLDA.

If *statement* requires input parameter values (that is, if it contains parameter markers), these values must be supplied in the input XSQLDA, *in_xsqlda*.

For statements that return multiple rows of data, use **isc_dsql_prepare()**, **isc_dsql_execute2()**, and **isc_dsql_fetch()**.

Example The following program fragment calls **isc_dsql_exec_immed2()**:

```
ISC_STATUS status_vector[20];
XSQLDA *in_xsqlda, *out_xsqlda;
char *execute_p1 = "EXECUTE PROCEDURE P1 ?";
/* Set up input and output XSQLDA structures here. */
. . .
isc_dsql_exec_immed2(
   status_vector,
   &database_handle, /* Set in previous isc_attach_database() call. */
   &tr_handle, /* Set in previous isc_start_transaction() call. */
   0,
   execute_p1,
   1,
   in_xsqlda,
   out_xsqlda);
if (status_vector[0] == 1 && status_vector[1])
{
   /* Process error. */
   isc_print_status(status_vector);
   return(1);
}
```

Return Value **isc_dsql_exec_immed2()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to *isc_bad_db_handle*, *isc_bad_trans_handle*, or another InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_dsql_execute2()**, **isc_dsql_prepare()**

For more information about creating and populating the XSQLDA, see **"Understanding the XSQLDA" on page 83**.

isc_dsql_fetch()

Retrieves data returned by a previously prepared and executed DSQL statement.

Syntax

```
ISC_STATUS isc_dsql_fetch(
ISC_STATUS *status_vector,
isc_stmt_handle *stmt_handle,
unsigned short dialect,
XSQLDA *xsqlda);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
stmt_handle	*isc_stmt_handle* *	Pointer to a statement handle previously allocated with **isc_dsql_allocate_statement()** or **isc_dsql_alloc_statement2()**; the handle returns an error in status_vector if it is NULL
dialect	unsigned short	Indicates the version of the extended SQL descriptor area (XSQLDA) passed to the function; set this value to 1
xsqlda	XSQLDA *	Pointer to an optional, previously allocated XSQLDA used for results of statement execution

Description **isc_dsql_fetch()** retrieves one row of data into *xsqlda* each time it is called. It is used in a loop to retrieve and process each row of data for statements that return multiple rows in a cursor.

A cursor is a one-way pointer into the ordered set of rows retrieved by a statement. A cursor is only needed to process positioned UPDATE and DELETE statements made against the rows retrieved by **isc_dsql_fetch()** for SELECT statements that specify an optional FOR UPDATE OF clause.

It is up to the application to provide the loop construct for fetching the data.

Before calling **isc_dsql_fetch()**, a statement must be prepared with **isc_dsql_prepare()**, and executed with **isc_dsql_execute()** (or **isc_dsql_execute2()** with a NULL output XSQLDA argument). Statement execution produces a *result set* containing the data returned. Each call to **isc_dsql_fetch()** retrieves the next available row of data from the result set into *xsqlda*.

Example The following program fragment illustrates a sequence of calls that allocates an output XSQLDA, prepares a statement for execution, allocates an XSQLVAR structure in the XSQLDA for each column of data to be retrieved, executes the statement, producing a select list of returned data, then fetches and processes each row in a loop:

```
#include <ibase.h>
#define LASTLEN 20
#define FIRSTLEN 15
#define EXTLEN 4
typedef struct vary {
   short vary_length;
   char vary_string[1];
} VARY;
ISC_STATUS status_vector[20], retcode;
long SQLCODE;
XSQLDA *osqlda;
XSQLVAR *ovar;
short flag0, flag1, flag2;
char *str =
      "SELECT last_name, first_name, phone_ext FROM phone_list
         WHERE location = "Monterey" ORDER BY last_name, first_name";
char last_name[LASTLEN + 2];
char first_name[FIRSTLEN + 2];
char phone_ext[EXTLEN + 2];
VARY *vary;
/* Allocate an output XSQLDA osqlda. */
osqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(3);
osqlda->version = SQLDA_VERSION1;
osqlda->sqln = 3;
/* Prepare the statement. */
isc_dsql_prepare(
   status_vector,
   &tr_handle, /* Set in previous isc_start_transaction() call. */
   &stmt_handle,
   /* Allocated previously by isc_dsql_allocate_statement()
         or isc_dsql_alloc_statement2() call. */
   0,
   str,
   1,
   osqlda);
if (status_vector[0] == 1 && status_vector[1])
{
    /* Process error. */
    isc_print_status(status_vector);
    return(1);
}
/* Set up an output XSQLVAR structure to allocate space for each item
to be returned. */
osqlda->sqlvar[0].sqldata = last_name;
osqlda->sqlvar[0].sqltype = SQL_VARYING + 1;
osqlda->sqlvar[0].sqlind = &flag0;
osqlda->sqlvar[1].sqldata = first_name;
osqlda->sqlvar[1].sqltype = SQL_VARYING + 1;
osqlda->sqlvar[1].sqlind = &flag1;
osqlda->sqlvar[2].sqldata = phone_ext;
osqlda->sqlvar[2].sqltype = SQL_VARYING + 1;
osqlda->sqlvar[2].sqlind = &flag2;
/* Execute the statement. */
isc_dsql_execute(
    status_vector,
    &tr_handle, /* Set in previous isc_start_transaction() call. */
    &stmt_handle,
        /* Allocated previously by isc_dsql_allocate_statement()
            or isc_dsql_alloc_statement2() call. */
    1,
    NULL);
if (status_vector[0] == 1 && status_vector[1])
{
    /* Process error. */
    isc_print_status(status_vector);
    return(1);
}

printf("\n%-20s %-15s %-10s\n\n", "LAST NAME", "FIRST NAME",
"EXTENSION");
/* Fetch and print the records in the select list one by one. */
while ((retcode = isc_dsql_fetch(
    status_vector,
    &stmt_handle,
    1,
    osqlda)) == 0)
{
    vary = (VARY *)last_name;
    printf("%-20.*s ", vary->vary_length, vary->vary_string);
    vary = (VARY *)first_name;
    printf("%-15.*s ", vary->vary_length, vary->vary_string);
    vary = (VARY *)phone_ext;
    printf("%-4.*s ", vary->vary_length, vary->vary_string);
}
if (retcode != 100L)
{
    SQLCODE = isc_sqlcode(status_vector);
    isc_print_sqlerror(SQLCODE, status_vector);
    return(1);
}
```

Return Value **isc_dsql_fetch()** returns the second element of the status vector. Zero indicates success. The value 100 indicates that no more rows remain to be retrieved. Any other nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to *isc_bad_stmt_handle*, or another InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_dsql_execute()**, **isc_dsql_execute2()**, **isc_dsql_prepare()**

isc_dsql_free_statement()

Frees a statement handle and all resources allocated for it, or closes a cursor associated with the statement referenced by a statement handle.

Syntax

```
ISC_STATUS isc_dsql_free_statement(
ISC_STATUS *status_vector,
isc_stmt_handle *stmt_handle,
unsigned short option);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
stmt_handle	*isc_stmt_handle* *	Pointer to a statement handle previously allocated with **isc_dsql_allocate_statement()** or **isc_dsql_alloc_statement2()**; the handle returns an error in status_vector if it is NULL
option	unsigned short	Either *DSQL_close* or *DSQL_drop*

Description **isc_dsql_free_statement()** either frees a statement handle and all resources allocated for it (*option* = *DSQL_drop*), or closes a cursor associated with the statement (*option* = *DSQL_close*).

Note **isc_dsql_free_statement()** does nothing if it is called with an *option* value other than *DSQL_drop* or *DSQL_close*.

▸ *DSQL_close*

Call **isc_dsql_free_statement()** with the *DSQL_close* option to close a cursor after it is no longer needed, that is, after fetching and processing all the rows resulting from the execution of a query. A cursor need only be closed in this manner if it was previously opened and associated with *stmt_handle* by **isc_dsql_set_cursor_name()**.

DSQL_close closes a cursor, but the statement it was associated with remains available for further execution.

If you have used a cursor to perform updates or deletes on all the rows returned from the execution of a query, and you want to perform other update or delete operations on rows resulting from execution of the same statement again (possibly with different input parameters), follow these steps:

1. Close the cursor with **isc_dsql_free_statement()**.
2. Re-open it with **isc_dsql_set_cursor_name()**.
3. If desired, change the input parameters to be passed to the statement.
4. Re-execute the statement to retrieve a new select list.
5. Retrieve rows in a loop with **isc_dsql_fetch()** and process them again with **isc_dsql_execute_immediate()**.

DSQL_drop

Statement handles allocated with **isc_dsql_allocate_statement()** must be released when no longer needed by calling **isc_dsql_free_statement()** with the *DSQL_drop* option. This option frees all resources associated with the statement handle, and closes any open cursors associated with the statement handle.

Example The following program fragment shows examples of the two types of **isc_dsql_free_statement()** calls. It assumes that *stmt_handle1* and *stmt_handle2* are statement handles, each of which was previously allocated with either **isc_dsql_allocate_statement()** or **isc_dsql_alloc_statement2()**. A cursor is also assumed to have been associated with the statement referenced by *stmt_handle1*.

```
#include <ibase.h>
ISC_STATUS status_vector[20];
. . .
/* Free the cursor associated with stmt_handle1. */
isc_dsql_free_statement(
   status_vector,
   &stmt_handle1,
   DSQL_close);
if (status_vector[0] == 1 && status_vector[1])
{
   isc_print_status(status_vector);
   return(1);
}
/* Free stmt_handle2. */
isc_dsql_free_statement(
   status_vector,
   &stmt_handle2,
   DSQL_drop);
if (status_vector[0] == 1 && status_vector[1])
{
   isc_print_status(status_vector);
   return(1);
}
```

Return Value **isc_dsql_free_statement()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to *isc_bad_stmt_handle*, or another InterBase error code.To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_dsql_allocate_statement()**, **isc_dsql_alloc_statement2()**, **isc_dsql_set_cursor_name()**

isc_dsql_prepare()

Prepares a DSQL statement for repeated execution.

Syntax

```
ISC_STATUS isc_dsql_prepare(
ISC_STATUS *status_vector,
isc_tr_handle *trans_handle,
isc_stmt_handle *stmt_handle,
unsigned short length,
char *statement,
unsigned short dialect,
XSQLDA *xsqlda);
```

Parameter	**Type**	**Description**
status_vector	ISC_STATUS *	Pointer to the error status vector
trans_handle	*isc_tr_handle* *	Pointer to a transaction handle whose value has been set by a previous **isc_start_transaction()** call; *trans_handle* returns an error if NULL
stmt_handle	*isc_stmt_handle* *	Pointer to a statement handle previously allocated with **isc_dsql_allocate_statement()** or **isc_dsql_alloc_statement2()**; the handle returns an error in status_vector if it is NULL
length	unsigned short	Length of the DSQL statement, in bytes; set to 0 in C programs to indicate a null-terminated string

Parameter	Type	Description
statement	char *	DSQL string to be executed
dialect	unsigned short	Indicates the version of the extended SQL descriptor area (XSQLDA) passed to the function; set this value to 1
xsqlda	XSQLDA *	Pointer to an optional, previously allocated XSQLDA used for results of statement execution

Description **isc_dsql_prepare()** readies the DSQL statement specified in *statement* for repeated execution by checking it for syntax errors and parsing it into a format that can be efficiently executed. All SELECT statements must be prepared with **isc_dsql_prepare()**.

After a statement is prepared, it is available for execution as many times as necessary during the current session. Preparing a statement for repeated execution is more efficient than using **isc_dsql_execute_immediate()** or **isc_dsql_exec_immed2()** over and over again to prepare and execute a statement.

If a statement to be prepared does not return data, set the output XSQLDA to NULL. Otherwise, the output XSQLDA must be allocated prior to calling **isc_dsql_prepare()**. Allocate the XSQLDA using the macro, XSQLDA_LENGTH, defined in *ibase.h*, as follows:

```
xsqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(n));
```

XSQLDA_LENGTH calculates the number of bytes required when *n* result columns will be returned by the statement, and allocates the appropriate amount of storage.

After allocating the XSQLDA *xsqlda*, set *xsqlda->version* to SQLDA_VERSION1, and set *xsqlda_sqln* to indicate the number of XSQLVAR structures allocated.

When **isc_dsql_prepare()** is called, it fills in the other fields of the XSQLDA and all the XSQLVARs with information such as the datatype, length, and name of the corresponding select-list items in the statement. It fills in *xsqlda->sqld* with the actual number of select-list items returned. If *xsqlda->sqld* is greater than *xsqlda->sqln*, then enough room is not allocated, and the XSQLDA must be resized by following these steps:

1. Record the current value of the *xsqlda->sqld*.
2. Free the storage previously allocated for *xsqlda*.
3. Reallocate storage for *xsqlda*, this time specifying the correct number (from step 1) in the argument to XSQLDA_LENGTH.
4. Reset *xsqlda->sqld* and *xsqlda->version*.
5. Execute **isc_dsql_describe()** to fill in the *xsqlda* fields.

Note If the prepared statement requires input parameter values, then an input XSQLDA will need to be allocated and filled in with appropriate values prior to calling **isc_dsql_execute()** or **isc_dsql_execute2()**. You can either allocate and directly fill in all the fields of the input XSQLDA, or you can allocate it, call **isc_dsql_describe_bind()** to get information regarding the number and types of parameters required, then fill in appropriate values.

Example The following program fragment illustrates the allocation of the output XSQLDA, and a call to **isc_dsql_prepare()**:

```
#include <ibase.h>
ISC_STATUS status_vector[20];
XSQLDA *osqlda;
char *query =
       "SELECT CITY, STATE, POPULATION FROM CITIES \
          WHERE STATE = "NY" ORDER BY CITY DESCENDING";

osqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(3);
osqlda->version = SQLDA_VERSION1;
osqlda->sqln = 3;

isc_dsql_prepare(
   status_vector,
   &tr_handle, /* Set in previous isc_start_transaction() call. */
   &stmt_handle,
      /* Allocated previously by isc_dsql_allocate_statement()
         or isc_dsql_alloc_statement2() call. */
   0,
   query,
   1,
   osqlda);
if (status_vector[0] == 1 && status_vector[1])
{
   isc_print_status(status_vector);
   return(1);
}
```

More complete examples showing the subsequent execution and fetching of result data are provided in the example programs for **isc_dsql_execute()**, **isc_dsql_execute2()**, and **isc_dsql_fetch()**.

Return Value **isc_dsql_prepare()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to *isc_bad_stmt_handle*, *isc_bad_trans_handle*, or another InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_dsql_describe()**, **isc_dsql_describe_bind()**, **isc_dsql_execute()**, **isc_dsql_execute2()**, **isc_dsql_fetch()**

For more information about creating and populating the XSQLDA, see **"Understanding the XSQLDA" on page 83** of **Chapter 6, "Working with Dynamic SQL."**

isc_dsql_set_cursor_name()

Defines a cursor name and associates it with a DSQL statement.

Syntax

```
ISC_STATUS isc_dsql_set_cursor_name(
ISC_STATUS *status_vector,
isc_stmt_handle *stmt_handle,
char *cursor_name,
unsigned short type);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
stmt_handle	*isc_stmt_handle* *	Pointer to a statement handle previously allocated with **isc_dsql_allocate_statement()** or **isc_dsql_alloc_statement2()**; the handle returns an error in status_vector if it is NULL
cursor_name	char *	String name of a cursor
type	unsigned short	Reserved for future use; set to NULL

Description **isc_dsql_set_cursor_name()** defines a cursor name and associates it with a DSQL statement handle for a statement that returns multiple rows of data (for example, SELECT), effectively opening the cursor for access.

A cursor is a one-way pointer into the ordered set of rows retrieved by a statement. A cursor is only needed to process positioned UPDATE and DELETE statements made against the rows retrieved by **isc_dsql_fetch()** for SELECT statements that specify an optional FOR UPDATE OF clause.

Note In UPDATE or DELETE statements, the cursor name cannot be supplied as a parameter marker (?).

When a cursor is no longer needed, close it with the *DSQL_close* option of **isc_dsql_free_statement()**.

Example The following pseudo-code illustrates the calling sequence necessary to execute an UPDATE or DELETE with the WHERE CURRENT OF clause using a cursor name established and opened with **isc_dsql_set_cursor_name()**:

```
#include <ibase.h>
ISC_STATUS status_vector[20], fetch_stat;
isc_stmt_handle st_handle = NULL;
char *cursor = "S";

/* Allocate the statement handle st_handle. */
isc_dsql_allocate_statement(
   status_vector,
   &db, /* Database handle set by isc_attach_database() call. /*
   &st_handle);
if (status_vector[0] == 1 && status_vector[1])
{
   isc_print_status(status_vector);
   return(1);
}
/* Set up an output XSQLDA osqlda here. */
/* Call isc_dsql_prepare() to prepare the SELECT statement. */
/* Set up an input XSQLDA, if needed, for the SELECT statement. */
/* Call isc_dsql_execute() to execute the SELECT statement. */
/* Set up an input XSQLDA (if needed) for the UPDATE or DELETE
statement. */
/* Declare the cursor name, and associate it with st_handle. */
isc_dsql_set_cursor_name(
   status_vector,
   &st_handle,
   cursor, 0);
if (status_vector[0] == 1 && status_vector[1])
{
   isc_print_status(status_vector);
```

```
    return(1);
}
/* Fetch rows one by one, with the cursor pointing to each row as it
is fetched, and execute an UPDATE or DELETE statement to update or
delete the row pointed to by the cursor. */
while ((fetch_stat = isc_dsql_fetch(
    status_vector, &st_handle, 1, osqlda)) == 0)
{
    . . .
    /* Update or delete the current row by executing an "UPDATE ...
        WHERE CURRENT OF S" or "DELETE ... WHERE CURRENT OF S"
        statement, where "S" is the name of the cursor declared in
        isc_dsql_set_cursor_name(). */
}
```

Return Value **isc_dsql_set_cursor_name()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to *isc_bad_stmt_handle*, or another InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_dsql_fetch()**, **isc_dsql_free_statement()**

isc_dsql_sql_info()

Returns requested information about a prepared DSQL statement.

Syntax

```
ISC_STATUS isc_dsql_sql_info(
ISC_STATUS *status_vector,
isc_stmt_handle *stmt_handle,
unsigned short item_length,
char *items,
unsigned short buffer_length,
char *buffer);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
stmt_handle	*isc_stmt_handle* *	Pointer to a statement handle previously allocated with **isc_dsql_allocate_statement()** or **isc_dsql_alloc_statement2()**; the handle returns an error in status_vector if it is NULL
item_length	unsigned short	Number of bytes in the string of information items in items
items	char *	String of requested information items
buffer_length	unsigned short	Number of bytes in the result buffer, *buffer*
buffer	char *	User-provided buffer for holding returned data; must be large enough to hold the information requested

Description **isc_dsql_sql_info()** returns requested information about a statement prepared with a call to **isc_dsql_prepare()**. The main application need for this function is to determine the statement type of an unknown prepared statement, for example, a statement entered by the user at run time.

Requested information can include the:

- Statement type.
- Number of input parameters required by the statement.
- Number of output values returned by the statement.
- Detailed information regarding each input parameter or output value, including its datatype, scale, and length.

Example The following illustrates a call to **isc_dsql_sql_info()** to determine the statement type of the statement whose handle is referenced by *stmt*:

```
int statement_type;
int length;
char type_item[] = {isc_info_sql_stmt_type};
char res_buffer[8];
isc_dsql_sql_info(
   status_vector,
   &stmt,
      /* Allocated previously by isc_dsql_allocate_statement() or
         isc_dsql_alloc_statement2() call. */
   sizeof(type_item),
   type_item,
   sizeof(res_buffer),
   res_buffer);

if (res_buffer[0] == isc_info_sql_stmt_type)
{
   length = isc_vax_integer(buffer[1], 2);
   statement_type = isc_vax_integer(buffer[3], length);
}
```

Return Value **isc_dsql_sql_info()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_dsql_describe_bind()**, **isc_dsql_describe()**, **isc_vax_integer()**

For more information about determining unknown statement types at run time, see **"Determining an unknown statement type at runtime" on page 111** of **Chapter 6, "Working with Dynamic SQL."**

isc_encode_date()

Translates a date from the C *tm* format to InterBase ISC_QUAD format prior to inserting or updating a DATE value in a table.

Syntax

```
void isc_encode_date(
void *tm_date,
ISC_QUAD *ib_date);
```

Parameter	**Type**	**Description**
tm_date	void *	Pointer to a C *tm* structure
ib_date	ISC_QUAD *	Pointer to an eight-byte ISC_QUAD structure containing a date in InterBase format

Description **isc_encode_date()** translates a date in a C time structure into an ISC_QUAD format internal to InterBase. This call is used prior to writing DATE data to a table to guarantee that the date is in a format recognized by InterBase.

Use the **isc_dsql** family of API calls to insert or update DATE data from the ISC_QUAD structure in a table.

Example The following code fragment illustrates declaring time structures and calling **isc_encode_date()** to translate a C time format into an InterBase date format prior to inserting or updating a table:

```
#include <time.h>
#include <ibase.h>
. . .
struct tm hire_time;
ISC_QUAD hire_date;
. . .
/* Store date info into the tm struct here. */
. . .
isc_encode_date(&hire_time, &hire_date);
/* Now use a DSQL INSERT or UPDATE statement to move the date into a
DATE column. */
```

Return Value None.

See Also **isc_decode_date()**

isc_event_block()

Allocates two event parameter buffers (EPBs) for subsequent use with other API event calls.

Syntax

```
long isc_event_block(
char **event_buffer,
char **result_buffer,
unsigned short id_count,
. . .);
```

Parameter	Type	Description
event_buffer	char **	Address of a character pointer; this function allocates and initializes an event parameter buffer and stores its address into the character pointer
result_buffer	char **	Address of a character pointer; this function allocates an event parameter buffer, and stores its address into the character pointer
id_count	unsigned short	Number of event identifier strings that follow
…	char *	Up to 15 null-terminated and comma-separated strings that each name an event

Description **isc_event_block()** must be called before any other event functions. It:

- Allocates two event parameter buffers of the same size, and stores their addresses into the character pointers addressed by *event_buffer* and *result_buffer*.
- Stores into the buffer referenced by *event_buffer* the names and event counts for each of the specified events. The names are the ones that appear as the final arguments to **isc_event_block()**. The event counts are initialized to zero and are used to specify how many times each event has been posted prior to each wait for events to occur.
- Returns the length, in bytes, of the buffers.

The buffers, and their lengths, are used in subsequent calls to the functions **isc_wait_for_event()**, **isc_que_events()**, and **isc_event_counts()**. *event_buffer* is used to indicate the events of interest, and to hold the counts in effect before a wait for one of the events. After an event is posted, *result_buffer* is filled in exactly as *event_buffer*, except that the event counts are updated. **isc_event_counts()** is then called to determine which events were posted between the time the counts were set in *event_buffer*, and the time the counts are set in *result_buffer*.

Example The following program fragment illustrates a call to **isc_event_block()**:

```
#define number_of_stocks 3;

char *event_buffer, *result_buffer;
long length;

length = isc_event_block(
   &event_buffer,
   &result_buffer,
   number_of_stocks,
   "DEC", "HP", "SUN");
```

Return Value **isc_event_block()** returns a number that is the size, in bytes, of each event parameter buffer it allocates.

See Also **isc_event_counts()**, **isc_que_events()**, **isc_wait_for_event()**

isc_event_counts()

Compares event parameter buffers (EPBs) to determine which events have been posted, and prepares the event parameter buffers for the next call to **isc_que_events()** or **isc_wait_for_event()**.

Syntax

```
void isc_event_counts(
ISC_STATUS *status_vector,
short buffer_length,
char *event_buffer,
char *result_buffer);
```

Parameter	**Type**	**Description**
status_vector	long *	Pointer to the status vector, which is used to store the differences in event counts for each corresponding event in *event_buffer* and *result_buffer*

Parameter	Type	Description
buffer_length	short	Length of the event parameter buffers, returned by the **isc_event_block()** call that allocated them
event_buffer	char *	Pointer to the event parameter buffer that specifies the event counts prior to the previous call to **isc_wait_for_event()** or **isc_que_events()**
result_buffer	char *	Pointer to the event parameter buffer filled in as a result of posting an event

Description **isc_event_counts()** compares the event counts in the event parameter buffers, *event_buffer* and *result_buffer*, and sets up to the first 15 elements of *status_array* to contain the differences. It then modifies *event_buffer* to contain the same event counts as *result_buffer* in preparation for the next call to either **isc_wait_for_event()** or **isc_que_events()**.

The counts in *event_buffer* specify how many times each event had been posted since the previous call to **isc_event_wait()** or **isc_que_events()**. The counts in *result_buffer* equal the values in *event_buffer* plus the number of additional times an event is posted after the current call to **isc_event_wait()** or **isc_que_events()**. If an event is posted after a call to either of these functions, its count is greater in *result_buffer* than in *event_buffer*. Other event counts may also be greater because an event may have been posted between calls to either of these functions. The values in *sta*tus*_array* are the differences in values between *event_buffer* and *result_buffer*. This mechanism of comparing all the counts ensures that no event postings are missed.

Example The following program fragment illustrates the set-up and waiting on any of the events named “DEC”, “HP”, or “SUN”, then calling **isc_event_counts()** to determine which events have been posted:

```
#include <ibase.h>
#define number_of_stocks 3;

char *event_buffer, *result_buffer;
ISC_STATUS status_vector[20];
char *event_names[] = {"DEC", "HP", "SUN"};
long length;
int i;

length = isc_event_block(
   &event_buffer,
   &result_buffer,
    number_of_stocks,
    "DEC", "HP", "SUN");

isc_wait_for_event(
    status_vector,
    &database_handle, /* Set by previous isc_attach_database(). */
    length,   /* Returned from isc_event_block(). */
    event_buffer,
    result_buffer);
if (status_vector[0] == 1 && status_vector[1])
{
    isc_print_status(status_vector); /* Display error message. */
    return(1);
}

isc_event_counts(
    status_vector,
    (short) length,
    event_buffer,
    result_buffer);

for (i=0; i<number_of_stocks; i++)
    if (status_vector[i])
    {
        /* The event has been posted. Do whatever is appropriate, for
example,
         initiating a buy or sell order. */
        ;
    }
```

Return Value None.

See Also **isc_que_events()**, **isc_wait_for_event()**

isc_expand_dpb()

Dynamically builds or expands a database parameter buffer (DPB) to include database parameters.

Syntax

```
void isc_expand_dpb(
char **dpb,
unsigned short *dpb_size,
. . .);
```

Parameter	Type	Description
dpb	char **	Pointer to an existing DPB
dpb_size	unsigned short *	Pointer to the current size, in bytes, of the DPB
…	char *	Pointers to items to insert into the expanded DPB

Description **isc_expand_dpb()** builds or expands a DPB dynamically. Its main use is to simplify the building of the DPB prior to a call to **isc_attach_database()**, or to allow an end user to supply a user name and password combination at run time. In many cases, the DPB must be constructed programmatically, but **isc_expand_dpb()** enables an application to pass user names, password, message file, and character set parameters to the function, which then adds them to an existing DPB.

A pointer to a previously allocated and initialized DPB must be passed to **isc_expand_dpb()** along with a pointer to a variable containing the current size of the DPB when this function is called. If the space allocated for the DPB is not large enough for the parameters passed to **isc_expand_dpb()**, then the function reallocates a larger DPB, preserving its current contents, and adds the new parameters.

To ensure proper memory management, applications that call **isc_expand_dpb()** should always allocate DPBs large enough to hold all anticipated parameters.

Example The following code calls **isc_expand_dpb()** to create a DPB, then attaches to a database using the newly created DPB. *user_name* and *user_password* are assumed to be variables whose values have been filled in, for example, after asking the user to specify the name and password to be used.

```
#include <ibase.h>
char *dpb;
ISC_STATUS status_vector[20];
isc_db_handle handle = NULL;
short dpb_length;
```

```
/* Build the database parameter buffer. */

dpb = (char *) malloc(50);
dpb_length = 0;

isc_expand_dpb(&dpb, &dpb_length, isc_dpb_user_name, user_name,
isc_dpb_password, user_password, NULL);

isc_attach_database(
   status_vector,
   0,
   "employee.db",
   &handle,
   dpb_length,
   dpb_buffer);
if (status_vector[0] == 1 && status_vector[1])
{
   /* An error occurred. */
   isc_print_status(status_vector);
   return(1);
}
```

Return Value None.

See Also **isc_attach_database()**

isc_get_segment()

Reads a segment from an open Blob.

Syntax

```
ISC_STATUS isc_get_segment(
ISC_STATUS *status_vector,
isc_blob_handle *blob_handle,
unsigned short *actual_seg_length,
unsigned short seg_buffer_length,
char *seg_buffer);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
blob_handle	*isc_blob_handle* *	Pointer to the handle of the Blob you want to read.
actual_seg_length	unsigned short *	Pointer to the actual segment length that InterBase reads into the buffer; useful if the segment length is shorter than the buffer length
seg_buffer_length	unsigned short	Length of the segment buffer
seg_buffer	char *	Pointer to the segment buffer

Description

isc_get_segment() reads a Blob segment from a previously opened Blob. You can set the *seg_buffer_length* parameter to a size that is efficient for a particular type of Blob data. For example, if you are reading Blob data from a text file, you might set the segment buffer length to 80, to take advantage of the 72 to 80 character line lengths that are common in text files. By periodically checking the value of the actual segment length in your loop, you can determine an end-of-line or end-of-file condition.

Before reading any part of a Blob, you must open the Blob with a call to **isc_open_blob2()**. **isc_get_segment()** behaves differently depending on which call precedes it. If the most recent call is to **isc_open_blob2()**, then a call to **isc_get_segment()** reads the first segment in the Blob. If the most recent call is to **isc_get_segment()**, then it reads the next segment.

If Blob filters are specified when a Blob is opened, then each segment retrieved by **isc_get_segment()** is filtered on read.

Note Blob filters are not supported on NetWare.

You can read bitmaps and other binary files directly, without filtering, if you don't need to change from one format to another, say from .TIF to .JPEG. You can also store compressed bitmaps directly in a database in formats such as .JPG (JPEG), .BMP (Windows native bitmaps), or .GIF (CompuServe Graphic Interchange Format). No filtering is required.

You can store bitmaps in a database in row-major or column-major order.

If the buffer is not large enough to hold the entire current segment, the function returns *isc_segment*, and the next call to **isc_get_segment()** gets the next chunk of the oversized segment rather than getting the next segment.

When **isc_get_segment()** reads the last segment of the Blob, the function returns the code *isc_segstr_eof*.

For more information about reading data from a Blob, see **Chapter 7, "Working with Blob Data."**

Example The following call gets a segment from one Blob and writes it to another:

```
get_status = isc_get_segment(status, &from_blob, &seg_len, 80,
buffer);
if (status[0] == 1 && status[1])
{
    isc_print_status(status);
    return(1);
}
if (get_status != isc_segstr_eof)
    write_status = isc_put_segment(status, &to_blob, seg_len, buffer);
if (status[0] == 1 && status[1])
{
    isc_print_status(status);
    return(1);
}
```

Return Value **isc_get_segment()** returns the second element of the status vector. Zero indicates success. *isc_segment* indicates the buffer is not large enough to hold the entire current segment; the next call to **isc_get_segment()** gets the next chunk of the oversized segment rather than getting the next segment. *isc_segstr_eof* indicates that the last segment of the Blob has been read. Any other nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_create_blob2()**, **isc_open_blob2()**, **isc_put_segment()**

isc_interprete()

Extracts the text for an InterBase error message from the error status vector to a user-defined buffer.

Syntax

```
ISC_STATUS isc_interprete(
char *buffer,
ISC_STATUS **status_vector);
```

Parameter	Type	Description
buffer	char *	Application buffer for storing an InterBase error message
status_vector	ISC_STATUS **	Pointer to a pointer to the error status vector

Description Given both the location of a storage buffer allocated in a program, and the address of the status vector, **isc_interprete()** builds an error message string from the information in the status vector, puts the formatted string in the buffer where the program can manipulate it, and advances the status vector pointer to the start of the next cluster of error message information. For example, you might declare an error string buffer, call **isc_interprete()** to retrieve the first error message and insert the message into the buffer, write the buffer to a log file, then peek at the next cluster to see if it contains more error information.

isc_interprete() retrieves and formats a single message each time it is called. When an error occurs, however, the status vector usually contains more than one error message. To retrieve all relevant error messages, you must make repeated calls to **isc_interprete()** until no more messages are returned.

Note Do not pass the address of the status vector directly, because each time **isc_interprete()** is called, it modifies the pointer to the status vector to point to the start of the next available message.

To display all error messages on the screen instead of to a buffer, use **isc_print_status()**.

Example The following code declares a message buffer, a status vector, and a pointer to the vector, then illustrates how repeated calls are made to **isc_interprete()** to store all messages in the buffer:

```
#include <ibase.h>
char msg[512];
```

```
ISC_STATUS status_vector[20];
long *pvector; /* Pointer to pointer to status vector. */
FILE *efile; /* Code fragment assumes this points to an open file. */
. . .
pvector = status_vector; /* (Re)set to start of status vector. */
isc_interprete(msg, &pvector); /* Retrieve first message. */
fprintf(efile, "%s\n", msg); /* Write buffer to log file. */
msg[0] = '-'; /* Append leading hyphen to secondary messages. */
while(isc_interprete(msg + 1,&pvector)) /* More messages? */
{
   fprintf(efile, "%s\n", msg); /* If so, write them, too. */
}
fclose(efile);
. . .
```

Return Value If successful, **isc_interprete()** returns the length of the error message string it stores in *buffer*. It also advances the status vector pointer to the start of the next cluster of error message information.

If there are no more messages in the status vector, or if **isc_interprete()** cannot interpret the next message, it returns 0.

See Also **isc_print_sqlerror()**, **isc_print_status()**, **isc_sqlcode()**, **isc_sql_interprete()**

isc_modify_user( )

Modifies a user record from the password database, *isc4.gdb*.

Syntax

```
ISC_STATUS isc_modify_user(
ISC_STATUS *status
USER_SEC_DATA *user_sec_data);
```

Parameter	Type	Description
status vector	ISC_STATUS *	Pointer to the error status vector
user_sec_data	USER_SEC_DATA *	Pointer to a struct that is defined in *ibase.h*

Description The three security functions, **isc_add_user()**, **isc_delete_user()**, and **isc_modify_user()** mirror functionality that is available in the **gsec** command-line utility. **isc_modify_user()** modifies a record from *isc4.gdb*, InterBase's password database.

At a minimum, you must provide the user name. Any additional user information that you supply, such as first name, last name, or password, overwrites the information that is already in *isc4.gdb*.

If the server is not local, you must provide both a server name and a protocol. Valid choices for the protocol field are *sec_protocol_tcpip*, *sec_protocol_netbeui*, *sec_protocol_spx*, and *sec_protocol_local*.

InterBase reads the settings for the ISC_USER and ISC_PASSWORD environment variables if you do not provide a DBA user name and password.

The definition for the USER_SEC_DATA struct in *ibase.h* is as follows:

```
typedef struct {
   short  sec_flags;      /* which fields are specified */
   int    uid;            /* the user's id */
   int    gid;            /* the user's group id */
   int    protocol;       /* protocol to use for connection */
   char   *server;        /* server to administer */
   char   *user_name;     /* the user's name */
   char   *password;      /* the user's password */
   char   *group_name;    /* the group name */
   char   *first_name;    /* the user's first name */
   char   *middle_name;   /* the user's middle name */
   char   *last_name;     /* the user's last name */
   char   *dba_user_name; /* the dba user name */
   char   *dba_password;  /* the dba password */
} USER_SEC_DATA;
```

When you pass this struct to one of the three security functions, you can tell it which fields you have specified by doing a bitwise OR of the following values, which are defined in *ibase.h*:

```
sec_uid_spec                0x01
sec_gid_spec                0x02
sec_server_spec             0x04
sec_password_spec           0x08
sec_group_name_spec         0x10
sec_first_name_spec         0x20
sec_middle_name_spec        0x40
sec_last_name_spec          0x80
sec_dba_user_name_spec      0x100
sec_dba_password_spec        0x200
```

No bit values are available for user name and password, since they are required.

The following error messages exist for this function:

Code	Value	Description
isc_usrname_too_long	335544747	The user name passed in is greater than 31 bytes
isc_password_too_long	335544748	The password passed in is longer than 8 bytes
isc_usrname_required	335544749	The operation requires a user name
isc_password_required	335544750	The operation requires a password
isc_bad_protocol	335544751	The protocol specified is invalid
isc_dup_usrname_found	335544752	The user name being added already exists in the security database.

TABLE 12.19 Error messages for user security functions

Code	Value	Description
isc_usrname_not_found	335544753	The user name was not found in the security database
isc_error_adding_sec_record	335544754	An unknown error occurred while adding a user
isc_error_deleting_sec_record	335544755	An unknown error occurred while deleting a user
isc_error_modifying_sec_record	335544756	An unknown error occurred while modifying a user
isc_error_updating_sec_db	335544757	An unknown error occurred while updating the security database

TABLE 12.19 Error messages for user security functions

Example The following example modifies *isc4.gdb* to change the password for the user Socks, using the bitwise OR technique for passing values from the USER_SEC_DATA struct.

```
{
   ISC_STATUS status[20];
   USER_SEC_DATA sec;

   sec.server       = "kennel";
   sec.dba_user_name= "sysdba";
   sec.dba_password = "masterkey";
   sec.protocol     = sec_protocol_tcpip;
   sec.user_name    = "socks";
   sec.password     = "feed_me!"; /* Note: do not hardcode passwords
*/
   sec.sec_flags    = sec_server_spec
                    | sec_password_spec
                    | sec_dba_user_name_spec
                    | sec_dba_password_spec;

   isc_add_user(status, &sec);
   /* check status for errors */
   if (status[0] == 1 && status[1])
   {
      switch (status[1]) {
      case isc_usrname_too_long:
         printf("Security database cannot accept long user names\n");
         break;
      ...
```

```
        }
    }
}
```

Return Value **isc_modify_user()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. See the "Description" section for this function for a list of error codes. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_add_user( )**, **isc_delete_user( )**

isc_open_blob2()

Opens an existing Blob for retrieval and optional filtering.

Syntax

```
ISC_STATUS isc_open_blob2(
ISC_STATUS *status_vector,
isc_db_handle *db_handle,
isc_tr_handle *trans_handle,
isc_blob_handle *blob_handle,
ISC_QUAD *blob_id,
short bpb_length,
char *bpb_address);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
db_handle	*isc_db_handle* *	Pointer to a database handle set by a previous call to **isc_attach_database()** *db_handle* returns an error in *status_vector* if it is NULL
trans_handle	*isc_tr_handle* *	Pointer to a transaction handle whose value has been set by a previous **isc_start_transaction()** call; *trans_handle* returns an error if NULL
blob_handle	*isc_blob_handle* *	Pointer to the Blob handle, which must be NULL when you make this call

Parameter	Type	Description
blob_id	ISC_QUAD *	Pointer to the 64-bit system-defined Blob ID, which is stored in a field in the table and points to the first segment of the Blob or to a page of pointers to Blob fragments
bpb_length	short	Length of the Blob parameter buffer (BPB)
bpb_address	char *	Pointer to the BPB

Description **isc_open_blob2()** opens an existing Blob for retrieval and optional filtering from one Blob subtype to another.

Note Using Blob filters is not supported on NetWare.

Input and output Blob filter types are passed to **isc_open_blob2()** as subtype information in a previously populated BPB, pointed to by *bpb_address*. If Blob filters are not needed or cannot be used, a BPB is not needed; pass 0 for *bpb_length* and NULL for *bpb_address*.

The *blob_id* identifies which particular Blob is to be opened. This *blob_id* is set by a sequence of DSQL function calls.

On success, **isc_open_blob2()** assigns a unique ID to *blob_handle*. Subsequent API calls use this handle to identify the Blob against which they operate.

After a blob is opened, its data can be read by a sequence of calls to **isc_get_segment()**.

When finished accessing the Blob, close it with **isc_close_blob()**.

For more information about opening a Blob for retrieval and optional filtering, see **Chapter 7, "Working with Blob Data."**

Example The following fragment is excerpted from the example file, *api9.c*. The example program displays job descriptions that are passed through a filter.

```
while ((fetch_stat = isc_dsql_fetch(status, &stmt, 1, sqlda)) == 0)
{
   printf("\nJOB CODE: %5s GRADE: %d", job_code, job_grade);
   printf(" COUNTRY: %-20s\n\n", job_country);
   /* Open the blob with the fetched blob_id. */
   isc_open_blob2(status, &DB, &trans, &blob_handle, &blob_id, 9,
bpb);
   if (status[0] == 1 && status[1])
   {
      isc_print_status(status);
      return(1);
```

```
    }
}
```

Return Value **isc_open_blob2()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_close_blob()**

isc_prepare_transaction()

Executes the first phase of a two-phase commit against multiple databases.

Syntax

```
ISC_STATUS isc_prepare_transaction(
ISC_STATUS *status_vector,
isc_tr_handle *trans_handle);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
trans_handle	*isc_tr_handle* *	Pointer to a transaction handle whose value has been set by a previous **isc_start_transaction()** call; *trans_handle* returns an error if NULL

Description **isc_prepare_transaction()** initiates the first phase of a two-phase commit under program direction. It alerts InterBase, which polls all database participants and waits for replies. The **isc_prepare_transaction()** function puts the transaction in limbo.

Because a call to this function indicates that you intend to control all phases of the commit, you must complete the second phase of the commit by explicitly calling the **isc_commit_transaction()** function.

If a call to **isc_prepare_transaction()** fails, the application should roll back the transaction with a call to the **isc_rollback_transaction()** function.

Note If you want InterBase to automatically perform the two-phase commit, call **isc_commit_transaction()** without calling **isc_prepare_transaction()**.

Example The following example executes the first phase of a two-phase commit and includes a rollback in case of failure:

```
isc_prepare_transaction(status_vector, &trans);
if (status_vector[0] == 1 && status_vector[1])
   rb_status = isc_rollback_transaction(status_vector, &trans)
else
{
   isc_commit_transaction(status_vector, &trans);
   if (!(status_vector[0] == 1 && status_vector[1]))
      fprintf("Commit successful.");
}
```

Return Value **isc_prepare_transaction()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_commit_transaction()**, **isc_prepare_transaction2()**, **isc_rollback_transaction()**

isc_prepare_transaction2()

Performs the first phase of a two-phase commit for multi-database transactions.

Syntax

```
ISC_STATUS isc_prepare_transaction2(
ISC_STATUS *status_vector,
isc_tr_handle *trans_handle,
unsigned short msg_length,
char *message);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
trans_handle	*isc_tr_handle* *	Pointer to a transaction handle whose value has been set by a previous **isc_start_transaction()** call; *trans_handle* returns an error if NULL
msg_length	unsigned short	Length of message in bytes
message	char *	Transaction description buffer

Description **isc_prepare_transaction2()** performs the first phase of a two-phase commit, just as **isc_prepare_transaction()** does, but **isc_prepare_transaction2()** expects you to provide two additional arguments:

- An information message to write to the RDB$TRANSACTION_DESCRIPTION column in the RDB$TRANSACTIONS system table that describes the transaction to commit, so that recovery is possible in the event a system crash occurs during the completion of the commit.
- The length, in bytes, of the information message.

By electing to use **isc_prepare_transaction2()**, you are, in effect, disabling the automatic recovery functions inherent in the two-phase commit. It is your responsibility to deal with recovery issues that might occur during failure of the two-phase commit. Normally, InterBase automatically writes to the RDB$TRANSACTION_DESCRIPTION column in the RDB$TRANSACTIONS system table information that makes it possible to reconnect following a system crash during the commit. You can manually write a message string into RDB$TRANSACTIONS, by using the *message* parameter in this function.

At the risk of preventing recovery in the event of a system crash, you might choose to avoid writing a message to RDB$TRANSACTION altogether if you determine that there is too much overhead associated with this extra action every time your application commits.

Example The following example executes the first phase of a two-phase commit and includes a rollback in case of failure:

```
isc_prepare_transaction2(status_vector, &trans, msg_len, msg);
if (status_vector[0] == 1 && status_vector[1])
   rb_status = isc_rollback_transaction(status_vector, &trans);
```

Return Value **isc_prepare_transaction2()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_commit_transaction()**, **isc_prepare_transaction()**, **isc_rollback_transaction()**

isc_print_sqlerror()

Displays an SQLCODE value, a corresponding SQL error message, and any additional InterBase error messages in the error status vector.

Syntax

```
void isc_print_sqlerror(
short SQLCODE,
ISC_STATUS *status_vector);
```

Parameter	Type	Description
SQLCODE	short	Variable containing an SQLCODE value
status_vector	ISC_STATUS *	Pointer to the error status vector

Description During the processing of DSQL API calls, SQL errors can occur. SQL errors are generally reported in a variable called SQLCODE. DSQL calls return error information to a user-defined error status vector like any other API call, but **isc_print_sqlerror()** can be used to interpret the primary error condition as an SQL error message for direct display on the screen. To use **isc_print_sqlerror()**, an application must declare both an SQLCODE variable for holding the SQL error number, and an error status vector for holding InterBase error information. **isc_print_sqlerror()** displays the SQLCODE value, a related SQL error message, and any additional InterBase error messages in the status array.

Note Some windowing systems do not permit direct screen writes. Do not use **isc_print_sqlerror()** when developing applications for these environments. Instead, use **isc_sql_interprete()** and **isc_interprete()** to capture messages to a buffer for display.

Example The following code calls **isc_print_sqlerror()** when an error occurs:

```
#include <ibase.h>
long SQLCODE;
ISC_STATUS status_vector[20];
. . .
if (status_vector[0] == 1 && status_vector[1])
{
   SQLCODE = isc_sqlcode(status_vector);
   isc_print_sqlerror(SQLCODE, status_vector);
}
```

Return Value None.

See Also **isc_interprete()**, **isc_print_status()**, **isc_sql_interprete()**, **isc_sqlcode()**

isc_print_status()

Builds and displays error messages based on the contents of the InterBase error status vector.

Syntax

```
ISC_STATUS isc_print_status(ISC_STATUS *status_vector);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector

Description **isc_print_status()** builds all error messages based on the contents of the error status vector, and displays them on the screen. *status_vector* must be declared in the program as an array of twenty elements.

Example The following code displays error messages when an error occurs during processing:

```
#include <ibase.h>
ISC_STATUS status_vector[20];
. . .
if (status_vector[0] == 1 && status_vector[1])
{
   isc_print_status(status_vector);
   return(1);
}
```

Return Value **isc_print_status()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_interprete()**, **isc_print_sqlerror()**, **isc_sqlcode()**, **isc_sql_interprete()**

isc_put_segment()

Writes a Blob segment.

Syntax

```
ISC_STATUS isc_put_segment(
ISC_STATUS *status_vector,
isc_blob_handle *blob_handle,
unsigned short seg_buffer_length,
char *seg_buffer);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
blob_handle	*isc_blob_handle* *	Pointer to the handle of the Blob to which you want to write; use **isc_create_blob2()** to set a value for this handle
seg_buffer_length	unsigned short	Length of the Blob segment buffer
seg_buffer_address	char *	Pointer to the Blob segment buffer that contains data for writing

Description **isc_put_segment()** writes a Blob segment in *seg_buffer_address* to a Blob previously created and opened with **isc_create_blob2()**.

If a Blob filter was specified when the Blob was created, then each segment is filtered before storing the result into the Blob.

The behavior of **isc_put_segment()** depends on what call preceded it. If the most recent call was to **isc_create_blob()** or **isc_create_blob2()**, then a call to **isc_put_segment()** writes the first segment of the Blob. If the most recent call was to **isc_put_segment()**, then it writes the next segment.

You can write bitmaps and other binary files directly, without filtering, unless you intend to change from one format to another, say from .GEM to .BMP. You can also store compressed bitmaps directly in a database, in formats such as .JPG (JPEG), .BMP (Windows native bitmaps), or .GIF (CompuServe Graphic Interchange Format).

You can store bitmaps in your database in row-major or column-major order.

You cannot update a Blob directly. If you want to modify Blob data, you must do one of the following:

- Create a new Blob.
- Read the old Blob data into a buffer where you can edit or modify it.
- Write the modified data to the new Blob.
- Prepare and execute an UPDATE statement that will modify the Blob column to contain the Blob ID of the new Blob, replacing the old Blob's Blob ID.

For more information about creating and writing Blob data, see **Chapter 7, "Working with Blob Data."**

Note To read a segment that you wrote with a call to **isc_put_segment()**, you must close the Blob with **isc_close_blob()**, and then open it with **isc_open_blob2()**.

Example The following example reads a segment of one Blob and writes it to another Blob:

```
get_status = isc_get_segment(status, &from_blob, &seg_len, 80,
buffer);
if (status[0] == 1 && status[1])
{
   isc_print_status(status);
   return(1);
}
if (get_status != isc_segstr_eof)
   write_status = isc_put_segment(status, &to_blob, seg_len, buffer);
if (status[0] == 1 && status[1])
{
   isc_print_status(status);
   return(1);
}
```

Return Value **isc_put_segment()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_close_blob()**, **isc_get_segment()**, **isc_open_blob2()**

isc_que_events()

Requests asynchronous notification of one of a specified group of events.

Syntax

```
ISC_STATUS isc_que_events(
ISC_STATUS *status_vector,
isc_db_handle *db_handle,
ISC_LONG *event_id,
short length,
char *event_buffer,
isc_callback event_function,
void *event_function_arg);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
db_handle	*isc_db_handle* *	Pointer to a database handle set by a previous call to **isc_attach_database()**; the handle identifies the database against which the events are expected to be posted *db_handle* returns an error in *status_vector* if it is NULL
event_id	ISC_LONG *	Pointer to an event identifier to set
length	short	Length of the event parameter buffers, returned by the **isc_event_block()** call which allocated them

Parameter	Type	Description
event_buffer	char *	Pointer to the event parameter buffer that specifies the current counts of the events to be waited on; this buffer should have been initially allocated and filled in by a call to **isc_event_block()**
event_function	*isc_callback*	Pointer to the address of the function to receive event notification
event_function_arg	void *	First argument to be passed to *event_function*, usually a pointer to the event parameter buffer you want filled in with updated event counts

Description **isc_que_events()** is called to request asynchronous notification of any of the events listed in *event_buffer*. Upon completion of the call, but before events are posted, control is returned to the calling application, which can continue other processing. When a requested event is posted, InterBase calls the function specified in *event_function* to process event occurrence.

After *event_function* is called, you must call **isc_que_events()** again if you want to start another asynchronous wait on the specified events.

Note **isc_que_events()** cannot be called from within *event_function*.

If you want to cancel your **isc_que_events()** request for asynchronous event notification, call **isc_cancel_events()**.

Note To request *synchronous* notification, call **isc_wait_for_event()**.

Example The following program fragment illustrates calling **isc_que_events()** to wait asynchronously for event occurrences. Within a loop, it performs other processing, and checks the event flag (presumably set by the specified event function) to determine when an event has been posted. If one has, the program resets the event flag, calls **isc_event_counts()** to determine which events have been posted since the last call to **isc_que_events()**, and calls **isc_que_events()** to initiate another asynchronous wait.

```
#include <ibase.h>
#define number_of_stocks 3;
#define MAX_LOOP 10

char *event_names[] = {"DEC", "HP", "SUN"};
char *event_buffer, *result_buffer;
ISC_STATUS count_array[number_of_stocks];
short length;
ISC_LONG event_id;
```

```
int i, counter;
int event_flag = 0;

length = (short)isc_event_block(
   &event_buffer,
   &result_buffer,
   number_of_stocks,
   "DEC", "HP", "SUN");

isc_que_events(
   status_vector,
   &database_handle, /* Set in previous isc_attach_database(). */
   &event_id,
   length, /* Returned from isc_event_block(). */
   event_buffer,
   (isc_callback)event_function,
   result_buffer);
if (status_vector[0] == 1 && status_vector[1])
{
   isc_print_status(status_vector); /* Display error message. */
   return(1);
};

counter = 0;
while (counter < MAX_LOOP)
{
   counter++;

   if (!event_flag)
   {
      /* Do whatever other processing you want. */
      ;
   }
   else
   {  event_flag = 0;
      isc_event_counts(
         count_array,
         length,
         event_buffer,
         result_buffer);
      if (status_vector[0] == 1 && status_vector[1])
      {
            isc_print_status(status_vector); /* Display error message.
*/
            return(1);
        }

        for (i=0; i<number_of_stocks; i++)
            if (count_array[i])
            {
                /* The event has been posted. Do whatever is appropriate,
                    for example, initiating a buy or sell order.
                    Note: event_names[i] tells the name of the event
                    corresponding to count_array[i]. */
                ;
            }

        isc_que_events(
            status_vector,
            &database_handle,
            &event_id,
            length,
            event_buffer,
            (isc_callback)event_function,
            result_buffer);
        if (status_vector[0] == 1 && status_vector[1])
        {
            isc_print_status(status_vector); /* Display error message.
*/
            return(1);
        }
    }  /* End of else. */
}  /* End of while. */

/* Let InterBase know you no longer want to wait asynchronously. */
isc_cancel_events(
    status_vector,
    &database_handle,
    &event_id);
if (status_vector[0] == 1 && status_vector[1])
{
    isc_print_status(status_vector); /* Display error message. */
    return(1);
}
```

Return Value **isc_que_events()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_cancel_events()**, **isc_event_block()**, **isc_event_counts()**, **isc_wait_for_event()**

For more information about writing an asynchronous event trap (AST) function, see **Chapter 11, "Working with Events."**

isc_rollback_transaction()

Undoes changes made by a transaction, and restores the database to its state prior to the start of the specified transaction.

Syntax

```
ISC_STATUS isc_rollback_transaction(
ISC_STATUS *status_vector,
isc_tr_handle *trans_handle);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
trans_handle	*isc_tr_handle* *	Pointer to a transaction handle whose value has been set by a previous **isc_start_transaction()** call; *trans_handle* returns an error if NULL

Description **isc_rollback_transaction()** rolls back a specified transaction, closes record streams, frees system resources, and sets the transaction handle to zero. It is typically used to undo all database changes made by a transaction when an error occurs.

A call to this function can fail only if:

- You pass a NULL or invalid transaction handle.
- The transaction dealt with more than one database and a communications link fails during the rollback operation. If that happens, subtransactions on the remote node will end up in limbo. You must use the database maintenance utility to manually roll back those transactions.

Example The following call rolls back a transaction:

```
isc_rollback_transaction(status_vector, &trans);
```

Return Value **isc_rollback_transaction()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_commit_transaction()**

isc_sqlcode()

Translates an InterBase error code in the error status vector to an SQL error code number.

Syntax

```
ISC_LONG isc_sqlcode (ISC_STATUS *status_vector);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector

Description **isc_sqlcode()** searches *status_vector* for a reported SQL error, and if it finds it, translates the InterBase error code number into an appropriate SQL error code. Typically, this call is used to populate a program variable (usually called SQLCODE for portability among SQL implementations) with an SQL error number for use in an SQL error-handling routine.

Example The following code illustrates how **isc_sqlcode()** might be called in a DSQL application:

```
#include <ibase.h>
long SQLCODE;
ISC_STATUS status_vector[20];
. . .
if (status_vector[0] == 1 && status_vector[1])
{
   SQLCODE = isc_sqlcode(status_vector);
   isc_print_sqlerror(SQLCODE, status_vector);
}
```

Return Value If successful, **isc_sqlcode()** returns the first valid SQL error code decoded from the InterBase status vector.

If no valid SQL error code is found, **isc_sqlcode()** returns –999.

See Also **isc_interprete()**, **isc_print_sqlerror()**, **isc_print_status()**, **isc_sql_interprete()**

isc_sql_interprete()

Builds an SQL error message string and stores it in a user-defined buffer.

Syntax

```
void isc_sql_interprete(
short SQLCODE,
char *buffer,
short buffer_length);
```

Parameter	Type	Description
SQLCODE	short	Variable containing an SQLCODE value
buffer	char *	Application buffer into which to store an SQL error message
buffer_length	short	Length, in bytes, of *buffer*

Description Given an SQLCODE value less than zero, **isc_sql_interprete()** builds a corresponding SQL error message string, and stores it in a user-defined buffer. The size of the buffer, in bytes, must also be passed to this function.

To display an SQL error message corresponding to an SQLCODE value, use **isc_print_sqlerror()** instead of this call.

Example The following code fragment illustrates a call to **isc_sql_interprete()**:

```
#include <ibase.h>
long SQLCODE;
char err_buf[256];
. . .
if (status_vector[0] == 1 && status_vector[1])
{
   SQLCODE = isc_sqlcode(status_vector);
   isc_sql_interprete(SQLCODE, err_buf, sizeof(err_buff));
}
```

Return Value None.

See Also **isc_interprete()**, **isc_print_sqlerror()**, **isc_print_status()**, **isc_sqlcode()**

isc_start_multiple()

Begins a new transaction against multiple databases.

Syntax

```
ISC_STATUS isc_start_multiple(
ISC_STATUS *status_vector,
isc_tr_handle *trans_handle,
short db_handle_count,
void *teb_vector_address);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
trans_handle	*isc_tr_handle* *	Pointer to a transaction handle whose value has been set by a previous **isc_start_transaction()** call; *trans_handle* returns an error if NULL
db_handle_count	short	Number of database handles passed in this call via transaction existence buffers (TEBs)
teb_vector_address	void *	Pointer to the TEB

Description Call **isc_start_multiple()** if you:

- Are using a language that does not support a variable number of arguments in a function call.
- Do not know how many databases you want to attach to when coding the start transaction function.

isc_start_multiple() passes information about each target database to InterBase. That information is stored in an array of transaction existence blocks (TEBs) pointed to by the *teb_vector* parameter.

teb_vector is a pointer to a byte array that consists of consecutive TEBs, one TEB for each database to connect to. Each TEB consists of three items: a pointer to the database handle for a database against which the transaction should run; the length, in bytes, of the transaction parameter buffer (TPB) for the database, and a pointer to the TPB. The items in a TEB correspond to the items passed directly as parameters in calls to **isc_start_transaction()**. C programmers should use **isc_start_transaction()** instead of **isc_start_multiple()** whenever possible because it does not require setting up TEBs.

For more information about establishing TEBs and calling **isc_start_multiple()**, see **"Calling isc_start_multiple( )" on page 69** of **Chapter 5, "Working with Transactions."**

Example The following program starts a multiple-database transaction:

```
#include <ibase.h>

typedef struct { /* Define the ISC_TEB structure. */
   int *dbb_ptr;
   longtpb_len;
   char*tpb_ptr;
} ISC_TEB;

ISC_TEB teb_vec[2]; /* Declare the TEB vector. */

ISC_STATUS isc_status[20]; /* Status vector. */
long *db0, *db1, /* Database handle. */
long *trans; /* Transaction handle. */

static char
   isc_tpb_0[] = { /* Declare the first transaction parameter
buffer. */
      isc_tpb_version3, /* InterBase version. */
      isc_tpb_write,/* Read-write access. */
      isc_tpb_consistency, /* Serializable. */
      isc_tpb_wait, /* Wait on lock. */
      isc_tpb_lock_write, 3, /* Reserving IDS for update. */
      'I','D','S',
      isc_tpb_protected},/* Don't allow other transactions to
                  write to the table. */

   isc_tpb_1[] = {  /* Declare the second transaction.*/
                  /* Parameter buffer. */
      isc_tpb_version3, /* InterBase version. */
      isc_tpb_write,/* Read-write access. */
      isc_tpb_consistency, /* Serializable. */
      isc_tpb_wait, /* Wait on lock. */
      isc_tpb_lock_write, 3, /* Reserving table OZS for update. */
      'O','Z','S',
      isc_tpb_protected};/* Don't allow other transactions to
                  write to the table. */

main()
{
db0 = db1 = 0;
trans = 0;

/* If you can’t attach to test_0 database, attach to test_1. */

isc_attach_database(isc_status, 0, "test_0.gdb", &db0, 0,0);
if (isc_status[0] == 1 && isc_status[1])
   isc_attach_database(isc_status, 0, "test_1.gdb", &db1, 0,0);

if (db0 && db1)
   {           /* Assign database handles, tpb length, and
               tbp handle to the teb vectors. */
   teb_vec[0].dbb_ptr = &db0;
   teb_vec[0].tpb_len = sizeof (isc_tpb_0);
   teb_vec[0].tpb_ptr = isc_tpb_0;

   teb_vec[1].dbb_ptr = &db1;
   teb_vec[1].tpb_len = sizeof (isc_tpb_1);
   teb_vec[1].tpb_ptr = isc_tpb_1;

   if (isc_start_multiple(isc_status, &trans, 2, teb_vec))
   isc_print_status(isc_status);
   }

if (trans)
   isc_commit_transaction(isc_status, &trans);

if (db0 && !trans)
   isc_detach_database(isc_status, &db0);

if (db1 && !(trans && db0))
   isc_detach_database(isc_status, &db1);

if (isc_status[0] == 1 && isc_status[1])
   isc_print_status(isc_status);
}
```

Return Value **isc_start_multiple()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_commit_transaction()**, **isc_prepare_transaction()**, **isc_prepare_transaction2()**, **isc_rollback_transaction()**, **isc_start_transaction()**

For more information about transaction handles, see **"Creating transaction handles" on page 58** of **Chapter 5, "Working with Transactions."** For more information about creating and populating a TPB, see **"Creating a transaction parameter buffer" on page 60** of **Chapter 5, "Working with Transactions."** For more information on TEBs, see **"Calling isc_start_multiple( )" on page 69** of **Chapter 5, "Working with Transactions."**

isc_start_transaction()

Starts a new transaction against one or more databases.

Syntax

```
ISC_STATUS isc_start_transaction(
ISC_STATUS *status_vector,
isc_tr_handle *trans_handle,
short db_handle_count,
isc_db_handle *db_handle,
unsigned short tpb_length,
char *tpb_address,
[isc_db_handle *db_handle,
unsigned short tpb_length,
char *tpb_address ...]);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
trans_handle	*isc_tr_handle* *	Pointer to a transaction handle whose value has been set by a previous **isc_start_transaction()** call; *trans_handle* returns an error if NULL
db_handle_count	short	Number of database handles passed in this call

Parameter	Type	Description
db_handle	*isc_db_handle* *	Pointer to a database handle set by a previous call to **isc_attach_database()**; the handle identifies the database against which the events are expected to be posted *db_handle* returns an error in *status_vector* if it is NULL
tpb_length	unsigned short	Length of the transaction parameter buffer (TPB)
tpb_address	char *	Pointer to the TPB

Description **isc_start_transaction()** starts a new transaction against one or more databases specified as database handles.

Note If you have a variable number of databases to update, or are using a language that does not support a variable number of arguments in a function call, use **isc_start_multiple()** instead of **isc_start_transaction()**.

A single transaction can access multiple databases. This function passes information about each database it accesses and the conditions of access for that database in a *transaction parameter buffer* (TPB). The TPB is a variably-sized vector of bytes declared and populated by the program. It contains information describing intended transaction behavior such as its access and lock modes.

isc_start_transaction() can start a transaction against up to 16 databases. You must pass a database handle and a TPB for *each* referenced database. If you want to use defaults for the transaction, set *tpb_length* to zero. In this case, *tpb_vector* is a NULL pointer.

Example The following program includes a call to the start transaction function:

```
#include <ibase.h>

long
   isc_status[20], /* Status vector. */
   *db, /* Database handle. */
   *trans, /* Transaction handle. */

static char
   isc_tpb_0[] = {
      isc_tpb_version3, /* InterBase version. */
      isc_tpb_write,/* Read-write access. */
      isc_tpb_consistency, /* Consistency-mode transaction. */
      isc_tpb_wait, /* Wait on lock. */
      isc_tpb_lock_write, 3, /* Reserving IDS table for update. */
```

```
        "I","D","S",
        isc_tpb_protected};/* Don’t allow other transactions to
                    write against this table. */
main()
{
db = trans = 0;
isc_attach_database(isc_status, 0, "test.gdb", &db, 0,0);

if (db)
{
    isc_start_transaction(
        isc_status, &trans, 1, &db,
        sizeof(isc_tpb_0), isc_tpb_0);
    if (isc_status[0] == 1 && isc_status[1])
        isc_print_status(isc_status);
}
if (trans)
    isc_commit_transaction(isc_status, &trans);

if (db && !trans)
    isc_detach_database(isc_status, &db);

if (status_vector[0] == 1 && status_vector[1])
    isc_print_status(isc_status);
}
```

Return Value **isc_start_transaction()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, “Handling Error Conditions.”**

See Also **isc_commit_transaction()**, **isc_prepare_transaction()**, **isc_prepare_transaction2()**, **isc_rollback_transaction()**, **isc_start_multiple()**

For more information about transaction handles, see **“Creating transaction handles” on page 58** of **Chapter 5, “Working with Transactions.”** For more information about creating and populating a TPB, see **“Creating a transaction parameter buffer” on page 60** of **Chapter 5, “Working with Transactions.”**

isc_transaction_info()

Returns information about the specified named transaction.

Syntax

```
ISC_STATUS isc_transaction_info(
ISC_STATUS *status_vector,
isc_tr_handle *trans_handle,
short item_list_buffer_length,
char *item_list_buffer,
short result_buffer_length,
char *result_buffer);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
trans_handle	*isc_tr_handle* *	Pointer to a transaction handle whose value has been set by a previous **isc_start_transaction()** call; *trans_handle* returns an error if NULL
item_list_buffer_length	short	Number of bytes in the item-list buffer
item_list_buffer	char *	Pointer to the item-list buffer
result_buffer_length	short	Number of bytes in the result buffer
result_buffer	char *	Pointer to the result buffer

Description **isc_transaction_info()** returns information necessary for keeping track of transaction IDs. This call is used internally by **isc_prepare_transaction()**. You should not need to use it in your own applications.

You can explicitly retrieve information about the transaction ID by including the following constant in the item-list buffer, where the transaction items about which you want information are listed:

Item	Purpose	Size of next value	Value
isc_info_tra_id	Determine the transaction ID	2 bytes	transaction ID

TABLE 12.20 Transaction information request item

isc_transaction_info() uses two buffers defined in the calling program: the *item-list buffer*, which lists transaction items about which you want information, and a *result buffer*, where the information requested is reported.

To define the *item-list buffer*, include the parameters *item_list_buffer_length* and *item_list_buffer_address*. The item-list buffer is a regular byte vector with no structure.

To define the result buffer, include the parameters *result_buffer_length* and *result_buffer_address*. These parameters specify the length and address of a buffer where the InterBase engine will place the return values from the function call.

The values returned to the result buffer are unaligned clusters of generic binary numbers. Furthermore, all numbers are represented in a generic format, with the least significant byte first, and the most significant byte last. Signed numbers have the sign in the last byte. Convert the numbers to a datatype native to your system before interpreting them.

In your call, include the item specifying the transaction ID, *isc_info_tra_id*. InterBase returns the transaction ID in the result buffer. In addition to the information InterBase returns in response to a request, InterBase can also return one or more of the following status messages to the result buffer. Each status message is one unsigned byte in length:

Item	Description
isc_info_end	End of the messages
isc_info_truncated	Result buffer is too small to hold any more requested information
isc_info_error	Requested information is unavailable; check the status vector for an error code and message

TABLE 12.21 Status message return items

The function return value indicates only that InterBase accepted the request for information. It does not mean that it understood the request or that it supplied all of the requested information. Your application must interpret the contents of the result buffer for details about the transaction.

Example The following code fragment gets information about a transaction:

```
static char         /* Declare item-list buffer. */
tra_items[] =
   {isc_info_tra_id};
                    /* Declare result buffer. */
CHAR tra_info[32];
```

```
isc_transaction_info(status_vector,
      &tr_handle,
      sizeof (tra_items), /* Length of item-list buffer. */
      &tra_items,   /* Address of item-list buffer. */
      sizeof (tra_info), /* Length of result buffer. */
      &tra_info);   /* Address of result buffer. */
if (status_vector[0] == 1 && status_vector[1])
{
   isc_print_status(status_vector);
   return(1);
}
```

Return Value **isc_transaction_info()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_start_transaction()**

isc_vax_integer()

Reverses the byte order of an integer.

Syntax

```
ISC_LONG isc_vax_integer(
char *buffer,
short length);
```

Parameter	Type	Description
buffer	char *	Pointer to the integer to convert
length	short	Length, in bytes, of the integer to convert Valid lengths are 1, 2, and 4 bytes

Description **isc_vax_integer()** reverses the byte order of an integer, specified in *buffer*, and returns the newly ordered value.

A typical use for this function is to convert integer values passed into a database parameter buffer to a format where the least significant byte must be first and the most significant byte last. In InterBase, integer values must be represented in input parameter buffers (for example, the DPB) and are returned in result buffers in a generic format where the least significant byte is first, and the most significant byte last. **isc_vax_integer()** is used to convert integers to and from this format.

Example The following code fragment converts a 2-byte value, stored in a character buffer that is the result buffer returned by a function such as **isc_database_info()**:

```
#include <ibase.h>
char *p;
. . .
for(p = res_buffer; *p != isc_info_end;)
{
   /* Read item type of next cluster in the result buffer. */
   item = *p++;
   /* Read length of next value in result buffer, and convert. */
   len = isc_vax_integer(p, 2);
   p += len;
   /* Now process the actual value, len bytes in size. */
   . . .
}
```

Return Value **isc_vax_integer()** always returns a byte-reversed long integer value.

See Also **isc_attach_database()**, **isc_database_info()**

isc_version()

Returns database implementation and version information.

Syntax

```
int isc_version(
isc_db_handle *db_handle,
isc_callback function_name,
void *user_arg);
```

Parameter	Type	Description
db_handle	*isc_db_handle* *	Pointer to a database handle set by a previous call to **isc_attach_database()** *db_handle* returns an error in *status_vector* if it is NULL
function_name	*isc_callback*	Pointer to a function to call with the relevant information; passing a NULL pointer in C programs calls **printf()**
user_arg	void *	An application-specified parameter to pass as the first of two arguments to *function_name*

Description **isc_version()** determines the database implementation and on-disk structure (ODS) version numbers for the database specified by *db_handle*. It passes this information in two separate calls to the callback function pointed to by *function_name*.

function_name should point to an application function that takes two arguments: a void pointer, *user_arg*, and a char pointer. Applications can pass any kind of parameter desired in *user_arg*.

isc_version() makes two calls to *function_name*. First it determines the database implementation number, builds a string containing the information, and calls *function_name* with *user_arg*, and a pointer to the string containing the implementation number in the following format:

```
<implementation>(<class>), version "<version>"
```

where:

- *<implementation>* is a text string, such as “InterBase/NT”.
- *<class>* is a text string specifying the implementation class, such as “access method”.
- *<version>* is a version identification string, such as “4.0”.

The callback function specified by *function_name* is free to do with this information what it pleases.

After the callback function returns control to **isc_version()**, **isc_version()** builds a new string containing the ODS major and minor version numbers, then calls *function_name* a second time with *user_arg*, and a pointer to the string containing the ODS version number in the following format:

```
on disk structure version <ods_major_num>.<ods_minor_num>
```

where:

- *<ods_major_num>* is the major ODS number. Databases with different major version numbers have different physical layouts on disk and are incompatible with one another. A database engine can only access databases with a particular ODS major number.
- *<ods_minor_num>* is the minor ODS number. Differences in the minor ODS number, but not the major one indicate a non-structural change that still permits access by any database engine that recognizes the major version number.

TIP If a NULL pointer is passed for *function_name*, **isc_version()** sets *function_name* to point to the C **printf()** function.

Examples The following code fragment calls **isc_version()** with a NULL callback function:

```
#include <ibase.h>
. . .
int ret;
. . .
ret = isc_version(&db1, NULL, "\t%s\n");
```

Return Value If successful, **isc_version()** returns 0. Otherwise, it returns a nonzero value.

See Also **isc_database_info()**

isc_wait_for_event()

Waits synchronously until one of a specified group of events is posted.

Note The **isc_wait_for_event()** function was called **gds_$event_wait()** in InterBase 3.3. It is therefore the only function that can't be translated from 3.3 nomenclature to all later versions by replacing **gds_$** with **isc_**.

Syntax

```
ISC_STATUS isc_wait_for_event(
ISC_STATUS *status_vector,
isc_db_handle *db_handle,
short length,
char *event_buffer,
char *result_buffer);
```

Parameter	Type	Description
status_vector	ISC_STATUS *	Pointer to the error status vector
db_handle	*isc_db_handle* *	Pointer to a database handle set by a previous call to **isc_attach_database()**; the handle identifies the database against which the events are expected to be posted *db_handle* returns an error in *status_vector* if it is NULL
length	short	Length of the event parameter buffers, returned by the **isc_event_block()** call which allocated them
event_buffer	char *	Pointer to the event parameter buffer that specifies the current counts of the events to be waited on; this buffer should have been initially allocated and filled in by a call to **isc_event_block()**
result_buffer	char *	Pointer to the event parameter buffer to be filled in with updated event counts as a result of this function call; this buffer should have been initially allocated by a call to **isc_event_block()**

Description **isc_wait_for_event()** is used to wait synchronously until one of a specified group of events is posted. Control is not returned to the calling application until one of the specified events occurs.

Events to wait on are specified in *event_buffer*, which should have been initially allocated and filled in by a previous call to **isc_event_block()**.

When one of these events is posted, **isc_wait_for_event()** fills in *result_buffer* with data that exactly corresponds to that in the initial buffer, except that the event counts will be the updated ones. Control then returns from **isc_wait_for_event()** to the calling application. The application should then call **isc_event_counts()** to determine which event was posted.

Note To request *asynchronous* notification of event postings, use **isc_que_events()** instead of **isc_wait_for_event()**. You *must* use asynchronous notifications in Microsoft Windows applications, or wherever a process must not stop processing.

Example The following program fragment illustrates a call to **isc_wait_for_event()** to wait for a posting of any of the events named "DEC", "HP", or "SUN".

```
#include <ibase.h>
#define number_of_stocks 3;

char *event_buffer, *result_buffer;
short length;

length = (short)isc_event_block(
   &event_buffer,
   &result_buffer,
   number_of_stocks,
   "DEC", "HP", "SUN");

isc_wait_for_event(
   status_vector,
   &database_handle,
   length, /* Returned from isc_event_block(). */
   event_buffer,
   result_buffer);
if (status_vector[0] == 1 && status_vector[1])
{
   isc_print_status(status_vector); /* Display error message. */
   return(1);
}

/* Call isc_event_counts() to compare event counts in the buffers and
thus determine which event(s) were posted. */
```

Return Value **isc_wait_for_event()** returns the second element of the status vector. Zero indicates success. A nonzero value indicates an error. For InterBase errors, the first element of the status vector is set to 1, and the second element is set to an InterBase error code.

To check for an InterBase error, examine the first two elements of the status vector directly. For more information about examining the status vector, see **Chapter 10, "Handling Error Conditions."**

See Also **isc_event_block()**, **isc_que_events()**

APPENDIX

A
InterBase Document Conventions

This appendix describes the InterBase 5 documentation set, the printing conventions used to display information in text and in code examples, and conventions for naming database objects and files in applications.

The InterBase documentation set

The InterBase documentation set is an integrated package designed for all levels of users. It consists of five printed books. Each of these books is also provided in Adobe Acrobat PDF format and is accessible on line through the Help menu. If Adobe Acrobat is not already installed on your system, you can find it on the InterBase distribution CD-ROM or at *http//www.adobe.com/prodindex/acrobat/readstep.html*. Acrobat is available for Windows NT, Windows 95, and most flavors of UNIX. Windows users also have help available through the WinHelp system.

Book	**Description**
Operations Guide	Provides an introduction to InterBase and an explanation of tools and procedures for performing administrative tasks on databases and database servers. Also includes full reference on InterBase utilities, including isql, gbak, Server Manager for Windows, and others.
Data Definition Guide	Explains how to create, alter, and delete database objects through ISQL.
Language Reference	Describes SQL and DSQL syntax and usage.
Programmer's Guide	Describes how to write embedded SQL and DSQL database applications in a host language, precompiled through gpre.
API Guide	Explains how to write database applications using the InterBase API.

TABLE A.1 Books in the InterBase 5 documentation set

Printing conventions

The InterBase documentation set uses various typographic conventions to identify objects and syntactic elements.

The following table lists typographic conventions used in text, and provides examples of their use:

Convention	Purpose	Example
UPPERCASE	SQL keywords, SQL functions, and names of all database objects such as tables, columns, indexes, and stored procedures.	The following SELECT statement retrieves data from the CITY column in the CITIES table.
italic	New terms, emphasized words, file names, and host- language variables.	The *isc4.gdb s*ecurity database is not accessible without a valid user name and password.
bold	Utility names, user-defined functions, and host-language function names. Function names are always followed by parentheses to distinguish them from utility names.	Use **gbak** to back up and restore a database. Use the **datediff()** function to calculate the number of days between two dates.

TABLE A.2 Text conventions

Syntax conventions

The following table lists the conventions used in syntax statements and sample code, and provides examples of their use:

Convention	Purpose	Example
UPPERCASE	Keywords that must be typed exactly as they appear when used.	`SET TERM !!;`
italic	Parameters that cannot be broken into smaller units. For example, a table name cannot be subdivided.	CREATE GENERATOR *name*;
<*italic*>	Parameters in angle brackets that *can* be broken into smaller syntactic units.	WHILE (<*condition*>) DO <*compound_statement*>
[ ]	Optional syntax: you do not need to include anything that is enclosed in square brackets.	`CREATE [UNIQUE][ASCENDING|DESCENDING]`
{ }	One of the enclosed options *must* be included in actual statement use. If the contents are separated by a pipe symbol (|), you must choose only one.	`{SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION}`

|	You can choose only one of a group whose elements are separated by this pipe symbol.
	When objects separated by this symbol occur within curly brackets, you *must* choose one; when they are within square brackets you can choose one or none.
		`SET {DATABASE | SCHEMA}`
		`SELECT [DISTINCT |ALL]`

...	The clause enclosed in brackets with the ... symbol can be repeated as many times as necessary.	(<*col*> [,<*col*>...])

TABLE A.3 Syntax conventions

APPENDIX

B Data Structures

This appendix documents the data structures, compile-time constants, parameter buffers, and information buffers utilized in InterBase API applications.

This information also appears throughout the rest of this API Guide, but is consolidated here as a convenience. See other sections of this manual for more information.

All the structures and compile-time constants mentioned are defined in the *ibase.h* header file. Items are documented alphabetically, as follows:

- Array descriptor
- Blob descriptor
- Blob information item-list buffer and result buffer
- Blob parameter buffer
- Database information request buffer and result buffer
- Database parameter buffer
- SQL datatype macro constants
- Status vector
- Transaction parameter buffer
- XSQLDA and XSQLVAR structures

Array descriptor

An array descriptor ISC_ARRAY_DESC is a structure defined in the *ibase.h* header file as:

```
typedef struct {
   unsigned char array_desc_dtype;
   char array_desc_scale;
   unsigned short array_desc_length;
   char array_desc_field_name [32];
   char array_desc_relation_name [32];
   short array_desc_dimensions;
   short array_desc_flags;
   ISC_ARRAY_BOUND array_desc_bounds [16];
} ISC_ARRAY_DESC;
```

ISC_ARRAY_BOUND is defined as:

```
typedef struct {
   short array_bound_lower; /* lower bound */
   short array_bound_upper; /* upper bound */
} ISC_ARRAY_BOUND;
```

Field	**Description**
array_desc_dtype	Datatype (see below)
array_desc_scale	Scale for numeric datatypes
array_desc_length	Length in bytes of each array element
array_desc_field_name	NULL-terminated column name
array_desc_relation_name	NULL-terminated relation name
array_desc_dimensions	Number of array dimensions
array_desc_flags	Specifies whether array is to be accessed in row- major or column-major order • 0: row-major • 1: column-major
array_desc_bounds	Lower and upper bounds for each dimension

TABLE B.1 Array descriptor fields

Datatypes for array descriptors

The *array_desc_dtype* field of an array descriptor must be expressed as one of the datatypes in the following table:

array_desc_dtype value	Corresponding InterBase datatype
blr_text	CHAR
blr_text2	CHAR
blr_short	SMALLINT
blr_long	INTEGER
blr_quad	ISC_QUAD structure
blr_float	FLOAT
blr_double	DOUBLE PRECISION
blr_date	DATE
blr_varying	VARCHAR
blr_varying2	VARCHAR
blr_blob_id	ISC_QUAD structure
blr_cstring	NULL-terminated string
blr_cstring2	NULL-terminated string

TABLE B.2 Datatypes for array descriptors

Blob descriptor

A Blob descriptor is defined as:

```
typedef struct {
    short blob_desc_subtype;
    short blob_desc_charset;
    short blob_desc_segment_size;
    unsigned char blob_desc_field_name [32];
    unsigned char blob_desc_relation_name [32];
} ISC_BLOB_DESC;
```

Field	Description
blob_desc_subtype	Type of Blob data 0: unstructured 1: TEXT negative integer between –1 and –32678: user-defined subtype
blob_desc_charset	Character set (see below)
blob_desc_segment_size	Segment size
blob_desc_field_name	NULL-terminated column name
blob_desc_relation_name	NULL-terminated table name

TABLE B.3 Blob descriptor fields

Character sets

InterBase supports a number of character sets. For a list of the character sets supported, and the character set value that must be entered in the *blob_desc_charset* field of a Blob descriptor, see the *Data Definition Guide*.

Blob information buffers

The **isc_blob_info()** call enables an application to query for Blob information such as the total number of segments in the Blob, or the length of the longest segment.

isc_blob_info() requires two application-provided buffers, an item-list buffer, where the application specifies the information it needs, and a result buffer, where InterBase returns the requested information. An application populates the item-list buffer with information prior to calling **isc_blob_info()**. InterBase returns information in the result buffer. If InterBase attempts to pass back more information than can fit in the result buffer, it puts the value, *isc_info_truncated*, defined in *ibase.h*, in the final byte of the result buffer.

Item-list buffer

The item-list buffer is a byte vector into which is placed a sequence of byte values, one per requested item of information. Each byte is an *item type*, specifying the kind of information desired. Compile-time constants for all item types are defined in *ibase.h*:

```
#define isc_info_blob_num_segments        4
#define isc_info_blob_max_segment         5
#define isc_info_blob_total_length        6
#define isc_info_blob_type                7
```

Result buffer

The result buffer is a byte vector in which InterBase returns a series of *clusters* of information, one per item requested. Each cluster consists of three parts:

1. A one-byte *item type*. Each is the same as one of the item types in the item-list buffer.
2. A 2-byte number specifying the number of bytes that follow in the remainder of the cluster.
3. A *value*, stored in a variable number of bytes, whose interpretation depends on the item type.

A calling program is responsible for interpreting the contents of the result buffer and for deciphering each cluster as appropriate.

The clusters returned to the result buffer are not aligned. Furthermore, all numbers are represented in a generic format, with the least significant byte first, and the most significant byte last. Signed numbers have the sign in the last byte. Convert the numbers to a datatype native to your system, if necessary, before interpreting them. The API call, **isc_vax_integer()**, can be used to perform the conversion.

Blob buffer items

The following table lists items about which information can be requested and returned, and the values reported:

Request and return item	Return value
isc_info_blob_num_segments	Total number of segments
isc_info_blob_max_segment	Length of the longest segment
isc_info_blob_total_length	Total size, in bytes, of Blob
isc_info_blob_type	Type of Blob (0: segmented, or 1: stream)

TABLE B.4 Blob information items and return values

In addition to the information InterBase returns in response to a request, InterBase can also return one or more of the following status messages to the result buffer. Each status message is one unsigned byte in length:

Item	Description
isc_info_end	End of the messages
isc_info_truncated	Result buffer is too small to hold any more requested information
isc_info_error	Requested information is unavailable. Check the status vector for an error code and message

TABLE B.5 Status message return items

Blob parameter buffer

A *Blob Parameter Buffer* (BPB) is an application-defined byte vector, passed as an argument to **isc_open_blob2() or isc_create_blob2()**, that specifies Blob attributes required for filtering Blob data.

A BPB consists of the following parts:

1. A byte specifying the version of the parameter buffer, always the compile-time constant*, isc_bpb_version1*.
2. A contiguous series of one or more *clusters* of bytes, each describing a single parameter.

Each cluster consists of the following parts:

1. A one-byte parameter type. There are compile-time constants defined for all the parameter types (for example, *isc_bpb_target_type*).
2. A one-byte number specifying the number of bytes that follow in the remainder of the cluster.
3. A variable number of bytes, whose interpretation depends on the parameter type.

All numbers in the Blob parameter buffer must be represented in a generic format, with the least significant byte first, and the most significant byte last. Signed numbers should have the sign in the last byte. The API function **isc_vax_integer()** can be used to reverse the byte order of a number. For more information about **isc_vax_integer()**, see **"isc_vax_integer()" on page 332** of **Chapter 12, "API Function Reference."**

The following table lists the parameter types and their meaning. For lists of the possible subtypes and character sets, see "Blob descriptor" on page 345.

Parameter type	**Description**
isc_bpb_target_type	Target subtype
isc_bpb_source_type	Source subtype
isc_bpb_target_interp	Target character set
isc_bpb_source_interp	Source character set

TABLE B.6 Blob parameter buffer parameter types

Database information request buffer and result buffer

The **isc_database_info()** call enables an application to query for information about an attached database.

isc_database_info() requires two application-provided buffers, a request buffer, where the application specifies the information it needs, and a result buffer, where InterBase returns the requested information. An application populates the request buffer with information prior to calling **isc_database_info().** InterBase returns information in the result buffer. If InterBase attempts to pass back more information than can fit in the result buffer, it puts the value, *isc_info_truncated*, defined in *ibase.h*, in the final byte of the result buffer.

Request buffer

The request buffer is a byte vector into which is placed a sequence of byte values, one per requested item of information. Each byte is an *item type*, specifying the kind of information desired. Compile-time constants for all item types are defined in *ibase.h* and shown below.

Result buffer

The result buffer is a byte vector in which InterBase returns a series of *clusters* of information, one per item requested. Each cluster consists of three parts:

1. A one-byte *item return type*. These are the same as the item types specified in the request buffer.
2. A two-byte number specifying the number of bytes that follow in the remainder of the cluster.
3. A *value*, stored in a variable number of bytes, whose interpretation (e.g., as a number or as a string of characters) depends on the item return type.

A calling program is responsible for interpreting the contents of the result buffer and for deciphering each cluster as appropriate. In many cases, the value simply contains a number or a string (sequence of characters). But in other cases, the value is a number of bytes whose interpretation depends on the item return type.

The clusters returned to the result buffer are not aligned. Furthermore, all numbers are represented in a generic format, with the least significant byte first, and the most significant byte last. Signed numbers have the sign in the last byte. Convert the numbers to a datatype native to your system, if necessary, before interpreting them. The API call, **isc_vax_integer()**, can be used to perform the conversion.

In addition to the information InterBase returns in response to a request, InterBase can also return one or more of the following status messages to the result buffer. Each status message is one unsigned byte in length:

Item	Description
isc_info_end	End of the messages
isc_info_truncated	Result buffer is too small to hold any more requested information
isc_info_error	Requested information is unavailable; check the status vector for an error code and message

TABLE B.7 Status message return items

Request buffer items and result buffer values

The following sections show the request buffer items and result buffer contents for the following categories of database information:

- Database characteristics
- Environmental characteristics
- Performance statistics
- Database operation counts

Database characteristics

Several items are provided for determining database characteristics, such as its size and major and minor ODS version numbers. The following table lists the request buffer items that can be passed, and the information returned in the result buffer for each item type:

Request buffer item	**Result buffer contents**
isc_info_allocation	Number of database pages allocated
isc_info_base_level	Database version (level) number: 1 byte containing the number 1 1 byte containing the version number
isc_info_db_id	Database file name and site name: • 1 byte containing the number 2 • 1 byte containing the length, *d*, of the database file name in bytes • A string of *d* bytes, containing the database file name • 1 byte containing the length, *l*, of the site name in bytes • A string of *l* bytes, containing the site name
isc_info_implementation	Database implementation number: • 1 byte containing a 1 • 1 byte containing the implementation number • 1 byte containing a “class” number, either 1 or 12

TABLE B.8 Database information items for database characteristics

Request buffer item	Result buffer contents
isc_info_no_reserve	0 or 1 • 0 indicates space is reserved on each database page for holding backup versions of modified records [Default] • 1 indicates no space is reserved for such records
isc_info_ods_minor_version	On-disk structure (ODS) minor version number; an increase in a minor version number indicates a non-structural change, one that still allows the database to be accessed by database engines with the same major version number but possibly different minor version numbers
isc_info_ods_version	ODS major version number • Databases with different major version numbers have different physical layouts; a database engine can only access databases with a particular ODS major version number • Trying to attach to a database with a different ODS number results in an error
isc_info_page_size	Number of bytes per page of the attached database; use with *isc_info_allocation* to determine the size of the database
isc_info_version	Version identification string of the database implementation: • 1 byte containing the number 1 • 1 byte specifying the length, *n*, of the following string • *n* bytes containing the version identification string

TABLE B.8 Database information items for database characteristics (*continued*)

Environmental characteristics

Several items are provided for determining environmental characteristics, such as the amount of memory currently in use, or the number of database cache buffers currently allocated. These items are described in the following table:

Request buffer item	**Result buffer contents**
isc_info_current_memory	Amount of server memory (in bytes) currently in use
isc_info_forced_writes	Number specifying the mode in which database writes are performed (0 for asynchronous, 1 for synchronous)
isc_info_max_memory	Maximum amount of memory (in bytes) used at one time since the first process attached to the database
isc_info_num_buffers	Number of memory buffers currently allocated
isc_info_sweep_interval	Number of transactions that are committed between “sweeps” to remove database record versions that are no longer needed
isc_info_user_names	NetWare only. Names of all the users currently attached to the database; for each such user, the result buffer will contain an *isc_info_user_names* byte followed by a 1-byte length specifying the number of bytes in the user name, followed by the user name

TABLE B.9 Database information items for environmental characteristics

Not all environmental information items are available on all platforms.

Performance statistics

There are four items providing performance statistics for a database. These statistics accumulate for a database from the moment it is first attached by any process until the last remaining process detaches from the database. A program requesting this information, therefore, sees information pertaining to its own attachment and all other attachments.

For example, the value returned for *isc_info_reads* is the number of reads since the current database was first attached: it is an *aggregate* of all reads done by all attached processes, rather than the number of reads done for the calling program since it attached to the database.

The items providing performance statistics are summarized in the following table:

Request buffer item	Result buffer contents
isc_info_fetches	Number of reads from the memory buffer cache
isc_info_marks	Number of writes to the memory buffer cache
isc_info_reads	Number of page reads
isc_info_writes	Number of page writes

TABLE B.10 Database information items for performance statistics

Database operation counts

Several information items are provided for determining the number of various database operations performed by the currently attached calling program. These values are calculated on a per-table basis.

When any of these information items is requested, InterBase returns to the result buffer:

1. 1 byte specifying the item type (for example, *isc_info_insert_count*).
2. 2 bytes telling how many bytes compose the subsequent value pairs.
3. A pair of values for each table in the database on which the requested type of operation has occurred since the database was last attached.

   Each pair consists of:

   - 2 bytes specifying the table ID.
   - 4 bytes listing the number of operations (for example, inserts) done on that table.

To determine an actual table name from a table ID, query the RDB$RELATION system table.

The following table describes the items which return count values for operations on the database:

Request buffer item	Result buffer contents
isc_info_backout_count	Number of removals of a version of a record
isc_info_delete_count	Number of database deletes since the database was last attached
isc_info_expunge_count	Number of removals of a record and all of its ancestors, for records whose deletions have been committed
isc_info_insert_count	Number of inserts into the database since the database was last attached
isc_info_purge_count	Number of removals of old versions of fully mature records (records committed, resulting in older—ancestor—versions no longer being needed)
isc_info_read_idx_count	Number of reads done via an index since the database was last attached
isc_info_read_seq_count	Number of sequential database reads, i.e., the number of sequential table scans (row reads) done on each table since the database was last attached
isc_info_update_count	Number of database updates since the database was last attached

TABLE B.11 Database information items for operation counts

A *Database Parameter Buffer* (DPB) is an application-defined byte vector, passed as an argument to **isc_attach_database()**, that specifies desired database characteristics.

A DPB consists of the following parts:

1. A byte specifying the version of the parameter buffer, always the compile-time constant*, isc_dpb_version1*.
2. A contiguous series of one or more *clusters* of bytes, each describing a single parameter.

Each cluster consists of the following parts:

1. A one-byte parameter type. There are compile-time constants defined for all the parameter types (for example, *isc_dpb_num_buffers*).
2. A one-byte number specifying the number of bytes that follow in the remainder of the cluster.

3. A variable number of bytes, whose interpretation (as a number or as a string of characters, for example) depends on the parameter type.

The following table lists DPB items by purpose:

User Validation	**Item**
User name	*isc_dpb_user_name*
Password	*isc_dpb_password*
Encrypted password	*isc_dpb_password_enc*
System database administrator's user name	*isc_dpb_sys_user_name*
Authorization key for a software license	*isc_dpb_license*
Database encryption key	*isc_dpb_encrypt_key*
Environmental control	
Number of cache buffers	*isc_dpb_num_buffers*
dbkey context scope	*isc_dpb_dbkey_scope*
System management	
Force writes to the database to be done asynchronously or synchronously	*isc_dpb_force_write*
Specify whether or not to reserve a small amount of space on each database page for holding backup versions of records when modifications are made.	*isc_dpb_no_reserve*
Specify whether or not the database should be marked as damaged	*isc_dpb_damaged*
Perform consistency checking of internal structures	*isc_dpb_verify*
Shadow control	
Activate the database shadow, an optional, duplicate, in-sync copy of the database	*isc_dpb_activate_shadow*
Delete the database shadow	*isc_dpb_delete_shadow*

TABLE B.12 DPB parameters

User Validation	Item
Replay logging system control	
Activate a replay logging system to keep track of all database calls	*isc_dpb_begin_log*
Deactivate the replay logging system	*isc_dpb_quit_log*
Character set and message file specification	
Language-specific message file	*isc_dpb_lc_messages*
Character set to be utilized	*isc_dpb_lc_ctype*

TABLE B.12 DPB parameters (*continued*)

The following table lists DPB parameters in alphabetical order. For each parameter, it lists its purpose, the length, in bytes, of any values passed with the parameter, and the value to pass:

Parameter	Purpose	Length	Value
isc_dpb_activate_shadow	Directive to activate the database shadow, which is an optional, duplicate, in-sync copy of the database	1 (Ignored)	0 (Ignored)
isc_dpb_damaged	Number signifying whether or not the database should be marked as damaged; 1 = mark as damaged, 0 = do not mark as damaged	1	0 or 1
isc_dpb_dbkey_scope	Scope of *dbkey* context; 0 limits scope to the current transaction, 1 extends scope to the database session.	1	0 or 1
isc_dpb_delete_shadow	Directive to delete a database shadow that is no longer needed	1 (Ignored)	0 (Ignored)
isc_dpb_encrypt_key	String encryption key, up to 255 characters	Number of bytes in string	String containing key
isc_dpb_force_write	Specifies whether database writes are synchronous or asynchronous; 0 = asynchronous, 1 = synchronous	1	0 or 1

TABLE B.13 Alphabetical list of DPB parameters

Parameter	Purpose	Length	Value
isc_dpb_lc_ctype	String specifying the character set to be utilized	Number of bytes in string	String containing character set name
isc_dpb_lc_messages	String specifying a language-specific message file	Number of bytes in string	String containing message file name
isc_dpb_license	String authorization key for a software license	Number of bytes in string	String containing key
isc_dpb_no_reserve	Specifies whether or not a small amount of space on each database page is reserved for holding backup versions of records when modifications are made; keep backup versions on the same page as the primary record to optimize update activity 0 (default) = reserve space 1= do not reserve space	1	0 or 1
isc_dpb_num_buffers	Number of database cache buffers to allocate for use with the database Default = 75	1	Number of buffers to allocate
isc_dpb_password	String password, up to 255 characters	Number of bytes in string	String containing password
isc_dpb_password_enc	String encrypted password, up to 255 characters	Number of bytes in string	String containing password
isc_dpb_sys_user_name	String system DBA name, up to 255 characters	Number of bytes in string	String containing SYSDBA name
isc_dpb_user_name	String user name, up to 255 characters	Number of bytes in string	String containing user name

TABLE B.13 Alphabetical list of DPB parameters (*continued*)

Some parameters, such as *isc_dpb_delete_shadow*, are directives that do not require additional parameters. Even so, you *must* still provide length and value bytes for these parameters. Set length to 1, and value to 0. InterBase ignores these parameter values, but they are required to maintain the format of the DPB.

SQL datatype macro constants

InterBase defines a set of macro constants to represent SQL datatypes and NULL status information in an XSQLVAR. An application should use these macro constants to specify the datatype of parameters and to determine the datatypes of select-list items in an SQL statement. The following table lists each SQL datatype, its corresponding macro constant expression, C datatype or InterBase typedef, and whether or not the *sqlind* field is used to indicate a parameter or variable that contains NULL or unknown data:

SQL datatype	Macro expression	C datatype or typedef	*sqlind* used?
Array	SQL_ARRAY	ISC_QUAD	No
Array	SQL_ARRAY + 1	ISC_QUAD	Yes
Blob	SQL_BLOB	ISC_QUAD	No
Blob	SQL_BLOB + 1	ISC_QUAD	Yes
CHAR	SQL_TEXT	char[ ]	No
CHAR	SQL_TEXT + 1	char[ ]	Yes
DATE	SQL_DATE	ISC_QUAD	No
DATE	SQL_DATE + 1	ISC_QUAD	Yes
DECIMAL	SQL_SHORT, SQL_LONG, or SQL_DOUBLE	int, long, or double	No
DECIMAL	SQL_SHORT + 1, SQL_LONG + 1, or SQL_DOUBLE + 1	int, long, or double	Yes
DOUBLE PRECISON	SQL_DOUBLE	double	No
DOUBLE PRECISION	SQL_DOUBLE + 1	double	Yes
INTEGER	SQL_LONG	long	No
INTEGER	SQL_LONG + 1	long	Yes
FLOAT	SQL_FLOAT	float	No
FLOAT	SQL_FLOAT + 1	float	Yes

TABLE B.14 SQL datatypes, macro expressions, and C datatypes

SQL datatype	Macro expression	C datatype or typedef	*sqlind* used?
NUMERIC	SQL_SHORT, SQL_LONG, or SQL_DOUBLE	int, long, or double	No
NUMERIC	SQL_SHORT + 1, SQL_LONG + 1, or SQL_DOUBLE + 1	int, long, or double	Yes
SMALLINT	SQL_SHORT	short	No
SMALLINT	SQL_SHORT + 1	short	Yes
VARCHAR	SQL_VARYING	First 2 bytes: short containing the length of the character string. Remaining bytes: char[ ]	No
VARCHAR	SQL_VARYING + 1	First 2 bytes: short containing the length of the character string. Remaining bytes: char[ ]	Yes

TABLE B.14 SQL datatypes, macro expressions, and C datatypes (*continued*)

DECIMAL and NUMERIC datatypes are stored internally as SMALLINT, INTEGER, or DOUBLE PRECISION datatypes. To specify the correct macro expression to provide for a DECIMAL or NUMERIC column, use **isql** to examine the column definition in the table to see how InterBase is storing column data, then choose a corresponding macro expression.

The datatype information for a parameter or select-list item is contained in the *sqltype* field of the XSQLVAR structure. The value contained in *sqltype* provides two pieces of information:

- The datatype of the parameter or select-list item.
- Whether *sqlind* is used to indicate NULL values. If *sqlind* is used, its value specifies whether the parameter or select-list item is NULL (-1), or not NULL (0).

For example, if *sqltype* equals SQL_TEXT, the parameter or select-list item is a CHAR that does not use *sqlind* to check for a NULL value (because, in theory, NULL values are not allowed for it). If *sqltype* equals SQL_TEXT + 1, then *sqlind* can be checked to see if the parameter or select-list item is NULL.

The C language expression, *sqltype & 1*, provides a useful test of whether a parameter or select-list item can contain a NULL. The expression evaluates to 0 if the parameter or select-list item cannot contain a NULL, and 1 if the parameter or select-list item can contain a NULL.

Status vector

Most API functions return status information that indicates success or failure. Status information is reported in an error status vector, declared in applications as an array of twenty long integers, using the following syntax:

```
#include "ibase.h"
. . .
ISC_STATUS status_vector[20];
```

If you plan to write your own routines instead of the InterBase error-handling routines to read and react to the contents of the status vector, you need to know how to interpret it.

InterBase stores error information in the status vector in *clusters* of two or three longs. The first cluster in the status vector *always* indicates the primary cause of the error. Subsequent clusters may contain supporting information about the error, for example, strings or numbers for display in an associated error message. The actual number of clusters used to report supporting information varies from error to error.

In many cases, additional errors may be reported in the status vector. Additional errors are reported immediately following the first error and its supporting information, if any. The first cluster for each additional error message identifies the error. Subsequent clusters may contain supporting information about the error.

Meaning of the first long in a cluster

The first long in any cluster is a *numeric descriptor*. By examining the numeric descriptor for a cluster, you can always determine the:

- Total number of longs in the cluster.
- Kind of information reported in the remainder of the cluster.
- Starting location of the next cluster in the status vector.

The following table lists possible values for the first long in any cluster in the status vector. Note that the first cluster in the status vector can only contain values of 0, 1, or greater than 4:

Value	Longwords in cluster	Meaning
0	—	End of error information in the status vector
1	2	Second long is an InterBase error code
2	2	Second long is the address of string used as a replaceable parameter in a generic InterBase error message
3	3	Second long is the length, in bytes, of a variable length string provided by the operating system (most often this string is a file name); third long is the address of the string
4	2	Second long is a number used as a replaceable parameter in a generic InterBase error message
5	2	Second long is the address of an error message string requiring no further processing before display
6	2	Second long is a VAX/VMS error code
7	2	Second long is a Unix error code
8	2	Second long is an Apollo Domain error code
9	2	Second long is an MS-DOS or OS/2 error code
10	2	Second long is an HP MPE/XL error code
11	2	Second long is an HP MPE/XL IPC error code
12	2	Second long is a NeXT/Mach error code

NOTE As InterBase is adapted to run on other hardware and software platforms, additional numeric descriptors for specific platform and operating system error codes may be added to the end of this list.

TABLE B.15 Interpretation of status vector clusters

The following table lists the *ibase.h* #define equivalents of each numeric descriptor:

Value	#define	Value	#define
0	*isc_arg_end*	8	*isc_arg_domain*
1	*isc_arg_gds*	9	*isc_arg_dos*
2	*isc_arg_string*	10	*isc_arg_mpexl*
3	*isc_arg_cstring*	11	*isc_arg_mpexl_ipc*
4	*isc_arg_number*	15	*isc_arg_next_mach*
5	*isc_arg_interpreted*	16	*isc_arg_netware*
6	*isc_arg_vms*	17	*isc_arg_win32*
7	*isc_arg_unix*		

TABLE B.16 #defines for status vector numeric descriptors

Transaction parameter buffer

The *transaction parameter buffer* (TPB) is an optional, application-defined byte vector, passed as an argument to **isc_start_transaction()**, that sets up a transaction's *attributes*, its operating characteristics, such as whether the transaction has read and write access to tables, or read-only access, and whether or not other simultaneously active transactions can share table access with the transaction. Each transaction may have its own TPB, or transactions that share operating characteristics can use the same TPB.

If a TPB is not created for a transaction, a NULL pointer must be passed to **isc_start_transaction()** in its place. A default set of attributes is automatically assigned to such transactions.

A TPB is declared in a C program as a char array of one-byte elements. Each element is a parameter that describes a single transaction attribute. The first element in every TPB must be the *isc_tpb_version3* constant. The following table lists available TPB constants, describes their purposes, and indicates which constants are assigned as a default set of attributes when a NULL TPB pointer is passed to **isc_start_transaction()**:

Parameter	Description
isc_tpb_version3	InterBase version 3 transaction
isc_tpb_consistency	Table-locking transaction model
isc_tpb_concurrency	High throughput, high concurrency transaction with acceptable consistency; use of this parameter takes full advantage of the InterBase multi-generational transaction model [Default]
isc_tpb_shared	Concurrent, shared access of a specified table among all transactions.; use in conjunction with *isc_tpb_lock_read* and *isc_tpb_lock_write* to establish the lock option [Default]
isc_tpb_protected	Concurrent, restricted access of a specified table; use in conjunction with *isc_tpb_lock_read* and *isc_tpb_lock_write* to establish the lock option
isc_tpb_wait	Lock resolution specifies that the transaction is to wait until locked resources are released before retrying an operation [Default]
isc_tpb_nowait	Lock resolution specifies that the transaction is not to wait for locks to be released, but instead, a lock conflict error should be returned immediately
isc_tpb_read	Access mode of read-only that allows a transaction only to select data from tables
isc_tpb_write	Access mode of read-write that allows a transaction to select, insert, update, and delete table data [Default]
isc_tpb_lock_read	Read-only access of a specified table; use in conjunction with *isc_tpb_shared*, *isc_tpb_protected*, and *isc_tpb_exclusive* to establish the lock option
isc_tpb_lock_write	Read-write access of a specified table; use in conjunction with *isc_tpb_shared*, *isc_tpb_protected*, and *isc_tpb_exclusive* to establish the lock option [Default]

TABLE B.17 TPB constants

Parameter	Description
isc_tpb_read_committed	High throughput, high concurrency transaction that can read changes committed by other concurrent transactions; use of this parameter takes full advantage of the InterBase multi-generational transaction model
isc_tpb_rec_version	Enables an *isc_tpb_read_committed* transaction to read the most recently committed version of a record even if other, uncommitted versions are pending.
isc_tpb_no_rec_version	Enables an *isc_tpb_read_committed* transaction to read only the latest committed version of a record If an uncommitted version of a record is pending and *isc_tpb_wait* is also specified, then the transaction waits for the pending record to be committed or rolled back before proceeding; otherwise, a lock conflict error is reported at once

TABLE B.17 TPB constants (*continued*)

TPB parameters specify the following classes of information:

- *Transaction version number* is used internally by the InterBase engine. It is always be the first attribute specified in the TPB, and must always be set to *isc_tpb_version3.*
- *Access mode* describes the actions that can be performed by the functions associated with the transaction. Valid access modes are:

  isc_tpb_read

  isc_tpb_write
- *Isolation level* describes the view of the database given a transaction as it relates to actions performed by other simultaneously occurring transactions. Valid isolation levels are:

  isc_tpb_concurrency

  isc_tpb_consistency

  isc_tpb_read_committed, *isc_tpb_rec_version*

  isc_tpb_read_committed, *isc_tpb_no_rec_version*
- *Lock resolution* describes how a transaction should react if a lock conflict occurs. Valid lock resolutions are:

  isc_tpb_wait

  isc_tpb_nowait

- *Table reservation* optionally describes an access method and lock resolution for a specified table that the transaction accesses. When table reservation is used, tables are reserved for the specified access when the transaction is started, rather than when the transaction actually accesses the table. Valid reservations are:

  isc_tpb_shared, *isc_tpb_lock_write*

  isc_tpb_shared, *isc_tpb_lock_read*

  isc_tpb_protected, *isc_tpb_lock_write*

  isc_tpb_protected, *isc_tpb_lock_read*

XSQLDA and XSQLVAR

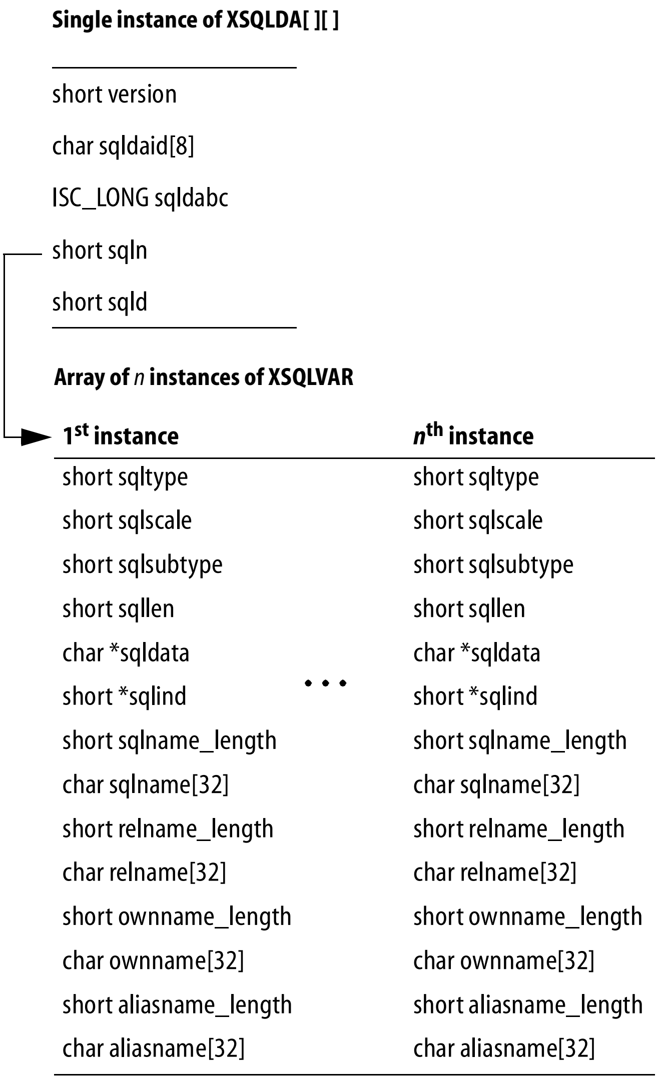
All DSQL applications must declare one or more extended SQL descriptor areas (XSQLDAs).

The XSQLDA is a host-language data structure that DSQL uses to transport data to or from a database when processing an SQL statement string. There are two types of XSQLDAs: *input* descriptors and *output* descriptors. Both input and output descriptors are implemented using the XSQLDA structure.

Syntax One field in the XSQLDA, *sqlvar*, is an XSQLVAR structure. The *sqlvar* is especially important because one XSQLVAR must be defined for each input parameter or column returned.

Applications do not declare instances of the XSQLVAR ahead of time, but must, instead, dynamically allocate storage for the proper number of XSQLVAR structures required for each DSQL statement before it is executed, then deallocate it, as appropriate, after statement execution.

The following figure illustrates the relationship between the XSQLDA and the XSQLVAR.



An input XSQLDA consists of a single XSQLDA structure, and one XSQLVAR structure for each input parameter. An output XSQLDA also consists of one XSQLDA structure and one XSQLVAR structure for each data item returned by the statement.

The **isc_dsql_prepare()**, **isc_dsql_describe()**, and **isc_dsql_describe_bind()** statements can be used to determine the proper number of XSQLVAR structures to allocate, and the XSQLDA_LENGTH macro can be used to allocate the proper amount of space.

XSQLDA field descriptions

The following table describes the fields that comprise the XSQLDA structure:

Field definition	Description
short version	Indicates the version of the XSQLDA structure; set by an application The current version is defined in *ibase.h* as SQLDA_VERSION1
char sqldaid[8]	Reserved for future use
ISC_LONG sqldabc	Reserved for future use
short sqln	Indicates the number of elements in the sqlvar array; the application should set this field whenever it allocates storage for a descriptor
short sqld	Indicates the number of parameters for an input XSQLDA, or the number of select-list items for an output XSQLDA; set by InterBase during an **isc_dsql_prepare, isc_dsql_describe(),** or **isc_dsql_describe_bind()** For an input descriptor, an sqld of 0 indicates that the SQL statement has no parameters; for an output descriptor, an sqld of 0 indicates that the SQL statement is not a SELECT statement
XSQLVAR sqlvar	The array of XSQLVAR structures; the number of elements in the array is specified in the sqln field

TABLE B.18 XSQLDA field descriptions

XSQLVAR field descriptions

The following table describes the fields that comprise the XSQLVAR structure:

Field definition	Description
short *sqltype*	Indicates the SQL datatype of parameters or select-list items; set by InterBase during **isc_dsql_prepare, isc_dsql_describe**(), or **isc_dsql_describe_bind()**
short *sqlscale*	Provides scale, specified as a negative number, for exact numeric datatypes (DECIMAL, NUMERIC); set by InterBase during **isc_dsql_prepare, isc_dsql_describe(),** or **isc_dsql_describe_bind()**
short *sqlsubtype*	Specifies the subtype for Blob data; set by InterBase during **isc_dsql_prepare, isc_dsql_describe(),** or **isc_dsql_describe_bind()**
short *sqllen*	Indicates the maximum size, in bytes, of data in the sqldata field; set by InterBase during **isc_dsql_prepare, isc_dsql_describe(),** or **isc_dsql_describe_bind()**
char **sqldata*	For input descriptors, specifies either the address of a select-list item or a parameter; set by the application For output descriptors, contains a value for a select-list item; set by InterBase
short **sqlind*	On input, specifies the address of an indicator variable; set by an application On output, specifies the address of column indicator value for a select-list item following a FETCH A value of 0 indicates that the column is not NULL; a value of –1 indicates the column is NULL; set by InterBase
short *sqlname_length*	Specifies the length, in bytes, of the data in field, sqlname; set by InterBase during **isc_dsql_prepare()** or **isc_dsql_describe()**
char *sqlname*[32]	Contains the name of the column. Not NULL (\0) terminated; set by InterBase during **isc_dsql_prepare()** or **isc_dsql_describe()**
short *relname_length*	Specifies the length, in bytes, of the data in field, relname; set by InterBase during **isc_dsql_prepare()** or **isc_dsql_describe()**

TABLE B.19 XSQLVAR field descriptions

Field definition	Description
char *relname*[32]	Contains the name of the table; not NULL (\0) terminated, set by InterBase during **isc_dsql_prepare()** or **isc_dsql_describe()**
short *ownname_length*	Specifies the length, in bytes, of the data in field, ownname; set by InterBase during **isc_dsql_prepare()** or **isc_dsql_describe()**
char *ownname*[32]	Contains the name of the table owner; not NULL (\0) terminated, set by InterBase during **isc_dsql_prepare()** or **isc_dsql_describe()**
short *aliasname_length*	Specifies the length, in bytes, of the data in field, aliasname; set by InterBase during **isc_dsql_prepare()** or **isc_dsql_describe()**
char *aliasname*[32]	Contains the alias name of the column or the column name if no alias exists; not NULL (\0) terminated, set by InterBase during **isc_dsql_prepare()** or **isc_dsql_describe()**

TABLE B.19 XSQLVAR field descriptions (*continued*)

Index