MWA Software

Firebird Pascal API Guide

Issue 1.2,
13 March 2017

McCallum Whyman Associates Ltd

EMail: info@ mccallumwhyman.com, http://www.mccallumwhyman.com

Registered in England Registration No. 2624328

COPYRIGHT

Disclaimer

Although our best efforts have been made to ensure that the information contained within is up-to-date and accurate, no warranty whatsover is offered as to its correctness and readers are responsible for ensuring through testing or any other appropriate procedures that the information provided is correct and appropriate for the purpose for which it is used.

CONTENTS **Page**

1

Introduction

The Firebird Pascal API Guide is a guide to the Firebird API created by MWA Software as Pascal Language Bindings for accessing the Firebird Client API from a Pascal Program. The purpose of these language bindings is to provide the API in a format where all data types for SQL data, interface parameters and results are native Pascal types. The Pascal API is pitched at a similar level and purpose to the IBPP Firebird API provided to the C++ world. The package is intended to be suitable for use on any platform supported by the Free Pascal Compiler. The package is simply known by the abbreviation *fbintf*. It is a required dependency for version 2 onwards of *IBX for Lazarus.*

From release 2.0.2 onwards, *fbintf* also supports the Delphi Win32 compiler.

The API is intended to be simple to use and to place the minimum burden on the API user when it comes to managing the Firebird client library and the various Pascal objects that are created to provide the API. It is implemented as reference counted COM interfaces which, for the API user, are as easy to use as other managed types such as AnsiStrings and dynamic arrays. The user only needs to worry about accessing and using the interface; disposing of interfaces is performed automatically whenever an interface goes out of scope.

Two interface implementations are provided. One is for the new Firebird 3 Client API and the other for the legacy Firebird Client API used for Firebird 2.x and earlier. The Firebird 3 API implementation is used whenever possible and the older API only if the Firebird 3 API is not available (see also 2.5).

The remainder of this guide is concerned with the Installation of the language bindings and how the Pascal API is used. The organisation of this guide has been deliberately based on the InterBase 6 API Guide. This is still the primary reference for the legacy Firebird 'C' API and provides a greater depth of discussion than this guide is intended to provide. Readers may occasionally find it useful to refer to the InterBase 6 API Guide and by using the same chapter headings the intention is to provide easy cross-reference.

The motivations for developing these language bindings are:

- To provide a route for the updating of the *IBX for Lazarus* package to support the new Firebird 3 API as well as providing continued support for the legacy Firebird API without having to separately maintain two codebases.
- To provide a standard FCL level Firebird API for use with *Free Pascal* (FPC) and Delphi without requiring the additional complexity introduced by the TDataset model.
- To provide access to the Firebird API using Pascal native data types without requiring the user to be aware of bit orders or actual encodings.
- To provide a complete implementation of the Firebird API in Pascal.

This API is offered to the community as a standard Pascal API for all versions of the Firebird Relational Database.

1.1 References

1. InterBase 6 API Guide (http://www.ibphoenix.com/files/60ApiGuide.zip)
2. Firebird 2.5 Language Reference (http://firebirdsql.org/file/documentation/reference_manuals/fblangref25-en/html/fblangref25.html)
3. InterBase 6 Data Definition Guide (http://www.ibphoenix.com/files/60DataDef.zip)
4. Firebird 3.0.1 Release Notes (http://www.firebirdsql.org/file/documentation/release_notes/html/en/3_0/rlsnotes30.html
5. IBX for Lazarus (MWA Software – http://www.mwasoftware.co.uk/ibx)

1.2 Change History

1.2.1 Version 1.1

This version has been updated to include:

- API changes to IFirebirdAPI.CreateDatabase
- API addition: IStatement.GetPerfStatistics and IStatement.EnableStatistics
- API Addition: IAttachment.GetArrayMetaData
- API Version Number added.
- Clarification on the handling of Firebird Character set “NONE”. (see 9.4).

1.2.2 Version 1.2

Version 1.2 is updated to include API changes and guidance resulting from code changes to support the Delphi Win32 compiler. This includes:

- All units now compiled using “mode delphi”.
- The AnsiString type is now used instead of the default “string” type. For FPC this is type compatible with the previous version and implies no change. For Delphi, this enforces AnsiString as the interface string type instead of the UTF-16 unicodestring that is used by default for Delphi. As Firebird does not support UTF-16 and the preferred encoding is UTF-8, the choice of AnsiString is thus appropriate for a Firebird interface.
- Delphi installation instructions.

2

Installation and Preparation for Use

The software is provided as a source code product only and distributed under the InterBase Public License and the compatible Initial Developer's Public License. Copies of both of these licences are included as part of the source code package. The package itself is a compressed archive in either tar.gz format or .zip format.

In order to use the Firebird Pascal API for development or operationally, the Firebird Client library must also be installed on the same system.

2.1 Installation under Lazarus

If you are also using *IBX for Lazarus* then the *fbintf* package is automatically installed with IBX. The following instructions are only relevant when installing the *fbintf* package without IBX.

To install under Lazarus, you must first expand the archive file in some suitable and permanent location. This could be the Lazarus component directory, or some other directory that you set aside for third party components.

Now open the Lazarus IDE and select the 'Package->Open Package File (*.lpk)' menu item. Now locate and open the 'fbintf.lpk' file which should be found in the directory into which you expanded the archive and within the 'fbintf' directory. The Package Manager Dialog should now appear as shown below.

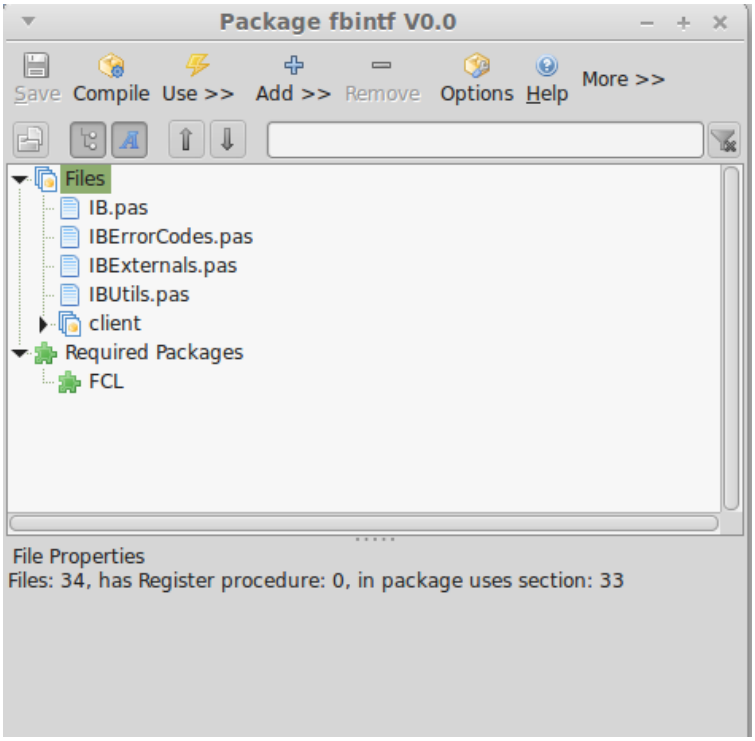

Illustration 1: The Package Manager

Click on the compile button to install. The package should compile without errors and is available for immediate use. The package manager dialog may now be closed.

Note that the package is run time only and does not need to be installed into the IDE.

2.2 Installation under FPC

You can alternatively install the Firebird Pascal API as part of the FPC library and hence available outside of Lazarus. Note that this is an alternative and if this installation approach is taken, the package should not be installed under Lazarus. This option is intended for knowledgeable users only.

The *fbintf* archive should be expanded into some temporary location. The top level directory includes a “Makefile.fpc” file, and this can be used to create a Makefile suitable for the target platform using the fpcmake utility. The “make” command can then be used to generate the compiled units in the “lib” directory. These are optimised units with no debug information. The object files can then be copied to your FPC library files directory.

For example, if you are running on Linux on an amd64 platform then the compiled units will be found in the **./lib/x86_64-linux** directory after running “make”. Assuming fpc 3.0.0, all fpc object files are located in

/usr/lib/fpc/3.0.0/units/x86_64-linux

You may create a directory within this location for fbintf and copy the object files to it. For example:

```
export FPCDIR=/usr/lib/fpc/`fpc -iV`
fpcmake
make
sudo mkdir /usr/lib/fpc/3.0.0/units/x86_64-linux/fbintf
sudo cp ./lib/x86_64-linux/* /usr/lib/fpc/3.0.0/units/x86_64-linux/fbintf
```

The files used to build fbintf may now be removed.

2.3 Installation under Delphi

Under Delphi, *fbintf* may be built and used as a runtime package.

To install under Delphi, you must first expand the archive file in some suitable and permanent location. This could be the Delphi directory, or some other directory that you set aside for third party components.

To build *fbintf* as a runtime package, open the fbintf.dproj file in the Delphi IDE and, in the Project Manager window, right click on the “fbintf.bpl” entry and select “Build” from the drop down list. By default, Delphi should compile the package and save it as “fbintf.bpl” in the package's installation directory.

To use *fbintf* in your project, open your project in the Delphi IDE and, in the Project Manager window, right click on the project name and select “options”. In the Project Options dialog, select “Run-time Package” in the left hand window and add *fbintf* to the list of packages in the right hand window.

If you select the list of run-time packages in the right hand window, a button should appear at the right of the line. Click on this button and the “Run Time Package” dialog appears. Click on the folder button and navigate to and select the “fbintf.bpl” file. Now click on the “Add” button to add *fbintf* to the list of run time packages.

When you deploy your program, remember to include the “fbintf.bpl” file in the program's application folder.

2.4 Installing Firebird

You need access to a minimum of the Firebird Client library in order to use the *fbintf* package. This applies to both development and deployment. Guidelines for deployment are give in chapter 13.

On a development system, the recommended approach is to download a pre-compiled installation package from http://www.firebirdsql.org and install the full system including examples. This will ensure that the example “employee” database is both installed and available for use by the *fbintf* testsuite, and a local server is available for testing. Firebird installation packages are available for both Linux and Windows as will as OSX.

With Linux, it is also possible to use the packages provided with your distribution. However, these will not necessarily be up-to-date. Under Debian/Ubuntu the example database is also provided as a separate package and you will need to install this package as well as unpack the database from a gzip archive and set the access permissions correctly before running the test suite. Paradoxically, unless you are very familiar with Firebird and Linux, it is often easier to install the firebirdsql package than the one from your distro.

After installation, you should check that the “employee” is correctly listed in the “aliases.conf file in the Firebird installation folder. For example, with 32-bit Firebird under Windows, the file

```
C:\Program Files (x86)\Firebird\Firebird_2_5\aliases.conf
```

should contain the line:

```
employee = C:\Program Files (x86)\Firebird\Firebird_2_5\examples\empbuild\employee.fdb
```

2.5 Which Firebird API?

Firebird 3 introduces a new API while continuing support for the legacy API. Older versions only support the legacy API. By default the *fbintf* package provides implementation support for both APIs. The Firebird 3 API is used if available and the legacy API if not.

It is possible to limit *fbintf* at compile time to one or other API. This means that the choice is fully predictable and avoids having to compile both APIs into the same program, whilst limiting your application as to which versions of Firebird it is compatible with. However, if you know that you are (e.g.) always going to ship with Firebird 3, then it may well make sense to limit the API choice at compile time.

The compile time choice is made by defined symbols located at the head of the “IB.pas” file. These are:

```
{$DEFINE USEFIREBIRD3API}
{$DEFINE USELEGACYFIREBIRDAPI}
```

Simply remove or comment out one or other of these symbols (e.g. by inserting a space between { and $ characters) in order to limit the choice of API. For example, modifying the above to:

```
{$DEFINE USEFIREBIRD3API}
{ $DEFINE USELEGACYFIREBIRDAPI}
```

will ensure that when compiled, only the Firebird 3 API is available for use.

3

Programming with the Firebird Pascal API

There are no LCL dependencies and the Firebird Pascal API may be used from the Lazarus IDE or any other development environment for FPC.

3.1 Using the API in your Project

If the package has been installed under Lazarus then you need to add the *fbintf* package to the list of required packages for your application. The API creates additional threads in order to manage Firebird Events and hence the Project's custom options should include "**-dUseCThreads**".

If you are developing a console mode Pascal program outside of Lazarus then you should include the "cthreads" unit as the first unit in your program file's uses clause.

All units that access the Firebird Client API must include the "IB" unit in their uses clause. Units that make use of symbolic constants for Firebird Engine error codes should also include the "IBErrorCodes" unit in their uses clause. These units were originally part of IBX and their names reflect their origin.

3.2 Accessing the API

The **IFirebirdAPI** interface provides access to the FirebirdClientAPI. This, like all interfaces provided by the API, is reference counted and hence automatically managed. The interface is released when it goes out of scope and the interface user is not required to release or free the interface.

This interface is provided by the function:

```
function FirebirdAPI: IFirebirdAPI;
```

The first time the function is called, it locates and loads the Firebird Client Library and then determines which version of the Firebird API to use. If it can, it will load the Firebird 3 Client API, otherwise and if this is not available, it will load the Firebird legacy API. A reference to the loaded API is then returned. On subsequent calls to the function, the currently loaded API is always returned.

If the function is unable to load the API, an exception is raised.

3.3 Locating the Firebird Client Library

The location of the Firebird Client Library depends upon the platform and the algorithm used is different for Linux, Windows and Darwin. Each is discussed below. It is also possible to override the default library name list (see 3.3.4).

3.3.1 Under Linux

The default list of Firebird Client Library names is given as a colon separated list:

For the Firebird 3 API:

```
libfbclient.so:libfbclient.so.2
```

For the Legacy API:

```
libfbembed.so:libfbembed.so.2.5:libfbembed.so.2.1:libfbclient.so:libfbclient.so.2
```

The FirebirdAPI function will try to load each in turn until it is successful. The Linux loader will, in turn, look in the standard locations for the library. If the library is in a non-standard location then this can be indicated by setting the LD_LIBRARY_PATH environment variable prior running the program. e.g.

```
export LD_LIBRARY_PATH=/opt/firebird/lib:$LD_LIBRARY_PATH
```

The above can be run as part of a shell script and extends the exist path by telling the Linux loader to look in “/opt/firebird/lib”. This has been chosen as an example, as it is a common location when Firebird is installed from a package[1] downloaded from http://www.firebirdsql.org.

3.3.2 Under Windows

The Firebird Pascal API uses the following algorithm to locate the Firebird DLL. The algorithm terminates as soon as the library has been located:

1. When the Firebird Library is to be loaded, *fbintf* first looks in the same folder as the application executable is located. It checks to see if *fbembed.dll* (the embedded server DLL) is present here. If it is then this is loaded. If not then it checks to see if *fbclient.dll* is present. If so, then it is loaded.

   In the latter case, *fbintf* also sets the FIREBIRD environment variable to the path to this folder, prior to loading the library. This has the effect of forcing the Firebird Client to look for the *firebird.conf* and *firebird.msg* files in the same folder. They must thus also be installed here. This is to ensure that the DLL uses the correct versions of these files. If the FIREBIRD environment variable is not set then the DLL will use the Windows registry

[1]Note that if you install using the installation script provided with Firebird then the library files are installed in a standard location and there is no need to set the LD_LIBRARY_PATH variable.

to find the files. If another Firebird installation is present on the same system this may point to a different version of these files.

2. If the FIREBIRD environment variable is set (prior to step 1) then the directory this points to is searched for the FB Client DLL and then the underlying "bin" directory

3. *fbintf* uses the Windows Registry to locate the most recent Firebird installation. It opens the registry key HKEY_LOCAL_MACHINE\SOFTWARE\Firebird Project\Firebird Server\Instances, and then reads the “Default Instance” string value. This is then assumed to be the full path Firebird installation. If the fbclient dll is present in this folder's “bin” subfolder, then the DLL is loaded.

   Note that in this case, the FIREBIRD environment variable is not set as the Firebird Client Library will also use the same registry entries to locate its support files.

4. *fbintf* now looks in the default installation folders for first Firebird 3.0, then Firebird 2.5 and finally Firebird 2.1. These are <Program Files Folder>\Firebird\Firebird_2_x

5. *fbintf* then uses the Windows Path to search for and load *fbclient.dll*.

6. If the DLL is still not found, then in quiet desperation, *fbintf* will attempt to load the legacy InterBase *gds32.dll* again using the Windows Search Path.

In practice, case 1 above should be used for deployed applications, whilst case 3 is the preferred approach for a development system. Case 2 is a special case for unusual installations, while the remaining cases are really attempts to get something to work on a broken system.

3.3.3 Under Darwin (OSX)

Darwin is treated as an extension of the Unix algorithm. If the standard unix search algorithm fails to the find the Firebird library then the loader will try:

/Library/Frameworks/Firebird.framework/Firebird

and then

/Library/Frameworks/Firebird.framework/Libraries/libfbclient.dylib

in the hope of finding the Firebird client library.

3.3.4 Overriding the Default Library Name

In cases where the above algorithm will fail to find the Firebird Client library then two approaches are available to explicitly direct *fbintf* to the Firebird Client library instead of using the above algorithm.

3.3.4.1 The FBLIB Environment Variable

If this environment variable is set, then it is assumed to identify by an absolute or relative path, the pathname of the Firebird Client library. The *fbintf* loader will try to load this library. If this fails then no further attempt is made to load the Firebird Client Library.

Note that this feature has to be explicitly enabled. The **AllowUseOfFBLIB** variable is defined in the IB unit and defaults to false. It must be set to true before the Firebird Pascal Client API is accessed in order to enable use of the FBLIB environment variable.

3.3.4.2 The OnGetLibraryName Event Handler

The **OnGetLibraryName** event handler is defined in the IB unit and has the type:

```
TOnGetLibraryName = procedure(var libname: string);
```

If this event handler is set before the first call the the Firebird Pascal Client API then it is called and should return the absolute or relative path, the pathname of the Firebird Client library. The *fbintf* loader will try to load this library. If this fails then no further attempt is made to load the Firebird Client Library. If the event handler returns an empty libname then it is ignored.

3.4 API Version Number

The IB.pas file includes API version information as compile time constants. These can be referenced from other units to modify behaviour according to the API Version used.

These constants are:

```
  FBIntf_Major = 1;
  FBIntf_Minor = 0;
  FBIntf_Release = 0;
  FBIntf_Version = '1.0.0';
```

3.5 Reference

```
  IFirebirdAPI = interface
    {Database connections}
    function AllocateDPB: IDPB;
    function OpenDatabase(DatabaseName: AnsiString; DPB: IDPB;
         RaiseExceptionOnConnectError: boolean=true): IAttachment;
    function CreateDatabase(DatabaseName: AnsiString; DPB: IDPB;
         RaiseExceptionOnError: boolean=true): IAttachment; overload;
    function CreateDatabase(sql: AnsiString; aSQLDialect: integer;
         RaiseExceptionOnError: boolean=true): IAttachment; overload;

    {Start Transaction against multiple databases}
    function AllocateTPB: ITPB;
    function StartTransaction(Attachments: array of IAttachment;
             TPB: array of byte; DefaultCompletion:
                     TTransactionAction): ITransaction; overload;
    function StartTransaction(Attachments: array of IAttachment;
             TPB: ITPB; DefaultCompletion: TtransactionAction):
                                          ITransaction; overload;

    {Service Manager}
    function HasServiceAPI: boolean;
    function AllocateSPB: ISPB;
    function GetServiceManager(ServerName: AnsiString; Protocol: TProtocol;
                                              SPB: ISPB): IServiceManager;

    {Information}
    function GetStatus: IStatus;
    function GetLibraryName: string;
    function IsEmbeddedServer: boolean;
    function HasRollbackRetaining: boolean;
    function GetImplementationVersion: AnsiString;

    {Firebird 3 API}
    function HasMasterIntf: boolean;
    function GetIMaster: TObject;

    {utility}
    function GetCharsetName(CharSetID: integer): AnsiString;
    function CharSetID2CodePage(CharSetID: integer;
                        var CodePage: TSystemCodePage): boolean;
    function CodePage2CharSetID(CodePage: TSystemCodePage;
                               var CharSetID: integer): boolean;
    function CharSetName2CharSetID(CharSetName: AnsiString;
                               var CharSetID: integer): boolean;
    function CharSetWidth(CharSetID: integer;
                                   var Width: integer): boolean;
end;
```

Method	Use
AllocateDPB	Allocates an empty Database Parameter Block (DPB) (see 4.1).
OpenDatabase	Attach to an existing Database (see 4.3)
CreateDatabase	Create a new Database (see 4.2).
AllocateTPB	Allocate a Transaction Parameter Block (TPB) (see 5.1)
StartTransaction	Start a new transaction (see 5.3).
HasServiceAPI	Query whether the Service API is supported by the Firebird Client API.
AllocateSPB	Allocate a Service Parameter Block (SPB) (see 12.1)
GetServiceManager	Attach to the Service Manager (see 12.2)
GetStatus	Returns the IStatus interface (see 10.2).
GetLibraryName	Returns the filename (without the path) of the file containing the loaded Firebird Client Library.
IsEmbeddedServer	Returns True if the Firebird Client library also provides an embedded server.
HasRollbackRetaining	True if the Firebird Client API supports RollbackRetaining
GetImplementationVersion	Returns '2.5' for the legacy API, or '3.x' for the Firebird 3 API, where 'x' is replaced by the API version returned by the Client Library.
HasMasterIntf	Returns true if the Firebird 3 IMaster “interface” is available.
GetIMaster	If the Firebird 3 IMaster “interface” is available this returns a reference to the IMaster “interface”. Note this is typed as a TObject in order to avoid having to make the IB unit dependent on the Firebird 3 API, and must be cast to IMaster before use.

Method	Use
GetCharsetName	Lookup the name of the Character set that corresponds to a Firebird Character Set ID.
CharSetID2CodePage	Lookup the Code Page that corresponds to a Firebird Character Set ID.
CodePage2CharSetID	Lookup the Firebird Character Set ID that corresponds to a Code Page.
CharSetName2CharSetID	Lookup the Firebird Character Set ID that corresponds to a Character Set Name.
CharSetWidth	Lookup the Character Set width that corresponds to a Firebird Character Set ID.

4

Working with Databases

All Database Connections are managed using the **IAttachment** interface. This interface is returned by a call to the IFirebirdAPI.OpenDatabase or the IFirebirdAPI.CreateDatabase methods. In each case, a Database Parameter Block (DPB) must be provided as a parameter to the call.

4.1 The Database Parameter Block (DPB)

The DPB is used to pass various parameters to an OpenDatabase or CreateDatabase method. These include the User Name and Password, the default Character Set and the SQL Dialect.

Building a DPB is simple enough. The IFirebirdAPI.AllocateDPB method is used to allocate an interface to an empty DPB (IDPB[2]) and this interface's **Add** method is used to add parameters to the block:

```
IDPB = interface
    function getCount: integer;
    function Add(ParamType: byte): IDPBItem;
    function getItems(index: integer): IDPBItem;
    function Find(ParamType: byte): IDPBItem;
    property Count: integer read getCount;
    property Items[index: integer]: IDPBItem read getItems; default;
end;
```

Note that once a parameter has been added to the parameter block, an interface to it (IDPBItem) is returned. This interface can be accessed later using the **find** method. It is also possible to enumerate the existing parameters using the getCount method and Items property. For example,

[2]See also Appendix A.

```
var MyDPB: IDPB;
begin
  MyDPB := FirebirdAPI.AllocateDPB;
  MyDPB.Add(isc_dpb_user_name).AsString := 'SYSDBA';
  MyDPB.Add(isc_dpb_password).AsString := 'masterkey';
  MyDPB.Add(isc_dpb_lc_ctype).AsString := 'UTF8';
  MyDPB.Add(isc_dpb_set_db_SQL_dialect).AsByte := 3;
```

is a typical example of the use of IDPB to populate a DPB prior to attaching to the database.

Note that the parameter to the **Add** method is one of the DPB symbolic constants defined by the Firebird API. The data type is dependent on the parameter.

The IDPBItem interface is defined as:

```
IDPBItem = interface(IParameterBlockItem) end;
```

It is defined by subclassing the IParameterBlockItem interface (see) Getter and setter methods are defined for string, integer and byte parameters, together with corresponding properties. The parameter type (e.g. isc_dpb_user_name) can be queried using the **getParamType** method. The following provides an example of enumerating a DPB to print out each parameter's value:

```
procedure TTestBase.PrintDPB(MyDPB: IDPB);
var i: integer;
begin
  writeln('DPB');
  writeln('Count = ', MyDPB.Count);
  for i := 0 to MyDPB.Count - 1 do
    writeln(MyDPB[i].getParamType,' = ', MyDPB[i].AsString);
  writeln;
end;
```

Note that all parameter types can be returned as a string value.

4.1.1 Reference

The following symbolic constants may be used in a DPB:

Constant	Type	Definition
isc_dpb_user_name	String	Login User Name
isc_dpb_password	String	Login Password
isc_dpb_lc_ctype	String	Default Character Set Name
isc_dpb_sql_role_name	String	Login Role name
isc_dpb_sql_dialect	Byte	Default SQL Dialect (1 or 3)
isc_dpb_page_size	Integer	Database Page Size (create database only)

Other symbolic constants are available for special use (e.g. gfix type operations). See the InterBase 6 API Guide for more information.

4.2 Creating a New Database

A new database is created using the IFirebirdAPI.CreateDatabase method. On a successful completion, this creates a database and returns an **IAttachment** Interface providing access to the connection to the newly created database. This function comes in two variants. The first is similar to OpenDatabase (see below) and uses a DPB to provide the database parameters.

```
function CreateDatabase(DatabaseName: AnsiString; DPB: IDPB;
                        RaiseExceptionOnError: boolean=true): IAttachment;
```

In this case, the DPB provides the login credentials for the user that is to become the database owner. The connection Default Character set becomes that defined for the database. The database page size DPB parameter (isc_dpb_page_size) is also recognised and used when creating the database.

The DatabaseName is either a path to the local database filename, or a connect string in the form:

serverName:aliasOrPath

where “serverName” is the domain name for the Firebird Server (localhost is permitted) and the “aliasOrPath” is either a valid database alias name defined in the server's “aliases.conf” file or the full path to the database file on the server.

If RaiseExceptionOnError is false then any errors are silently ignored and a nil interface reference is returned. Otherwise, if an error occurs then an exception is raised.

The second variant provides a means to create a database using the “Create Database” SQL statement. This is:

```
function CreateDatabase(sql: AnsiString; aSQLDialect: integer;
             RaiseExceptionOnError: boolean=true): IAttachment; overload;
```

In this case, the sql parameter must provide a “Create Database” SQL statement as described in the Firebird Language Guide.

4.3 Attaching to an Existing Database

You can attached to existing database using the IFirebirdAPI.OpenDatabase method. On successful completion, this opens a connection to the database and returns an **IAttachment** Interface providing access to it.

```
function OpenDatabase(DatabaseName: AnsiString; DPB: IDPB;
                RaiseExceptionOnConnectError: boolean=true): IAttachment;
```

In this case, the DPB provides the login credentials for the user that is logging into the database. The user must have the necessary access rights for access to the database.

The DatabaseName is as above and must identify an existing database.

If RaiseExceptionOnError is false then any errors are silently ignored and a nil interface reference is returned. Otherwise, if an error occurs then an exception is raised.

4.4 Controlling access to the DPB Password

The password added to a DPB is kept in memory and in clear. It can be accessed after a database has been opened. If the IAttachment interface is passed to an untrusted user then this could be a problem. To avoid this potential security hazard, the password should be invalidated after the connection is opened e.g.

```
var MyAttachment: IAttachment;
begin
  …
  MyAttachment := IFirebirdAPI.OpenDatabase 'path to database',MyDPB);
MyDPB.Find(isc_dpb_password).AsString := 'xxxxxxxx';
```

4.5 Disconnecting

An IAttachment interface is returned for an active connection. This connection can be terminated at any time by calling the IAttachment.Disconnect method. This terminates the connection but does not invalidate the interface which can still be used to reconnect to the database.

Prior to a database being disconnected, all active transactions are closed using their default completion.

4.6 Reconnecting

After a database connection has been terminated, the IAttachment.Connect method may be used to reconnect the attachment to the same database. On successful completion, the connection has been restored.

Note that if the password has been invalidated as discussed above in 4.4, then the connect will fail under the password is restored e.g.

```
MyAttachment.getDPB.Find(isc_dpb_password).AsString := 'masterkey';
MyAttachment.Connect;
```

4.7 Dropping a Database

The IAttachment.DropDatabase method can be used to drop an existing database, if the logged in user has sufficient privilege to drop the database. After this method is called, the database file on the server is removed, the connection is disconnected and any further calls to this attachment interface instance are undefined. For example:

```
MyAttachment.DropDatabase;
MyAttachment := nil; {ensure no further use}
```

Prior to a database being dropped, all active transactions are closed using their default completion.

4.8 Getting Database Information

The IAttachment interface also provides access various database statistics and other information using the IAttachment.GetDBInformation method. This method takes a list of request items as its parameter and returns an IDBInformation interface providing access to the requested information.

Information is requested using one of the following DB Information constants. Either a single item is requested, or a set of information items is requested:

isc_info_db_id	Database File Name and site name
isc_info_allocation	Number of database pages allocated
isc_info_base_level	Database Version (level) number
isc_info_implementation	Database Implementation Number
isc_info_no_reserve	Is space reserved for backup records
isc_info_ods_minor_version	ODS minor version number
isc_info_ods_version	ODS version number
isc_info_page_size	Number of bytes per page
isc_info_version	Database implementation version no.
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	List of logged in users.
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
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 that are committed, so that older ancestor versions are no longer needed)
isc_info_read_idx_count	Number of reads done via an index since the dataase was last attached
isc_info_read_seq_count	Number of sequential 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
isc_info_db_SQL_Dialect	Get Database SQL Dialect

The InterBase 6.0 API Guide provides more information on each of the above.

IDBInformation is a simple interface providing access to the buffer containing the information requested:

```
IDBInformation = interface
    function GetCount: integer;
    function GetItem(index: integer): IDBInfoItem;
    function Find(ItemType: byte): IDBInfoItem;
    property Count: integer read GetCount;
    property Items[index: integer]: IDBInfoItem read getItem; default;
  end;
```

This interface can be used to enumerate the individual information items requested. Each item is returns as an IDBInfoItem:

```
IDBInfoItem = interface
  function getItemType: byte;
  function getSize: integer;
  procedure getRawBytes(var Buffer);
  function getAsString: AnsiString;
  function getAsInteger: integer;
  procedure DecodeIDCluster(var ConnectionType: integer;
                            var DBFileName, DBSiteName: AnsiString);
  function getAsBytes: TByteArray;
  procedure DecodeVersionString(var Version: byte; var VersionString: AnsiString);
  function getOperationCounts: TDBOperationCounts;
  procedure DecodeUserNames(UserNames: TStrings);

  {user names only}
  function GetCount: integer;
  function GetItem(index: integer): IDBInfoItem;
  function Find(ItemType: byte): IDBInfoItem;
  property AsInteger: integer read getAsInteger;
  property AsString: AnsiString read GetAsString;
  property Count: integer read GetCount;
  property Items[index: integer]: IDBInfoItem read getItem; default;
end;
```

Each DB Information item can be a single value or a set of values that can itself be enumerated. Getter methods are provided for each data type that can be returned. Including the following special cases:

- DecodeIDCluster is used to decode information returned for information type isc_info_db_id.
- DecodeVersionString is used to decode information returned for isc_info_base_level
- getOperationCounts is used for returned operation counts (isc_info_backout_count onwrds)
- DecodeUserNames may be used for isc_info_user_names.
- GetAsBytes is used for isc_info_base_level and isc_info_implementation.

4.9 Database Activity Monitor

A simple means of polling for database API activity is provided by the IAttachment.HasActivity method. This returns true if any activity has taken place over this connection since the last time the method was called, and false otherwise.

This may be used to automatically disconnect idle connections after some period has elapsed.

4.10 Attaching to a Database using the Embedded Server

When running on a Unix platform*, fbint* sets up the local environment to avoid file permissions issues with the Firebird lock and temporary directories. That is, it will create on initialisation, in the default temporary file directory (typically /tmp under Linux), a directory called “Firebird_<username>, where <username> is the current login user name and, unless they are already defined, set the FIREBIRD_TMP and FIREBIRD_LOCK environment variables to point to this directory.

If your database path consists only of a path to a file on your local system and the embedded server is available then Firebird will attempt to attach to the database without connecting to the server. For this to be successful:

- Under Linux, the user name and password must not be present in the DPB.
- Under Windows, a user name and password should be present in the DPB. However, these should be set to the default of “SYSDBA “and “masterkey” respectively.

4.11 Reference

```
IAttachment = interface
    function getDPB: IDPB;
    function AllocateBPB: IBPB;
    procedure Connect;
    procedure Disconnect(Force: boolean=false);
    function IsConnected: boolean;
    procedure DropDatabase;
    function StartTransaction(TPB: array of byte;
        DefaultCompletion: TTransactionAction=taCommit): ITransaction; overload;
    function StartTransaction(TPB: ITPB;
        DefaultCompletion: TTransactionAction=taCommit): ITransaction; overload;
    procedure ExecImmediate(transaction: ITransaction; sql: AnsiString;
                                        SQLDialect: integer); overload;
    procedure ExecImmediate(TPB: array of byte; sql: AnsiString;
                                        SQLDialect: integer); overload;
    procedure ExecImmediate(transaction: ITransaction; sql: AnsiString); overload;
    procedure ExecImmediate(TPB: array of byte; sql: AnsiString); overload;
    function ExecuteSQL(TPB: array of byte; sql: AnsiString; SQLDialect: integer;
                                     params: array of const): IResults; overload;
    function ExecuteSQL(transaction: ITransaction; sql: AnsiString;
               SQLDialect: integer; params: array of const): IResults; overload;
    function ExecuteSQL(TPB: array of byte; sql: AnsiString;
                                     params: array of const): IResults; overload;
    function ExecuteSQL(transaction: ITransaction; sql: AnsiString;
                                     params: array of const): IResults; overload;
    function OpenCursor(transaction: ITransaction; sql: AnsiString;
                                      aSQLDialect: integer): IResultSet; overload;
    function OpenCursor(transaction: ITransaction; sql: AnsiString;
                              aSQLDialect: integer;
                              params: array of const): IResultSet; overload;
    function OpenCursor(transaction: ITransaction;
                                         sql: AnsiString): IResultSet; overload;
    function OpenCursor(transaction: ITransaction; sql: AnsiString;
                              params: array of const): IResultSet; overload;
    function OpenCursorAtStart(transaction: ITransaction; sql: AnsiString;
                                 aSQLDialect: integer): IResultSet; overload;
    function OpenCursorAtStart(transaction: ITransaction; sql: AnsiString;
                              aSQLDialect: integer;
                              params: array of const): IResultSet; overload;
    function OpenCursorAtStart(transaction: ITransaction;
                                           sql: AnsiString): IResultSet; overload;
    function OpenCursorAtStart(transaction: ITransaction; sql: AnsiString;
                              params: array of const): IResultSet; overload;
    function OpenCursorAtStart(sql: AnsiString): IResultSet; overload;
    function OpenCursorAtStart(sql: AnsiString;
                              params: array of const): IResultSet; overload;
    function Prepare(transaction: ITransaction; sql: AnsiString;
                           aSQLDialect: integer): IStatement; overload;
    function Prepare(transaction: ITransaction;
                                      sql: AnsiString): IStatement; overload;
    function PrepareWithNamedParameters(transaction: ITransaction; sql: AnsiString;
                        aSQLDialect: integer;
                             GenerateParamNames: boolean=false): IStatement; overload;
    function PrepareWithNamedParameters(transaction: ITransaction; sql: AnsiString;
                        GenerateParamNames: boolean=false): IStatement; overload;

    {Events}
    function GetEventHandler(Events: TStrings): IEvents; overload;
    function GetEventHandler(Event: AnsiString): IEvents; overload;

    {Blob - may use to open existing Blobs. However, ISQLData.AsBlob is
      preferred}
```

```
    function CreateBlob(transaction: ITransaction; RelationName,
                        ColumnName: AnsiString; BPB: IBPB=nil): IBlob; overload;
    function CreateBlob(transaction: ITransaction;
              BlobMetaData: IBlobMetaData; BPB: IBPB=nil): IBlob; overload;
    function CreateBlob(transaction: ITransaction; SubType: integer;
                    CharSetID: cardinal=0; BPB: IBPB=nil): IBlob; overload;
    function OpenBlob(transaction: ITransaction; RelationName,
              ColumnName: AnsiString; BlobID: TISC_QUAD; BPB: IBPB=nil): IBlob;

    {Array - may use to open existing arrays. However, ISQLData.AsArray is
        preferred}

    function OpenArray(transaction: ITransaction; RelationName,
                           ColumnName: AnsiString; ArrayID: TISC_QUAD): IArray;
    function CreateArray(transaction: ITransaction; RelationName,
                                              ColumnName: AnsiString): IArray;
overload;
    function CreateArray(transaction: ITransaction;
                               ArrayMetaData: IArrayMetaData): IArray; overload;
    function CreateArrayMetaData(SQLType: cardinal; Scale: integer; size: cardinal;
                  charSetID: cardinal; dimensions: cardinal;
                  bounds: TArrayBounds): IArrayMetaData;

    {Database Information}
    function GetSQLDialect: integer;
    function GetBlobMetaData(Transaction: ITransaction; tableName,
                                         columnName: AnsiString): IBlobMetaData;
    function GetArrayMetaData(Transaction: ITransaction; tableName,
                                        columnName: AnsiString): IArrayMetaData;
    function GetDBInformation(Requests: array of byte)
                                                 : IDBInformation; overload;
    function GetDBInformation(Request: byte): IDBInformation; overload;
    function HasActivity: boolean;
  end;
```

Method	Use
getDPB	Returns a reference to the DPB used to connect to the database.
AllocateBPB	Allocates an empty Blob Parameter Block (BPB) (See 7.6).
Connect	Reconnect to the database following a Disconnect (see 4.6)
Disconnect	Disconnect from the database. If “force” is true then errors are ignored (see 4.5).
IsConnected	Returns true if database connection is active.
DropDatabase	Requests that the current database is closed and deleted from the server (see 4.7).
StartTransaction	Starts a new transaction on the database (see 5.2).

Method	Use
ExecImmediate	Execute an SQL Statement with no input or output. (see 6.2)
ExecuteSQL	Executes an non-Select SQL Statement with input parameters and optional output (see 6.4.1).
OpenCursor	Execute an SQL Query Statement and return the results set (see 6.7).
OpenCursorAtStart	Execute an SQL Query Statement and return the results set with the cursor positioned at the first row, if any (see 6.7).
Prepare	Prepare an SQL Statement using positional parameters (see 6.4 and 6.6)
PrepareWithNamedParameters	Prepare an SQL Statement using the named parameters syntax (see 6.4 and 6.6)
GetEventHandler	Returns an Event Handler interface for handling events on this database (see 11.1)
CreateBlob	Returns an interface to an empty Blob (see 7.4).
OpenBlob	Returns an interface to an existing blob (See 7.3).
CreateArray	Returns an interface to an empty Array (see 8.4)
CreateArrayMetaData	Creates an array metadata structure from the provided information. (see 8.1)
OpenArray	Returns an interface to an existing array (see 8.3).
GetSQLDialect	Returns the connection's default SQL Dialect
GetBlobMetaData	Returns the metadata for a Blob Column (See 7.1).
GetArrayMetaData	Returns the metadata for an Array Column (see 8.1).

Method	Use
GetDBInformation	Get Database Information (see 4.8).
HasActivity	Returns true if the database connection has been used since the last call to the method (see 4.9).

5

Working with Transactions

Firebird is a transaction orientated database with all SQL activity taking place within the context of a transaction. Transactions can be isolated from each other and used to determine when changes are committed (i.e made available to concurrent connections). It is also possible to rollback a transaction (i.e. to discard all changes made under the transaction).

A transaction can be started on a single transaction or, simultaneously on multiple databases in to co-ordinate updates across more than one database.

The ITransaction interface provides access to a Firebird transaction.

5.1 The Transaction Parameter Block (TPB)

The Transaction Parameter Block is used to pass various parameters to a StartTransaction method. These include transaction isolation requirements, action on record locks and access types.

Creating a TPB is simple enough: the IFirebirdAPI.AllocateTPB method is used to allocate an interface to an empty TPB (ITPB) and this interface's **Add** method is used to add parameters to the TPB.

```
ITPB = interface
    function getCount: integer;
    function Add(ParamType: byte): ITPBItem;
    function getItems(index: integer): ITPBItem;
    function Find(ParamType: byte): ITPBItem;
    property Count: integer read getCount;
    property Items[index: integer]: ITPBItem read getItems; default;
  end;
```

This interface follows the pattern established for the DPB (see), with the Add method used to add a new item, a Find method to locate an existing item and the means provided to enumerate a TPB.

The ITPBItem interface is defined as:

```
ITPBItem = interface(IParameterBlockItem) end;
```

The common transaction parameters do not have any values associated with the, and a typical example of allocating and populating a TPB is:

```
var MyTPB: ITPB;
begin
  MyTPB := IFirebird.AllocateTPB;
  MyTPB.Add(isc_tpb_write);
  MyTPB.Add(isc_tpb_nowait);
  MyTPB.Add(isc_tpb_concurrency);
```

Note that because few TPB parameters take values, the StartTransaction method discussed below has a variation that only requires a set of TPB constants rather than an ITPB. The TPB is then built automatically from the set of constants.

Common TPB constants are:

Constant	Interpretation
isc_tpb_read	Read Only Transaction
isc_tpb_write	Read/Write 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 Firebird multi-generational transaction model [Default]
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_committed	High throughput, high concurrency transaction that can read changes committed by other concurrent transactions. Use of this parameter takes full advantage of the Firebird multi-generational transaction model.
isc_tpb_lock_read	Locks the table given by the parameter value (string: name of table) for write but permits read by other transactions.
isc_tpb_lock_write	Locks the table given by the parameter value (string: name of table) for write but permits read by read committed and concurrency transations

For additional constants and a more detailed interpretation of the above, the reader should refer to the InterBase 6.0 API Guide.

5.2 Starting a Transaction

The IAttachment.StartTransaction method is used to start a transaction on a single database. It returns a reference to the ITransaction interface for the newly started transaction. Two variants of this method are available:

```
function StartTransaction(TPB: array of byte;
    DefaultCompletion: TTransactionCompletion=taCommit): ITransaction; overload;

function StartTransaction(TPB: ITPB;
    DefaultCompletion: TTransactionCompletion=taCommit): ITransaction; overload;
```

The first variant may be used when none of the required transaction parameters takes a value. In this case, the TPB is expressed as an array of symbolic constants. The second variant requires that a TPB is built by the caller and provided as a method parameter.

In both cases, the default transaction completion (TARollback, TACommit) may be provided (default is taCommit). This is interpreted such that if the interface goes out of scope (i.e. is automatically freed) before an explicit commit or rollback, then the transaction is completed using the specific default completion.

For example:

```
MyTransaction := MyAttachment.StartTransaction([isc_tpb_write,
                                isc_tpb_nowait,isc_tpb_concurrency],taCommit);
```

Note: Under Delphi, interfaces are disposed of when the containing block is exited while under FPC, an interface is disposed of as soon as it becomes inaccessible. For example, when the variable referencing the interface is set to “nil”. This difference can be significant if your program relies on default transaction commit/rollback as this may occur at different points in the execution sequence depending on whether you are using FPC or Delphi.

5.3 Starting a Transaction on Multiple Databases

The IFirebirdAPI.StartTranscation method is used to start a transaction on multiple databases. This also has two variants:

```
function StartTransaction(Attachments: array of IAttachment;
             TPB: array of byte;
             DefaultCompletion: TTransactionCompletion=taCommit):
                                                    ITransaction; overload;

function StartTransaction(Attachments: array of IAttachment;
             TPB: ITPB; DefaultCompletion: TTransactionCompletion=taCommit):
                                                   ITransaction; overload;
```

The difference between these variants and those for a single database are that, for a single database, the database is implicit in the IAttachment, while for the multiple database case, the databases have to be provided as an array.

Note that if the array contains only a single attachment, this is treated identically to the single database attachment variant.

5.4 Committing a Transaction

A transaction is committed using the ITransaction.Commit or ITransaction.CommitRetaining methods:

```
procedure Commit(Force: boolean=false);
procedure CommitRetaining;
```

In the first case, the transaction ceases to be active when the call completes while, in the second case, the transaction remains active and further actions may take place in the context of the same transaction.

If the “Force” parameter is true then the errors are silently ignored.

Prior to a transaction being committed, all active Statements using the transaction are closed.

5.5 Two Phase Commit

The two phase commit procedure is used when a transaction has performed updates across multiple databases. It is used to ensure that if a problem occurs during the commit process an administrator can nevertheless perform a deterministic error recovery process ensuring that the transaction is committed on all databases.

The ITransaction.PrepareForCommit method is used to initiate the two phase commit process. Once this returns, all databases are guaranteed to be in the same state and the commit method may now be called to commit the transaction across all databases. For example:

```
MyTransaction.PrepareForCommit;
MyTransaction.Commit;
```

5.6 Transaction Rollback

A transaction is rolled back using the ITransaction.Rollback or ITransaction.RollbackRetaining methods:

```
procedure Rollback(Force: boolean=false);
procedure RollbackRetaining;
```

The semantics are the same as for the commit variants except that the database state is rolled back to the point at which the transaction was started or the last commitRetaining.

ior to a transaction being rolled back, all active Statements using the transaction are closed.

5.7 Restarting a Transaction

After a transaction has been committed or rolled back, it is possible to restart the transaction using the ITransaction.Start method:

```
procedure Start(DefaultCompletion: TTransactionCompletion=taCommit);
```

This restarts the transaction with the same TPB.

Note that the default completion may be changed at this point.

5.8 Transaction Activity Monitor

A simple means of polling for transaction API activity is provided by the ITransaction.HasActivity method. This returns true if any activity has taken place using this transaction since the last time the method was called, and false otherwise. Activity includes any SQL statement operating in the transaction context.

This may be used to automatically complete idle transactions after some period has elapsed.

5.9 Reference

```
ITransaction = interface
    function getTPB: ITPB;
    procedure Start(DefaultCompletion: TTransactionCompletion=taCommit);
    function GetInTransaction: boolean;
    procedure PrepareForCommit; {Two phase commit - stage 1}
    procedure Commit(Force: boolean=false);
    procedure CommitRetaining;
    function HasActivity: boolean;
    procedure Rollback(Force: boolean=false);
    procedure RollbackRetaining;
    function GetAttachmentCount: integer;
    function GetAttachment(index: integer): IAttachment;
    property InTransaction: boolean read GetInTransaction;
  end;
```

Method	Use
getTPB	Returns a reference to the TPB used to start the transaction.
Start	Restart a transaction (see 5.7)
GetInTransaction	Returns true if the transaction is active
PrepareForCommit	Start of two phase commit process for multiple databases (see 5.5)
Commit	Commit and terminate the transaction (see 5.5)
CommitRetaining	Commit and leave the transaction active (see 5.5).
HasActivity	Returns true if transaction acitivity has taken place since the last call to the method (see 5.8)
Rollback	Rollback and terminate the transaction (see 5.6).
RollbackRetaining	Rollback and leave the transaction active (see 5.6).

Method	Use
GetAttachmentCount	Returns the number of database attachments over which the transaction is active.
GetAttachment	Return a selected database attachment.

6

Working with Dynamic SQL

Firebird is an SQL Database. Data held within the database is access and modified using SQL Data Manipulation Language (DML) statements and the database metadata (e.g. table definitions) managed using the SQL Data Definition Language (DDL).

The Firebird Client API uses the Dynamic SQL variant of the Firebird SQL implementation for all database queries and data modifications. Dynamic SQL is used for statements that are built and executed dynamically at run time rather than being compiled into a program.

This section describes how SQL Statements are used with the Firebird Pascal API.

6.1 Dynamic SQL and the Firebird Pascal API

The SQL Statement syntax is described fully in the Firebird Language Guide and this document should be consulted for all SQL reference. However, this API also provides an extended syntax for statement parameter definition.

It is also worth noting that there are two SQL dialects supported. Dialect 1 is a more limited dialect for legacy applications, while dialect 3 is the more up-to-date one recommended for all new applications. One of the more notable differences between the dialects is that dialect 3 supports SQL Identifiers that are reserved words or case sensitive by placing them within double quotes.

6.1.1 Named Parameters

Firebird Dynamic SQL only supports positional parameters in SQL statements. For example:

```
Select * from MyTable Where MyKeyName like ?
```

Where the question mark is a placeholder for a positional parameter. The parameters are accessed by a zero based index number in the order they occur in the statement.

The Firebird Pascal API extends this syntax to allow for named parameters using the same conventions used for the Firebird Procedure and Trigger Language, where named parameters are case insensitive SQL identifiers preceded by a colon. For example:

```
Select * from MyTable Where MyKeyName like :PARAM
```

In the above, “PARAM” is a named parameter.

An SQL Statement containing named parameters is parsed by the Firebird Pascal API before the statement is passed to the Firebird API and the named parameters replaced by placeholders (question marks), A lookup table is retained to provide a mapping between parameter names and their position. It is then possible for the API user to specify parameter values by name, with the Firebird Pascal API looking up the name and setting the corresponding positional parameter with the required value.

As a further extension, parameter names are not required to be unique. When a non-unique parameter name is set to a given value, all positional parameters linked to the same name are set to the required value. The API user can thus set more than one parameter value in a single operation.

Duplicate Parameter Names can be very useful. For example, an SQL Select Statement may be given as

```
Select Col1, Col2
From Table_A
Where Col3 = :arg1
UNION
Select Col4, Col5
From Table_B
Where Col6 = :arg1
```

In this case, "arg1" need only be set once. e.g.

```
SQLParams.ByName('arg1').AsInteger := 3;
```

Both cases will be set to 3.

6.1.2 Column Names

An SQL Statement that results in output data (e.g. a select statement) provides a results set that allows the data items (fields) in each output to be accessed by statement position or by (case insensitive) name. For example:

```
Select EMP_NO,FULL_NAME from EMPLOYEES;
```

In this case, the fields in the results set can be accessed positionally, with EMP_NO at position 0 and FULL_NAME at position 1, or by name using EMP_NO and FULL_NAME as the field names.

The fields in a results set should always have field names identical to the source Firebird table column name, or, if provided, a column alias name given in the SQL Statement. However, there are exceptions.

Firebird identifiers (e.g. column names) are typically case insensitive and are converted to upper case when processed and reported. This translates into the Firebird Pascal API always reporting upper case column names and matching column names to field names using a case insensitive match.

However, in SQL Dialect 3, Firebird introduced the ability to enclose identifiers in double quotes. This is necessary if, for example, you want a column name that is the same as an SQL Reserved word. It also allows you to have case sensitive column names, or column names containing spaces.

For Example:

```
Create Table MY_TABLE (
  "KeyField" Integer,
  "GRANT" VarChar(32),
  "My Column" Float
);
```

The Firebird Pascal API could readily handle case sensitive column names and isn't bothered by SQL reserved words. However, looking forward to using the Firebird Pascal API from IBX, there is the problem that the Lazarus TDataSet model includes the ability to automatically generate TField properties and which are then added to the Form's list of properties. The name of the generated property is formed by concatenating the IBX object name with the column alias name.

Pascal identifiers are also case insensitive and this could cause problems if two column names differ only in the case of their letters: the generated property names will cause a compilation error. Neither can Pascal identifiers contain spaces.

The Firebird Pascal API handles this by forcing all column names to upper case, regardless of how they are defined in SQL. It also replaces spaces by underscores. The identifiers given to Generated column properties are then both valid Pascal and unambiguous. However, it is still necessary to handle cases where two column names differ only in their case - forcing the column names to upper case will only result in a name clash.

It is also the case that column alias names aren't always unique anyway. For example, in the SQL:

```
select sum(col1), sum(col2) from MyTable;
```

Firebird will generate the alias name "SUM" for both cases. It will also allow you to specify the same alias name multiple times in the same statement.

The Firebird Pascal API handles this by checking for non-unique alias names when the SQL is prepared and disambiguating the column names by adding a numerical suffix (starting from one) to each non-unique column name it finds after the first one. The same approach is used when non-unique column names result after forcing the column name to upper case.

For example, with a table defined as

```
Create Table MY_TABLE (
 TableKey Integer not null,
 "My Field" VarChar(32),
 "MY Field" VarChar(32),
 Primary Key(TableKey)
);
```

The column names used by the Firebird Pascal API will be

```
TABLEKEY
MY_FIELD
MY_FIELD1
```

respectively.

```
IResults.ByName('tableKey').AsInteger
IResults.ByName('MY_FIELD').AsString
IResults.ByName('my_field1').AsString
```

Are then all valid examples for accessing the column values.

6.2 SQL Statement with no input or output

An SQL Statement with no input or output (e.g. a DDL statement) may be executed quickly and efficiently using the IAttachment.ExecImmediate method. Several variants of this method are available:

```
procedure ExecImmediate(transaction: ITransaction; sql: AnsiString;
                                        SQLDialect: integer); overload;
procedure ExecImmediate(TPB: array of byte; sql: AnsiString;
                                        SQLDialect: integer); overload;
procedure ExecImmediate(transaction: ITransaction; sql: AnsiString); overload;
procedure ExecImmediate(TPB: array of byte; sql: AnsiString); overload;
```

In each case an SQL Statement is provided as a plain text string. The variations allow for the transaction to be provided explicitly or defined as TPB (no value) parameters, and to enable the SQL Dialect to be explicitly provided. By default, the default connection SQL Dialect is used.

If the transaction is defined by TPB parameters then a transaction is constructed for the statement, the statement is executed and the transaction committed. When the transaction is given explicitly, it is the responsibility of the caller to commit the transaction.

For example:

```
const
    sqlCreateTable =
    'Create Table TestData ('+
    'RowID Integer not null,'+
    'Title VarChar(32) Character Set UTF8,'+
    'BlobData Blob sub_type 0, '+
    'Primary Key(RowID)'+
    ')';

begin
    Attachment.ExecImmediate([isc_tpb_write,
                              isc_tpb_wait,
                              isc_tpb_consistency],sqlCreateTable);
…
```

6.3 Metadata

Metadata provides information about data and a database's metadata includes the definition of data structures such as tables. When executing a DML SQL Statement with parameters, it is also useful to know the metadata that describes the statements input and/or output. In this case, the metadata tells the user information about each input parameter or column in the result set that includes:

- the SQL Type
- any names or other identification information
- refinements of the SQL Type, such as the character set used for strings, or the number decimal places in fixed point data.
- Whether the column or parameter can be set to null.

The IAttachment.Prepare method is the first step in executing an SQL Statement and, on successful completion, also provides the statement's metadata, via the IStatement interface:

```
function Prepare(transaction: ITransaction; sql: AnsiString;
                                aSQLDialect: integer): IStatement; overload;
function Prepare(transaction: ITransaction; sql: AnsiString): IStatement; overload;
function PrepareWithNamedParameters(transaction: ITransaction; sql: AnsiString;
                      aSQLDialect: integer; GenerateParamNames: boolean=false;
                      ): IStatement; overload;
function PrepareWithNamedParameters(transaction: ITransaction; sql: AnsiString;
                      GenerateParamNames: boolean=false;
                      ): IStatement; overload;
```

As shown above, four variants of the **prepare** method are available; all return an IStatement interface. The first two are used for statements that contain either no parameters or positional parameters only. The latter two are intended for statements that used named parameters. However, they may also be used for any SQL Statement – the **prepare** variant simply avoids the processing overhead of parsing the SQL in the client API.

The other variation is whether or not the SQL Dialect is given explicitly or defaults to the default connection SQL Dialect.

When statements are prepared with named parameters it is also possible to set GenerateParamNames to true. This is an IBX hangover and, in this case, if a positional placeholder (i.e. a ?) is found then it is linked to a named parameter in the format 'IBXParam*n*' where *n* is position number of the parameter.

6.3.1 Input Parameter Metadata

After the completion of the prepare step, the IStatement interface can be queried to determine the input parameter metadata, if any, using the IStatement.SQLParams property. This property returns an ISQLParams interface:

```
  ISQLParams = interface
    function getCount: integer;
    function getSQLParam(index: integer): ISQLParam;
    function ByName(Idx: AnsiString): ISQLParam ;
    function GetModified: Boolean;
    property Modified: Boolean read GetModified;
    property Params[index: integer]: ISQLParam read getSQLParam; default;
    property Count: integer read getCount;
  end;
```

The ISQLParams interface identifies how many input parameters were found (the **Count** property) and allows access to each one, either by position or by name (named parameter statements only). If there are no input parameters the **Count** property returns zero.

Each parameter is returned as an ISQLParam interface:

```
ISQLParam = interface
    function GetIndex: integer;
    function GetSQLType: cardinal;
    function GetSQLTypeName: AnsiString;
    function getSubtype: integer;
    function getName: AnsiString;
    function getScale: integer;
    function getCharSetID: cardinal;
    function getCodePage: TSystemCodePage;
```

```
    function getIsNullable: boolean;
    function GetSize: cardinal;
    function GetAsBoolean: boolean;
    function GetAsCurrency: Currency;
    function GetAsInt64: Int64;
    function GetAsDateTime: TDateTime;
    function GetAsDouble: Double;
    function GetAsFloat: Float;
    function GetAsLong: Long;
    function GetAsPointer: Pointer;
    function GetAsQuad: TISC_QUAD;
    function GetAsShort: short;
    function GetAsString: AnsiString;
    function GetIsNull: boolean;
    function GetAsVariant: Variant;
    function GetAsBlob: IBlob;
    function GetAsArray: IArray;
    procedure Clear;
    function GetModified: boolean;
    procedure SetAsBoolean(AValue: boolean);
    procedure SetAsCurrency(aValue: Currency);
    procedure SetAsInt64(aValue: Int64);
    procedure SetAsDate(aValue: TDateTime);
    procedure SetAsLong(aValue: Long);
    procedure SetAsTime(aValue: TDateTime);
    procedure SetAsDateTime(aValue: TDateTime);
    procedure SetAsDouble(aValue: Double);
    procedure SetAsFloat(aValue: Float);
    procedure SetAsPointer(aValue: Pointer);
    procedure SetAsShort(aValue: Short);
    procedure SetAsString(aValue: AnsiString);
    procedure SetAsVariant(aValue: Variant);
    procedure SetIsNull(aValue: Boolean);
    procedure SetAsBlob(aValue: IBlob);
    procedure SetAsArray(anArray: IArray);
    procedure SetAsQuad(aValue: TISC_QUAD);
    procedure SetCharSetID(aValue: cardinal);
    property AsDate: TDateTime read GetAsDateTime write SetAsDate;
    property AsBoolean:boolean read GetAsBoolean write SetAsBoolean;
    property AsTime: TDateTime read GetAsDateTime write SetAsTime;
    property AsDateTime: TDateTime read GetAsDateTime write SetAsDateTime;
    property AsDouble: Double read GetAsDouble write SetAsDouble;
    property AsFloat: Float read GetAsFloat write SetAsFloat;
    property AsCurrency: Currency read GetAsCurrency write SetAsCurrency;
    property AsInt64: Int64 read GetAsInt64 write SetAsInt64;
    property AsInteger: Integer read GetAsLong write SetAsLong;
    property AsLong: Long read GetAsLong write SetAsLong;
    property AsPointer: Pointer read GetAsPointer write SetAsPointer;
    property AsShort: Short read GetAsShort write SetAsShort;
    property AsString: AnsiString read GetAsString write SetAsString;
    property AsVariant: Variant read GetAsVariant write SetAsVariant;
    property AsBlob: IBlob read GetAsBlob write SetAsBlob;
    property AsArray: IArray read GetAsArray write SetAsArray;
    property AsQuad: TISC_QUAD read GetAsQuad write SetAsQuad;
    property Value: Variant read GetAsVariant write SetAsVariant;
    property IsNull: Boolean read GetIsNull write SetIsNull;
    property IsNullable: Boolean read GetIsNullable;
    property Modified: Boolean read getModified;
    property Name: AnsiString read GetName;
    property SQLType: cardinal read GetSQLType;
  end;
```

The ISQLParam interface is a large interface, as it provides both type information and setters and getters as well as corresponding properties for each data type supported. It provides both the input parameter metadata as well as a means of setting each parameter.

The caller may use the **SQLType** property to determine the actual data type of the parameter. The Firebird Data Definition Guide should be consulted for information on SQL Types and use of the **scale** property for fixed point types. Constants for the available SQL Types are defined in the “IB” unit.

The use of this interface is discussed below in 6.4, and is typically used to set the values of the input parameters. For example:

```
MyStatement := MyAttachment.Prepare(MyTransaction,
                                    'Select * From MyTable where rowid = ?');
if (MyStatement.SQLParams.Count > 0) and
             (MyStatement.SQLParams[0].SQLType = SQL_INTEGER) then
  MyStatement.SQLParams[0].AsInteger := 1;
```

In the above example, the number of input parameters is checked to ensure that at least one is available and the input parameter type is tested to ensure that it is an integer. However, in practice, this can usually be assumed *a priori* and this test omitted, given that the programmer has also specified the SQL Statement.

Note that you can set an input parameter to any data type which can be converted by the Firebird Engine to the input parameter data type. For example, a date can also be expressed as a date string.

6.3.2 Output Metadata

The output metadata describes the structure of the dataset that an SQL Statement returns after it is executed. It therefore consists only of information and provides no access to data. After the successful completion of an IStatement.Prepare statement, the IStatement.Metadata property gives access to the Output Metadata and returns an IMetadata interface:

```
IMetaData = interface
    function getCount: integer;
    function getColumnMetaData(index: integer): IColumnMetaData;
    function GetUniqueRelationName: AnsiString;
                                  {Non empty if all columns come from the same table}
    function ByName(Idx: AnsiString): IColumnMetaData;
    property ColMetaData[index: integer]: IColumnMetaData
                                             read getColumnMetaData; default;
    property Count: integer read getCount;
end;
```

This is very similar in structure to the ISQLParams interface and allows the number of output columns to be determined, and to access each such column either by position or name. In this case, a name is always available and is the unique column name (or alias if provided) given in the SQL Statement. For each column, the interface returns an IColumnMetaData interface:

```
IColumnMetaData = interface
    function GetIndex: integer;
    function GetSQLType: cardinal;
    function GetSQLTypeName: AnsiString;
    function getSubtype: integer;
    function getRelationName: AnsiString;
    function getOwnerName: AnsiString;
    function getSQLName: AnsiString;    {Name of the column in original table}
    function getAliasName: AnsiString;  {Alias Name of column or Column Name
                                      if no alias}
    function getName: AnsiString;       {Disambiguated uppercase Field Name – see
6.1.2}
    function getScale: integer;
    function getCharSetID: cardinal;
```

```
    function getCodePage: TSystemCodePage;
    function getIsNullable: boolean;
    function GetSize: cardinal;
    function GetArrayMetaData: IArrayMetaData; {Valid only for Array SQL Type}
    function GetBlobMetaData: IBlobMetaData; {Valid only for Blob SQL Type}
    property Name: AnsiString read GetName;
    property Size: cardinal read GetSize;
    property SQLType: cardinal read GetSQLType;
    property Scale: integer read getScale;
    property SQLSubtype: integer read getSubtype;
    property IsNullable: Boolean read GetIsNullable;
end;
```

The caller may use the **SQLType** property to determine the actual data type of each column. The Firebird Data Definition Guide should be consulted for information on SQL Types and use of the **scale** property for fixed point types. The handling of Blob types is discussed in chapter 7, and the handling of array types is discussed in chapter 8. Otherwise:

- The SQLTypeName is the textual representation of the SQL Type
- The subtype applies only to Blobs and distinguishes different Blob types.
- The Relation Name is the original table name from which the column is sourced.
- The Owner Name is the login user name of the table owner.
- The SQL Name is the column name used in the SQL Statement (may not be unique).
- The Alias Name is the alias given in the SQL Statement (must be unique)
- The Name property is the unique name of the column used in “ByName” lookups.
- The Character Set ID applies to text data and is the Firebird Character set id for the text.
- The Code Page is the system code page that corresponds to the Firebird Character Set.
- The Size property depends on the data type. For variable length strings, it is the maximum string length.

6.4 SQL Statements with input parameters only

Examples of SQL Statements that have input parameters but no output include Insert, Update and Delete Statements. In the general case. These statement must be prepared, as discussed above and then executed using the IStatement.Execute method (although a short cut does exist – see below):

```
function Execute(aTransaction: ITransaction=nil): IResults;
```

If the **aTransaction** parameter is omitted or set to nil, the same transaction that was used to prepare the statement is used to execute it. However, it is possible to use an (e.g. long lived) transaction to prepare a statement and then use a different (e.g. short lived) transaction to execute the statement, as long as the first transaction is still active. This approach allows a prepared statement to be executed multiple times (possibly with different parameters values), saving the data each time by committing the transaction whilst avoiding having to prepare the statement each time. That is because once a transaction is committed, a statement prepared using that transaction is no longer valid.

Note: an alternative approach using CommitRetaining achieves the same effect and avoids having to use separate transactions.

For example:

```
MyStatement := MyAttachment.Prepare(MyTransaction,
                                'Update MyTable Set MyText = ? where rowid = ?');
MyStatement.SQLParams[0].AsString := 'Some new text';
MyStatement.SQLParams[1].AsInteger := 1;
MyStatement.Execute;
MyTransaction.CommitRetaining;
```

The above example, prepares an update statement with positional parameters, then sets the value of those parameters and executes the statement. The update is saved to the database with CommitRetaining. The parameters can now be set to different values and the statement executed again, without having to re-prepare the statement.

Note that the IResults interface is returned by the execute method. However, this is ignored in the above example as there is no useful information returned. However, there are cases when useful information is returned and this is discussed in 6.5 below.

6.4.1 The IAttachment.ExecuteSQL method

The IAttachment ExecuteSQL method provides a short cut for the above which is often more appropriate than having separate steps to prepare, assign parameter and execute a statement. There are a set of ExecuteSQL methods available:

```
    function ExecuteSQL(TPB: array of byte; sql: AnsiString; SQLDialect: integer;
                                    params: array of const): IResults; overload;
    function ExecuteSQL(transaction: ITransaction; sql: AnsiString;
              SQLDialect: integer; params: array of const): IResults; overload;
    function ExecuteSQL(TPB: array of byte; sql: AnsiString;
                                    params: array of const): IResults; overload;
    function ExecuteSQL(transaction: ITransaction; sql: AnsiString;
                                    params: array of const): IResults; overload;
```

These vary by whether or not the connection default SQL Dialect is used, or whether an existing transaction is used or whether the statement is executed and commited in a single step with the transaction parameters provided as a list TPB constants.

An Execute SQL statement may have positional parameters and if so, the parameter values are provided as an array of const. For example:

```
Attachment.ExecuteSQL(Transaction, 'Execute Procedure DELETE_EMPLOYEE ?', [8]);
```

The ExecuteSQL statement can return a single row of results in the IResults interface. See below.

6.5 SQL Statements with Output

An SQL Statement with output is defined here as a non-select SQL Statement that returns a single row of data values. An example of such a statement is “Insert ….Returning”. In this case, the IResults interface returned by the IStatement.Execute or IAttachment.ExecuteSQL methods provides the returned data.

```
IResults = interface
   function getCount: integer;
   function GetTransaction: ITransaction;
   function ByName(Idx: AnsiString): ISQLData;
   function getSQLData(index: integer): ISQLData;
   procedure GetData(index: integer; var IsNull:boolean;
                                       var len: short; var data: PByte);
   procedure SetRetainInterfaces(aValue: boolean); {see 6.8}
   property Data[index: integer]: ISQLData read getSQLData; default;
   property Count: integer read getCount;
end;
```

This interface may be used to determine how many data items are returned (**Count** property) and allows each data item to be accessed either by position or by name, where the data item name is the output column name. It can also be used to get direct access to the raw data for each column by position (using the **GetData** method). This returns a null indicator, the length of the data and a pointer to the raw data returned from the database. When the data type is SQL_TEXT or SQL_VARYING, the pointer is always to the first character in the string (of length **len** bytes).

Each data item may also be accessed as a properly formatted type via an ISQLData interface:

```
ISQLData = interface(IColumnMetaData)
    function GetAsBoolean: boolean;
    function GetAsCurrency: Currency;
    function GetAsInt64: Int64;
    function GetAsDateTime: TDateTime;
    function GetAsDouble: Double;
    function GetAsFloat: Float;
    function GetAsLong: Long;
    function GetAsPointer: Pointer;
    function GetAsQuad: TISC_QUAD;
    function GetAsShort: short;
    function GetAsString: AnsiString;
    function GetIsNull: Boolean;
    function GetAsVariant: Variant;
    function GetAsBlob: IBlob; overload;
    function GetAsBlob(BPB: IBPB): IBlob; overload;
    function GetAsArray: IArray;
    property AsDate: TDateTime read GetAsDateTime;
    property AsBoolean:boolean read GetAsBoolean;
    property AsTime: TDateTime read GetAsDateTime;
    property AsDateTime: TDateTime read GetAsDateTime ;
    property AsDouble: Double read GetAsDouble;
    property AsFloat: Float read GetAsFloat;
    property AsCurrency: Currency read GetAsCurrency;
    property AsInt64: Int64 read GetAsInt64 ;
    property AsInteger: Integer read GetAsLong;
    property AsLong: Long read GetAsLong;
    property AsPointer: Pointer read GetAsPointer;
    property AsQuad: TISC_QUAD read GetAsQuad;
    property AsShort: short read GetAsShort;
    property AsString: AnsiString read GetAsString;
    property AsVariant: Variant read GetAsVariant ;
    property AsBlob: IBlob read GetAsBlob;
    property AsArray: IArray read GetAsArray;
    property IsNull: Boolean read GetIsNull;
    property Value: Variant read GetAsVariant;
end;
```

ISQLData is primarily a set of getters for each data type. Type conversion is performed where possible. For example, all scaler types and dates can be returned as strings. AsDouble and AsCurrency automatically adjust fixed point types to reflect the “scale” specified in the metadata.

Note that the ISQLData interface inherits from the IColumnMetaData interface. It thus also provides direct access to the data item's metadata. An ISQLData interface is simply an IColumnMetaData interface plus getter methods for each data type supported and corresponding properties.

For example:

```
var theResults: IResults;

begin
  MyStatement := MyAttachment.PrepareWithNamedParameters(MyTransaction,
          'Insert into MyTable (MyText, RowID) Values (:INITIALTEXT,:ROWID) '+
            'Returning RowID';
  MyStatement.SQLParams.ByName('INITIALTEXT').AsString := 'Some text';
  MyStatement.SQLParams.ByName('ROWID').AsInteger := 1;
  theResults := MyStatement.Execute;
  writeln('Insert completed with Rowid = ',theResults[0].AsInteger);
```

6.6 Query Statements

A query statement is an SQL Select Statement. It may or may not have input parameters and can return zero, one or more rows of data. A query statement's metadata describes each column in the results set.

A query statement is prepared and has its input parameters, if any, set in the same way as any other SQL Statement. The difference comes when it is executed.

Note: A query statement can be distinguished from other SQL Statements, by checking the IStatement.SQLStatementType property. A query statement has an SQL Statement Type of SQLSelect.

A query statement is executed using the IStatement.OpenCursor the method:

```
function OpenCursor(aTransaction: ITransaction=nil): IResultSet;
```

This is very similar to the execute statement, except that it returns an IResultSet interface instead of an IResults interface:

```
IResultSet = interface(IResults)
    function FetchNext: boolean;
    function GetCursorName: AnsiString;
    function IsEof: boolean;
    procedure Close;
end;
```

Note that this interface inherits from IResults and hence allows access to each column's data item in the same way as an IResults interface. It extends IResults to provide a means of scrolling through the result set.

Note: with Firebird 3, the cursor name is always empty.

When the results set is first returned, it is not focused on any row of the dataset and any attempt to access any inherited IResults methods or properties will result in an error. The **FetchNext** method must first be called to advance the cursor to the first row. This will return false if there are no more rows in the dataset.

Note that the **IsEof** method always returns false until FetchNext is called for the first time, even when the dataset is empty.

The interface design is intended to facilitate the following processing:, using a while loop

```
var theResults: IResultSet;

begin
  MyStatement := MyAttachment.Prepare(MyTransaction,
                     'Select * From MyTable Where MyText like ?');
  MyStatement.SQLParams[0].AsString := '%text%';
  theResults := MyStatement.OpenCursor;
  while theResults.FetchNext do
    writeln('Row ',theResults.ByName('ROWID').AsInteger,
            ' has text ',theResults[1].AsString);
  theResults.Close;
```

Note that the IResultSet should be closed after use by calling the **Close** method. However, this is not essential as the result set is automatically closed when the interface goes out of scope.

If the empty dataset case needs to be handled separately, then the following may be used:

```
var theResults: IResultSet;

begin
  MyStatement := MyAttachment.Prepare(MyTransaction,
                     'Select * From MyTable Where MyText like ?');
  MyStatement.SQLParams[0].AsString := '%text%';
  theResults := MyStatement.OpenCursor;
  if not theResults.FetchNext then
   writeln('The Dataset was empty!')
  else
  repeat
    writeln('Row ',theResults.ByName('ROWID').AsInteger,
            ' has text ',theResults[1].AsString);
  until not theResults.FetchNext;
  theResults.Close;
```

6.7 Simplified Queries

While some queries do return large and complex datasets, others return much simpler information sets and sometimes only a single data item. For this latter case, *fbintf* offers convenience functions that avoid the user having to proceed through all the above steps in order to, for example, count all the rows in a table.

```
    function OpenCursor(transaction: ITransaction; sql: AnsiString;
                                     aSQLDialect: integer): IResultSet; overload;
    function OpenCursor(transaction: ITransaction; sql: AnsiString;
                             aSQLDialect: integer;
                             params: array of const): IResultSet; overload;
    function OpenCursor(transaction: ITransaction;
                                        sql: AnsiString): IResultSet; overload;
    function OpenCursor(transaction: ITransaction; sql: AnsiString;
                             params: array of const): IResultSet; overload;
    function OpenCursorAtStart(transaction: ITransaction; sql: AnsiString;
                                aSQLDialect: integer): IResultSet; overload;
    function OpenCursorAtStart(transaction: ITransaction; sql: AnsiString;
                             aSQLDialect: integer;
                             params: array of const): IResultSet; overload;
    function OpenCursorAtStart(transaction: ITransaction;
                                          sql: AnsiString): IResultSet; overload;
    function OpenCursorAtStart(transaction: ITransaction; sql: AnsiString;
                             params: array of const): IResultSet; overload;
    function OpenCursorAtStart(sql: AnsiString): IResultSet; overload;
    function OpenCursorAtStart(sql: AnsiString;
                             params: array of const): IResultSet; overload;
```

The above set of methods are all provided by the IAttachment interface and are intended to be used in cases where there are no input parameters and the result set is relatively simple. There are two main variants:

- The OpenCursor methods prepare and execute the SQL statement with the provided transaction and return the result set.
- The OpenCursorAtStart group do the same, but additionally call **fetchNext** on the results set before returning.

These also vary by whether the default SQL Dialect is used or if the transaction is given as a parameter or a list of TPB attributes, or simply a default transaction. When a default transaction is used, the method creates its own transaction with parameters: isc_tpb_read, isc_tpb_wait and isc_tpb_concurrency and returns the result. For example, to get a count of all the rows in a table, the following expression can be used:

```
int rowCount;
begin
  rowCount := MyAttachment.OpenCursorAtStart(
                                  'Select count(*) from MyTable')[0].AsInteger;
```

This works because on return, the result set has been advanced to the first and only row, and the first and only data item in that row is an integer value i.e. the row count.

Positional parameters in queries are supported. If present, then an array of const “params” must be provided with one parameter value for each positional parameter and in positional order. The parameter values must be type compatible with each parameter .e.g.

```
var employees: integer;
begin
  employees := Attachment.OpenCursorAtStart(
        'Select count(*) As Counter from EMPLOYEE Where EMP_NO < ?',[8])[0].AsInteger
```

6.8 Performance Optimisation

Behind each interface is an object which has to be created and which will be automatically destroyed when the interface reference goes out of scope. While this generally works well, this can result in a significant overhead when processing a large dataset.

Both the IResults (and IResultsSet) and the IStatement interface have a **SetRetainInterfaces** method that allows an internal flag to be set indicating whether or not subordinate interfaces are to be retained rather than automatically destroyed when they go out of scope:

- If the IResults.SetRetainInterfaces flag is set to true then all subsequent ISQLData interfaces returned by the interface are retained. This avoids the overhead of constantly creating and discarding the interfaces when processing a large dataset.
- If the IStatement.SetRetainInterfaces flag is set to true then all IColumnMetaData and ISQLParam interfaces returned by the interface are retained. This avoids the overhead of constantly creating and discarding the interfaces if they are regularly referenced when processing a large dataset.

Setting either flag to false releases all held interfaces (that are already out of scope) and the interface no longer retains the subordinate interfaces.

Note: If this optimisation is used then it is important that the SetRetainInterfaces flag is explicitly set to false when the interface is no longer required. Otherwise the retained interfaces will never be released, even when the parent interface goes out of scope and a memory leak will result.

6.9 Performance Statistics

The IStatement interface also allows access to performance statistics. These are collected each time a statement is executed or a cursor opened.

Statistics collection is disabled by default. To enable statistics collection for a statement, call the EnableStatistics procedure setting the parameter to true.

Once enabled, the performance statistics for the mostly recently executed statement or cursor opened (by this IStatement interface) can be obtained by a call to GetPerfStatistics. For example, to report statistics in an ISQL fashion:

```
var stats: TPerfCounters;
begin
  …
  if Statement.GetPerfStatistics(stats) then
  begin
    writeln('Current memory = ', stats[psCurrentMemory]);
    writeln('Delta memory = ', stats[psDeltaMemory]);
    writeln('Max memory = ', stats[psMaxMemory]);
    writeln('Elapsed time= ', FormatFloat('#0.000',stats[psRealTime]/1000),' sec');
    writeln('Cpu = ', FormatFloat('#0.000',stats[psUserTime]/1000),' sec');
    writeln('Buffers = ', stats[psBuffers]);
    writeln('Reads = ', stats[psReads]);
    writeln('Writes = ', stats[psWrites]);
    writeln('Fetches = ', stats[psFetches]);
  end;
end;
```

TPerfCounters is defined as an array:

```
TPerfStats = (psCurrentMemory, psMaxMemory,
                psRealTime, psUserTime, psBuffers,
                psReads, psWrites, psFetches,psDeltaMemory);

  TPerfCounters = array[TPerfStats] of Int64;
```

Where the counter indexes reference the following counters:

psCurrentMemory	Current server memory used in bytes
psMaxMemory	Max server memory used in bytes
psRealTime	Local Query Execution elapsed time in milliseconds
psUserTime	Local CPU time for execution in milliseconds
psBuffers	Buffers in use after query execution
psReads	Number of database reads during execution.

psWrites	Number of database writes during execution
psFetches	Number of database fetches during execution
psDeltaMemory	Different in server memory used before and after query execution.

6.10 Reference

```
IStatement = interface
    function GetMetaData: IMetaData;  {Output Metadata}
    function GetSQLParams: ISQLParams;{Statement Parameters}
    function GetPlan: AnsiString;
    function GetRowsAffected(var SelectCount, InsertCount,
                             UpdateCount, DeleteCount: integer): boolean;
    function GetSQLStatementType: TIBSQLStatementTypes;
    function GetSQLText: AnsiString;
    function GetSQLDialect: integer;
    function IsPrepared: boolean;
    procedure Prepare(aTransaction: ITransaction=nil);
    function Execute(aTransaction: ITransaction=nil): IResults;
    function OpenCursor(aTransaction: ITransaction=nil): IResultSet;
    function GetAttachment: IAttachment;
    function GetTransaction: ITransaction;
    procedure SetRetainInterfaces(aValue: boolean);
    procedure EnableStatistics(aValue: boolean);
    function GetPerfStatistics(var stats: TPerfCounters): boolean;
    property MetaData: IMetaData read GetMetaData;
    property SQLParams: ISQLParams read GetSQLParams;
    property SQLStatementType: TIBSQLStatementTypes read GetSQLStatementType;
end;
```

Method	Description
GetMetaData	Returns an interface to the query metadata.
GetSQLParams	Returns an interface to the query parameters
GetPlan	Returns the query plan
GetRowsAffected	Returns the number of rows affected by the last query execution, analysed by query type.
GetSQLStatementType	Returns the SQL Statement Type
GetSQLText	Returns the SQL Statement as plain text
IsPrepared	Returns true if the query is still in its prepared state
Prepare	Reprepare a query, optionally with a different transaction.

Execute	Execute a non-select query
OpenCursor	Executes a select query and returns the results set
GetAttachment	Returns a reference to the connection used by the statement
GetTransaction	Returns a reference to the transaction used to prepare the query
SetRetainInterfaces	See 6.8
EnableStatistics	See 6.9
GetPerfStatistics	See 6.9

7

Working with Blob Data

Binary Large Objects (Blobs) are containers for almost unlimited amounts of binary data held within a Firebird Database. In practice, Blobs are limited by the database architectural limits and available disk storage but, perhaps the most important point is that their individual size limit is not part of the metadata.

Blobs can be created and written outside of the handling of SQL Statements. Once created, a Blob is given a unique, database generated, identifier which can be stored in a database column and with a different value for each row. This identifier is composed of two four byte non-negative integers and is given the type name ISC_QUAD.

A Blob is normally accessed by reading a database row, retrieving the ISC_QUAD identifier and then using this to read the Blob itself. The Blob field is updated by creating a new one and assigning its ISC_QUAD identifier to the field[3] and committing the change to the database. Blobs do not have to be explicitly deleted as Blobs that are not referenced from any table or active transaction are automatically removed as part of database garbage collection.

Text mode Blobs can also be read and written to using the ISQLData.AsString and ISQLParam.AsString properties and without the need to use the IBlob interface. However, there are issues with the latter case (see 9.5).

7.1 Blob MetaData

7.1.1 Output Metadata

With reference to 6.3.2, if a database column in the output metadata (IColumnMetaData) has an SQLType of SQL_BLOB then the column is defined as a Blob.

The SQLSubType is valid for Blob columns and identifies the type of data stored in the Blob. A subtype of “1” is always text, while “0” is undefined data. Other values can have database specific

[3]The instance of a column value in each row is referred to here as a “field”.

interpretations. For text Blobs, the IColumnMetaData.getCharSetID method returns the Firebird Character Set ID for the text data[4].

The IColumnMetaData.GetBlobMetaData method may also be used to return additional metadata for the Blob accessed using the IBlobMetaData interface:

```
IBlobMetaData = interface
    function GetSubType: integer;
    function GetCharSetID: cardinal;
    function GetCodePage: TSystemCodePage;
    function GetSegmentSize: cardinal;
    function GetRelationName: AnsiString;
    function GetColumnName: AnsiString;
end;
```

Most of this information is already provided via the IColumnMetaData. Only the segment size is unique to this interface. This interface is inherited by the IBlob interface (see 7.2).

7.1.2 Input Metadata

The ISQLParam interface (see 6.3.1) will identify when the parameter type is Blob (SQLType = SQL_BLOB), and gives its sub type and character set id. However, it is not possible at this point to see the full Blob Metadata.

7.2 The IBlob Interface

A Blob is accessed, read and written, using the IBlob Interface:

```
IBlob = interface(IBlobMetaData)
    function GetBPB: IBPB;
    procedure Cancel;
    procedure Close;
    function GetBlobID: TISC_QUAD;
    function GetBlobMode: TFBBlobMode;
    function GetBlobSize: Int64;
    procedure GetInfo(var NumSegments: Int64; var MaxSegmentSize,
                      TotalSize: Int64; var BlobType: TBlobType);
    function Read(var Buffer; Count: Longint): Longint;
    function Write(const Buffer; Count: Longint): Longint;
    function LoadFromFile(Filename: AnsiString): IBlob;
    function LoadFromStream(S: TStream) : IBlob;
    function SaveToFile(Filename: AnsiString): IBlob;
    function SaveToStream(S: TStream): IBlob;
    function GetAsString: rawbytestring;
    procedure SetAsString(aValue: rawbytestring);
    function SetString(aValue: rawbytestring): IBlob;
    function GetAttachment: IAttachment;
    function GetTransaction: ITransaction;
    property AsString: rawbytestring read GetAsString write SetAsString;
end;
```

Note that the IBlob interface inherits from IBlobMetaData and hence Blob metadata is also available through the IBlob interface.

Blobs of all subtypes can be read and written to as strings. For anything other than subtype 1 (text), the string is simply raw data. For text Blobs, transliteration may take place when assigning to

[4]Note that this is not necessarily the character set used to store the Blob. If the database connection has a default character set defined then this will take precedence and the Blob text is returned using this character set, unless the Blob character set is “none” or “octets”.

the AsString property if the source string has a different code page to the one defined for the Blob in its metadata.

Blobs may be read from or written to files and TStream descendents.

7.2.1 IBlob Reference

Method	Description
GetBPB	Returns the BPB, if any, used to create/open the Blob
Cancel	Cancels the creation of a new blob. The IBlob interface should not be used after a call to this method and any further use is undefined.
Close	Completes the creation of a new blob. The Blob may not be written to after a call to Close.
GetBlobID	Returns the BlobID assigned to the Blob
GetBlobMode	Returns read or write (fbmRead,fbmWrite)
GetBlobSize	Returns the current size of the Blob in bytes as held within the database (undefined for Blobs in write mode).
GetInfo	Returns basic Blob information from the server (undefined for Blobs in write mode).
Read	Read the requested number of bytes from the Blob, starting at the current position. May return fewer bytes if less than the requested number remain.
Write	Append the buffer contents to the Blob.
LoadFromFile	Opens and copies (appends) all data from the file to the Blob (write mode only)
LoadFromStream	As above, but reads from a stream
SaveToFile	Creates the specified file and copies all data from the Blob to the file (read mode only).
SaveToStream	As above, but copies to a stream.
GetAsString	Returns a string containing all data from the Blob. If the Blob subtype is 1 (text), the string's code page will be set to match the character set of the Blob, otherwise the code page is CP_NONE.

Method	Description
	(read mode only).
SetAsString	Writes (appends) all data in the string to the Blob. If the Blob subtype is 1 (text), transliteration may take place if the string's code page is different from that required for the Blob. (write mode only).
GetAttachment	Returns the database IAttachment interface for the Blob
GetTransaction	Returns the transaction used to create/open the Blob.

7.3 Reading Blob Data

When a dataset's column has an SQL Type of SQL_BLOB and the field in the current row is non-null, then the ISQLData's AsBlob property (see 6.5) may be used to access the Blob using the IBlob interface. For example, assuming that MyTable has an integer RowID column and a Blob column named MyBlobColumn:

```
var theResults: IResultSet;
    theBlob: IBlob;

begin
  MyStatement := MyAttachment.Prepare(MyTransaction,
                      'Select MyBlobColumn,RowID From MyTable Where RowID < ?');
  MyStatement.SQLParams[0].AsInteger := 10;
  theResults := MyStatement.OpenCursor;
  while theResults.FetchNext do
  begin
    theBlob := theResults[0].AsBlob;
    theBlob.SaveToFile('someFileName' + theResults[1].AsString);
  end;
  theResults.Close;
```

This example iterates through the result set comprising all rows with a RowID less than 10 and writes out the Blob data to a file with a filename form from constant text plus the RowID value.

Note that an alternative method exists for accessing a Blob using the IAttachment.OpenBlob method.

```
function OpenBlob(transaction: ITransaction; RelationName,
              ColumnName: AnsiString; BlobID: TISC_QUAD; BPB: IBPB=nil): IBlob;
```

This method accesses a Blob and returns an IBlob interface to it provided that the Relation (Table) Name, Column Name and the BlobID is known. The BlobID is returned using the ISQLData.AsQuad property This method allows for Blob IDs to be read and stored for later use, opening the Blob only when required.

7.4 Creating or Modifying a Blob

As indicated above, the API user cannot check the ISQLParam to determine the correct character set id for text blobs when assigning a new blob to a field. Instead, an appropriate Blob has to be created using *a priori* knowledge. A Blob is created using the IAttachment.CreateBlob method:

```
function CreateBlob(transaction: ITransaction; RelationName,
                        ColumnName: AnsiString; BPB: IBPB=nil): IBlob; overload;
function CreateBlob(transaction: ITransaction;
              BlobMetaData: IBlobMetaData; BPB: IBPB=nil): IBlob; overload;
function CreateBlob(transaction: ITransaction; SubType: integer;
                    CharSetID: cardinal=0; BPB: IBPB=nil): IBlob; overload;
```

Three variations are defined which differ in the way that the Blob data type is specified. The first variant provides a relation (i.e. table) name and a column name in that table. Database metadata is then looked up and used to create a Blob of a type compatible with the column definition.

The second variant achieves the same but provides the Blob metadata directly as an IBlobMetaData interface. This variant is useful when a Select Statement has already been prepared for the table in which the Blob is to be assigned and hence the metadata is already available client side. The IBlobMetaData interface is obtain by calling the IColumnMetaData.GetBlobMetaData method.

The third variant defines the Blob directly by specifying the required subtype and, for subtype 1 (text), the character set id.

Note that the Blob Parameter Block (BPB) is only required when a Blob Filter is also specified (see 7.6).

Once a Blob has been created, data can be written to the Blob and the Blob identifier assigned to a field in a database table. For example:

```
var MyBlob: IBlob;
    MyStatement: IStatement;
begin
  MyBlob := MyAttachment.CreateBlob(MyTransaction,'MyTable','MyBlobColumn');
  MyBlob.LoadFromFile('path to source file');
  MyStatement := MyAttachment.Prepare(MyTransaction,
        'Update MyTable Set MyBlobColumn = ? Where RowID = ?');
  MyStatement.SQLParams[0].AsBlob := MyBlob;
  MyStatement.SQLParams[1].AsInteger := 1;
  MyStatement.Execute;
```

In the above example, a compatible Blob is created for the MyBlobColumn in MyTable, and its contents loaded from a file. An Update SQL Statement is then used to save the newly created blob in the database.

Note that it is important that the same transaction is used to both create the blob and to execute the update statement.

7.5 Removing a Blob

An existing Blob is simply removed by either replacing it with a new Blob in an update statement, or using an Update Statement to set the field to NULL.

7.6 Using Blob Filters

Blob Filters may be used to convert a Blob from one data type to another. As described in the InterBase 6.0 API Guide, there are both built-in and user defined Blob Filters. A Blob Filter is requested by providing a Blob Parameter Block (BPB) when the Blob is Opened or Created:

```
IBPB = interface
    function getCount: integer;
    function Add(ParamType: byte): IBPBItem;
    function getItems(index: integer): IBPBItem;
    function Find(ParamType: byte): IBPBItem;
    property Count: integer read getCount;
    property Items[index: integer]: IBPBItem read getItems; default;
end;
```

An empty BPB is returned using the IAttachment.AllocateBPB method and, as shown above, follows the same approach as the IDPB and ITPB interfaces (see). Each parameter in the BPB is access using the IBPBItem interface:

```
IBPBItem = interface (IParameterBlockItem) end;
```

Only four parameters are currently defined for the BPB. Their symbolic constants and use is described below:

Parameter	Type	Interpretation
isc_bpb_target_type	integer	The subtype identifier for the result of the conversion.
isc_bpb_target_interp	integer	When the target subtype is 1 (text), this identifies the target character set id.
isc_bpb_source_type	Integer	The subtype identifier for the source data
isc_bpb_source_interp	Integer	When the source subtype is 1 (text), this identifies the source character set id.

When a Blob Filter is defined for the CreateBlob method, the source sub type should be appropriate for the data written to the Blob. The target sub type should be compatible with the Blob column.

Blob Filters can also be used when a Blob is read from the database. In this case, it is not possible to use the ISQLData.AsBlob property to get the Blob interface as this provides no means to set a BPB. Instead the ISQLData.GetAsBlob method must be used:

```
function GetAsBlob(BPB: IBPB): IBlob;
```

This method also returns an IBlob interface, but with the requested Blob Filter used to read and convert the Blob Data.

8

Working with Array Data

Firebird also supports arrays, where an array column is defined as a multi-dimensional table of a single data type with well defined bounds on each dimension. Each row may contain a different array of values.

The implementation of arrays closely follows that of Blobs, such that an array can be understood as a structured Blob, where the structure is that of the array. A user could implement their own arrays using Blobs, or use the built in support.

8.1 Array Metadata

Array metadata is available for each column that has an SQLType of SQL_ARRAY. The array metadata interface is returned by the IColumnMetaData.GetArrayMetaData method and is:

```
 TArrayBound = record
    UpperBound: short;
    LowerBound: short;
  end;
TArrayBounds = array of TArrayBound;

IArrayMetaData = interface
    function GetSQLType: cardinal;
    function GetSQLTypeName: AnsiString;
    function GetScale: integer;
    function GetSize: cardinal;
    function GetCharSetID: cardinal;
    function GetTableName: AnsiString;
    function GetColumnName: AnsiString;
    function GetDimensions: integer;
    function GetBounds: TArrayBounds;
end;
```

Array metadata is arguably more useful than Blob metadata and provides the information that defines the array, including the SQL data type of each array element, the scale for fixed point data types and the character set id and size for text type. It also identifies the number of dimensions in the array and the bounds for each dimension.

It is also possible to create an IArrayMetadata from supplied parameters using IAttachment.CreateArrayMetaData.

8.2 The IArray Interface

An array of data values is accessed using the IArray interface.

```
IArray = interface(IArrayMetaData)
    function GetArrayID: TISC_QUAD;
    procedure Clear;
    function IsEmpty: boolean;
    procedure PreLoad;
    procedure CancelChanges;
    procedure SaveChanges;
    function GetMetaData: IArrayMetaData;
    function GetAsInteger(index: array of integer): integer;
    function GetAsBoolean(index: array of integer): boolean;
    function GetAsCurrency(index: array of integer): Currency;
    function GetAsInt64(index: array of integer): Int64;
    function GetAsDateTime(index: array of integer): TDateTime;
    function GetAsDouble(index: array of integer): Double;
    function GetAsFloat(index: array of integer): Float;
    function GetAsLong(index: array of integer): Long;
    function GetAsShort(index: array of integer): Short;
    function GetAsString(index: array of integer): AnsiString;
    function GetAsVariant(index: array of integer): Variant;
    procedure SetAsInteger(index: array of integer; AValue: integer);
    procedure SetAsBoolean(index: array of integer; AValue: boolean);
    procedure SetAsCurrency(index: array of integer; Value: Currency);
    procedure SetAsInt64(index: array of integer; Value: Int64);
    procedure SetAsDate(index: array of integer; Value: TDateTime);
    procedure SetAsLong(index: array of integer; Value: Long);
    procedure SetAsTime(index: array of integer; Value: TDateTime);
    procedure SetAsDateTime(index: array of integer; Value: TDateTime);
    procedure SetAsDouble(index: array of integer; Value: Double);
    procedure SetAsFloat(index: array of integer; Value: Float);
    procedure SetAsShort(index: array of integer; Value: Short);
    procedure SetAsString(index: array of integer; Value: AnsiString);
    procedure SetAsVariant(index: array of integer; Value: Variant);
    procedure SetBounds(dim, UpperBound, LowerBound: integer);
    function GetAttachment: IAttachment;
```

```
    function GetTransaction: ITransaction;
    procedure AddEventHandler(Handler: TArrayEventHandler);
    procedure RemoveEventHandler(Handler: TArrayEventHandler);
  end;
```

This interface provides the getters and setters for array elements of each data type available for arrays. In this case, each getter and setter requires an index that is an array of integers, with one integer for each dimension. The order in which the integers are provided is the same as in which the bounds are described in the metadata. Automatic type conversion takes place whenever types are compatible and follows the same rules as for ISQLData and ISQLParam. IArray inherits from IArrayMetaData.

Additionally:

Method	Description
GetArrayID	Returns the internal array ID. This is an ISC_QUAD (see Blobs). Any changes will be saved at this point.
Clear	Re-initialises the array to an empty array
IsEmpty	Returns true if the array is empty; an array is empty when it is first created or after a call to “clear”.
PreLoad	Normally an array only reads its data from the database the first time a getter method is called. PreLoad forces a database read before any getter method is called.
CancelChanges	Cancel any unsaved changes and restores the array to its initial state (new arrays) or refreshes the array from the database.
SaveChanges	Forces a write to the database of any changes to the array.
SetBounds	Restricts the IArray to a subrange of the array held in the database.
GetAttachment	Returns the database IAttachment interface for the Blob
GetTransaction	Returns the transaction used to create/open the Blob.
AddEventHandler	See 8.7
RemoveEventHandler	See 8.7

8.3 Reading Array Data

Array data is read from the database in much the same way as blob data.

When a dataset's column has an SQL Type of SQL_ARRAY and the field in the current row is non-null, then the ISQLData's AsArray property (see 6.5) may be used to access the array using the IArray interface. For example, assuming that MyTable has an integer RowID column and an array column named MyArrayColumn, for an one dimensional array of integers:

```
var theResults: IResultSet;
    theArray: IArray;
    Bounds: TArrayBounds;
    i,j: integer;
begin
  MyStatement := MyAttachment.Prepare(MyTransaction,
                    'Select MyArrayColumn,RowID From MyTable Where RowID < ?');
  MyStatement.SQLParams[0].AsInteger := 10;
  theResults := MyStatement.OpenCursor;
  while theResults.FetchNext do
  begin
    theArray := theResults[0].AsArray;
    if theArray.GetDimensions = 1 then
    begin
     Bounds := theArray.GetBounds;
     for i := Bounds[0].LowerBound to Bounds[0].UpperBound do
        write('(',i,': ',theArray.GetAsString([i]),') ');
     writeln;
    end;
  end;
  theResults.Close;
```

The above will write out all array values and their index. Although an integer array is assumed for the example, the above should work for all array types that can be converted to strings.

Note that an alternative method exists for accessing an array using the IAttachment.OpenArray method.

```
function OpenArray(transaction: ITransaction;
              RelationName, ColumnName: AnsiString; ArrayID: TISC_QUAD): IArray;
```

This method accesses an array and returns an IArray interface to it provided that the Relation (Table) Name, Column Name and the ArrayID is known. The ArrayID is returned using the ISQLData.AsQuad property This method allows for ArrayID to be read and stored for later use, opening the array only when required.

8.4 Creating or Modifying an Array

The API user cannot use the ISQLParam interface to determine array metadata. Instead, an appropriate array has to be created using *a priori* knowledge. An array is created using the IAttachment.CreateArray method:

```
function CreateArray(transaction: ITransaction; RelationName,
                                          ColumnName: AnsiString): IArray; overload;
function CreateArray(transaction: ITransaction;
                             ArrayMetaData: IArrayMetaData): IArray; overload;
```

Two variations are defined which differ in the way that the array metadata is identified. The first variant provides a relation (i.e. table) name and a column name in that table. Database metadata is then looked up and used to create an array of a type compatible with the column definition.

The second variant achieves the same but provides the array metadata directly as an IArrayMetaData interface. This variant is useful when a Select Statement has already been prepared for the table in which the array is to be assigned and hence the metadata is already

available client side. The IArrayMetaData interface is obtain by calling the IColumnMetaData.GetArrayMetaData method.

Once an array has been created, data can be written to the array and the array identifier assigned to a field in a database table. For example:

```
var MyArray: IArray;
    MyStatement: IStatement;
begin
  MyArray := MyAttachment.CreateBlob(MyTransaction,'MyTable','MyArrayColumn');
  MyArray.SetAsInteger([0],1);
  {other array element values may also be assigned here}

  MyStatement := MyAttachment.Prepare(MyTransaction,
        'Update MyTable Set MyArrayColumn = ? Where RowID = ?');
  MyStatement.SQLParams[0].AsArray := MyArray;
  MyStatement.SQLParams[1].AsInteger := 1;
  MyStatement.Execute;
```

In the above example, a compatible array is created for the MyArrayColumn in MyTable, and its element values assigned. An Update SQL Statement is then used to save the newly created array in the database.

Note that it is important that the same transaction is used to both create the array and to execute the update statement.

8.5 Reducing Array Bounds

If only a small subrange of a very large array needs to be accessed or modified, the IArray interface provide the **SetBounds** method to reduce the amount of data transferred between client and servier.

```
procedure SetBounds(dim, UpperBound, LowerBound: integer);
```

This method may be called once per dimension in order to reduce the upper and/or lower bounds of the array. This does not change the definition of the array in the database or its metadata. It just reduces the range in which the IArray operates.

A call to SetBounds always re-initialises the array and writes out any changes before it is actioned. It is therefore important that it is called on an IArray before any element is read or modified, or the PreLoad method is called. Otherwise, it will only increase data transfer overhead instead of reducing it.

8.6 Removing an Array

An existing array is simply removed by either replacing it with a new array in an update statement, or using an Update Statement to set the field to NULL.

8.7 Event Handlers

One or more event handlers may be registered with a given IArray so that modifications to the array can be reported to other parts of your application. The AddEventHandler method registers a new event handler, while the RemoveEventHandler method will remove it from the list of event handlers.

Each event handler must be a typed procedure as follows:

```
TArrayEventReason = (arChanging,arChanged);
TArrayEventHandler = procedure(Sender: IArray;
                                        Reason: TArrayEventReason) of object;
```

As implied by the Event Reason parameter, the event handler is called once before a change is applied and once after it is applied.

9

Working with Character Sets

Ideally all applications and databases would work with the same universal character set (e.g. UTF8). However, while increasingly this is true, there will always be exceptions due to legacy databases and applications, and to handle characters that are for one reason or another outside of UTF8.

A Firebird Database can specify a wide range of character sets for character and text mode blob columns. A client application can choose to read each column in its native character set or to have the Firebird Client library transliterate on its behalf.

Furthermore, from FPC 3.0.0 onwards, FPC AnsiStrings have their code page as a property of the string (see http://wiki.freepascal.org/FPC_Unicode_support#DefaultSystemCodePage), where the code page identifies the character set held by the string (e.g. UTF8, ASCII, WIN1252, etc).

The Firebird Pascal API aims to ensure that the AnsiString code page for strings returned by the database API is appropriate for the text data received from the database. When strings are written to the database, the API again aims to ensure that, if necessary, text strings are transliterated from the string data's code page to the character set expected by the database.

9.1 Firebird Character Sets

Mainly for legacy reasons, Firebird supports a wide range of character sets including UTF8, ASCII, ISO 8859 variants, Cyrillic, Chinese, Thai, Korean and Japanese character sets. It also supports two untyped character sets: NONE and OCTETS. Firebird character sets either have a fixed byte width of one (e.g. ASCII) or a variable byte width (UTF8 characters can be up to four bytes in length). Fixed two byte character sets, such as UTF16, are not supported. The character set determines both text data semantics and collation ordering.

The character sets supported by a Firebird database can be listed by the query:

```
Select RDB$CHARACTER_SET_NAME,RDB$CHARACTER_SET_ID
                                        from RDB$CHARACTER_SETS order by 2
```

When a database is created, a default character set for the database is also defined. This is the character set used for fixed and variable length text and text mode Blobs unless a different character set is explicitly given when a column's data type is defined or updated.

9.2 The Database Connection and the Default Character Set

Connections to Firebird databases may also have a default character set defined. This does not have to be the same character set as that defined when the database was created. When a connection default character set is defined, all text data in the database is returned in that character set, transliterating if necessary. The only exception is for database columns with a character set of NONE or OCTETS. In this case transliteration never occurs.

Likewise, all data sent to the database is expected to be in the connection default character set and may, if necessary, be transliterated when it is saved if the target column's character set is different.

A connection default character set is defined by adding an isc_dpb_lc_ctype parameter to the DPB used to connect to the database, and setting its value to the character set name. The character set name must be a character set name recognised by Firebird. e.g.

```
MyDPB.Add(isc_dpb_lc_ctype).AsString := 'UTF8';
```

When no default connection character set is defined then all text is returned in whatever character set is stored in the database. No transliteration takes place. Similarly, all data sent to the database is assumed to be in the correct character set.

Note that if the text data sent to the database contains invalid byte sequences for the column's character set or the default connection character set, if any, then a Database Engine error will be raised indicating a transliteration error.

9.3 Code Pages

The term “code page” refers to the character set and associated collation rules used for an entire application or for each string processed by the application.

FPC from 3.0.0 onwards associates a code page with each AnsiString. The default is usually UTF8, referred to by the symbolic constant CP_UTF8. Many other code pages exist including the CP_NONE code page which is used for untyped string data. AnsiStrings carry the codepage with them. It is also possible to transliterate a string from one code page to another if necessary.

In Lazarus, it is generally advisable to keep to UTF8 as many LCL routines implicitly assume UTF8.

9.4 Transliteration Rules

For all text columns including text mode Blobs, the IColumnMetadata provides the character set applicable to the text and the associated code page. The character set will be either:

- The connection default character set, or
- Character set 0 (NONE) or 1 (OCTETS), when the original column has a character set of NONE or OCTETS
- The character set used to define the column, if no connection default character set is specified.

The code page indicated in the IColumnMetadata is always the code page associated with the Firebird Character Set. The IFirebirdAPI interface provides a set of functions that can be used to query the mapping table between the two. i.e.

```
function GetCharsetName(CharSetID: integer): AnsiString;
function CharSetID2CodePage(CharSetID: integer;
                                    var CodePage: TSystemCodePage): boolean;
function CodePage2CharSetID(CodePage: TSystemCodePage;
                                    var CharSetID: integer): boolean;
function CharSetName2CharSetID(CharSetName: AnsiString;
                                  var CharSetID: integer): boolean;
function CharSetWidth(CharSetID: integer; var Width: integer): boolean;
```

When a string is returned from the Firebird Pascal API (e.g. using ISQLData.AsString), the string's code page will be set to the code page given by the column metadata.

When a string is assigned to a field using the Firebird Pascal API (e.g. using ISQLParam.AsString), the string's code page is compared with that specified in the column metadata. If they are different then the string is transliterated into the code page specified by the column metatadata before it is transferred to the database.

Note that the Firebird Character Set “NONE” is mapped to codepage CP_ACP i.e. the default ANSI code page, while the character set “OCTETS” is mapped to codepage CP_NONE. The former reflects the fact it occurs typically in legacy databases where the system default character set is assumed, while the latter is used for untyped binary data.

9.5 Text Blob Handling

Text Mode Blobs generally behave the same as fixed length and variable length text columns in respect of character sets, code pages and transliteration. The Blob metadata identifies both the character set and code page used to transfer the Blob.

However, when Blob Filters are used for text mode Blobs, no transliteration takes place regardless of the code page of the string or database column. If you use a Blob Filter for a text mode Blob, the API assumes that you know what you are doing.

10

Handling Error Conditions

Except for the exceptions discussed below, the Firebird Pascal Client API handles all errors by throwing an exception. This may be an **EIBClientError** exception or an **EIBInterBaseError** exception.

```
  EIBError = class(EDatabaseError)
   private
     FSQLCode: Long;
   public
     constructor Create(ASQLCode: Long; Msg: AnsiString);
     property SQLCode: Long read FSQLCode;
   end;

   { EIBInterBaseError - Firebird Engine errors}

   EIBInterBaseError = class(EIBError)
   private
     FIBErrorCode: Long;
   public
     constructor Create(Status: IStatus); overload;
     constructor Create(ASQLCode: Long; AIBErrorCode: Long; Msg: AnsiString);
overload;
     property IBErrorCode: Long read FIBErrorCode;
   end;

   {IB Client Exceptions}
   EIBClientError = class(EIBError);
```

The EIBInterBaseError exception is used to report errors returned by the Firebird Database Engine, while the EIBClientError is used to report Firebird Pascal Client API exceptions. EIBClientError is also used by *IBX for Lazarus*.

For an EIBInterBaseError exception, the SQLCode property corresponds to the SQL Error Code defined for the Firebird error, and the IBErrorCode property corresponds to the Firebird EngineCode. Symbolic names for each defined Firebird EngineCode may be found in the IBErrorCodes unit.

10.1 Exceptional Error Handling Cases

The IFirebirdAPI methods **OpenDatabase** and **CreateDatabase** by default also return exceptions on error. However, these two calls also have an optional paramters **RaiseExceptionOnError**. By default this is true. If set to false, then the methods return silently on error and return a nil interface pointer.

In this case, the error can still be identified and handled using the IStatus interface.

10.2 The IStatus Interface

The IStatus interface is returned from a call for the **GetStatus** method of the IFirebirdAPI. It is defined as:

```
  IStatus = interface
    function GetIBErrorCode: Long;
    function Getsqlcode: Long;
    function GetMessage: AnsiString;
    function CheckStatusVector(ErrorCodes: array of TFBStatusCode): Boolean;
    function GetIBDataBaseErrorMessages: TIBDataBaseErrorMessages;
    procedure SetIBDataBaseErrorMessages(Value:
                TIBDataBaseErrorMessages);
  end;
```

The GetIBErrorCode and Getsqlcode methods can be used to query the Firebird Engine Code and SQL Error Code returned by the last Firebird Client API call, and GetMessage can be used to query the text version of the error.

It is also possible to raise a EIBInterBaseError exception from an IStatus interface by the following code:

```
var Status: IStatus;
…
Status := FirebirdAPI.GetStatus;
raise EIBInterBaseError.Create(Status);
```

The IStatus interface can also be used to customise the error message returned, using the **IStatus.SetIBDataBaseErrorMessages** method. This can be used to set any combination of:

```
TIBDataBaseErrorMessage    = (ShowSQLCode,
                              ShowIBMessage,
                              ShowSQLMessage);
```

- ShowSQLCode adds the integer value of the SQL Error Code.
- ShowSQLMessage adds the text message associated with the SQL Error Code.
- ShowIBMessage adds the error message correspondiing to the EngineCode.

By default the generated error message contains all parts.

11

Working with Events

Firebird Events are alerts raised outside of the normal process flow and are generated from the Firebird Procedure and Trigger Language using the “POST_EVENT” PSQL Statement. POST_EVENT queues a name alert which is sent to all active database clients which have registered to receive that alert.

11.1 The IEvents Interface

The IEvents interface is used to register for one or more named events and to wait either synchronously or asynchronously for the event.

```
IEvents = interface
    procedure GetEvents(EventNames: TStrings);
    procedure SetEvents(EventNames: TStrings); overload;
    procedure SetEvents(EventName: AnsiString); overload;
    procedure Cancel;
    function ExtractEventCounts: TEventCounts;
    procedure WaitForEvent;
    procedure AsyncWaitForEvent(EventHandler: TEventHandler);
    function GetAttachment: IAttachment;
end;
```

An IEvents interface is returned by IAttachment.GetEventHandler

```
function GetEventHandler(Events: TStrings): IEvents; overload;
function GetEventHandler(Event: AnsiString): IEvents; overload;
```

Two variants of GetEventHandler are provided, creating an event handler for either one named event or for a list of events. Although an IEvents interface is created for one or more named events, the events on which the interface is waiting can be modified at any time using IEvents.SetEvents.

11.2 Asynchronous Event Handling

Creating an IEvents interface does not of itself cause the client to register for any events. This only happens when the caller explicitly waits for an event. To wait for an event asynchronously, the AsyncWaitForEvent method is called specifying an event callback with the type:

```
TEventHandler = procedure(Sender: IEvents) of object
```

AsyncWaitForEvent always returns immediately. When an alert is received from the database server for any of the named events handled by the IEvents, the event handler callback is called.

Note: the callback will occur within a different thread to the application main thread. It is the responsibility of the programmer to ensure that proper inter-thread communication takes place. This may include the use of TThread.Synchronize in order to process the alert within the main thread, or to use thread synchronisation mechanisms such as Critical Sections.

When a callback procedure is called, it should, at some point call IEvents.ExtractEventCounts to determine which event has been posted and, if necessary to check the event counts. IEvents.ExtractEventCounts returns a TEventCounts array, with one element for each event.

```
TEventInfo = record
    EventName: AnsiString;
    Count: integer;
end;

TEventCounts = array of TEventInfo;
```

For each event, a counter is returned giving the number of times the event has been seen on this connection. An increased event count from the last callback indicates that the named event has been raised.

AsyncWaitForEvent is a “one shot”. Once a callback has occurred, another call to AsyncWaitForEvent must be made in order to wait for more events.

AsyncWaitForEvent can be cancelled using the IEvents.Cancel method.

Changing the event names using IEvents.SetEvents is also an implicit Cancel and AsyncWaitForEvent should be called after changing the set of event names.

11.3 Synchronous Event Handling

The IEvents interface also supports a synchronous wait with the calling thread becoming blocked until an alert is received. It is unlikely that this will ever be used in an application's main thread, but may be appropriate for multi-threaded applications.

The IEvents.WaitForEvent method is used for a synchronous wait and returns only when an alert is raised. IEvents.ExtractEventCounts must still be used to check which event was raised.

IEvents.Cancel should allow a separate thread to cancel a synchronous wait.

12

Working with Services

The Service Manager allows you to perform database maintenance tasks such as database backup and restore, shutdown and restart, garbage collection, and scanning for invalid data structures. It also supports creating, modifying, and removing user entries in the security database, and requesting information about the configuration of databases and the server. As with attaching to a database, attaching to a Service Manager requires a Service Parameter Block (SPB).

12.1 The Service Parameter Block (SPB)

An SPB is allocated using the IFirebirdAPI.AllocateSPB method and has the usual set of methods for a parameter block (see):

```
ISPB = interface
    function getCount: integer;
    function Add(ParamType: byte): ISPBItem;
    function getItems(index: integer): ISPBItem;
    function Find(ParamType: byte): ISPBItem;
    property Count: integer read getCount;
    property Items[index: integer]: ISPBItem read getItems; default;
end;
```

This interface follows the pattern established for the DPB (see), with the Add method used to add a new item, a Find method to locate an existing item and the means provided to enumerate a SPB. The ISPBItem is subclass of the IParameterBlockItem interface (see):

```
ISPBItem = interface(IParameterBlockItem) end;
```

For example:

```
var MySPB: ISPB;
begin
  MySPB := FirebirdAPI.AllocateSPB;
  MySPB.Add(isc_spb_user_name).AsString := 'SYSDBA';
  MySPB.Add(isc_spb_password).AsString := 'masterkey';
```

This example creates an SPB and adds a user name and password to the SPB. This provides the login credentials. The login user must have sufficient privilege to use the Service Manager for the intended purpose.

12.2 Attaching to the Service Manager

A connection is established with the Service Manager on a given Firebird Server using the IFirebirdAPI.GetServiceManager method:

```
function GetServiceManager(ServerName: AnsiString; Protocol: TProtocol;
                                                    SPB: ISPB): IServiceManager;
```

This method requires the name of the server, the protocol use to connect to the server and the SPB. Alternative (and largely historical) protocols to TCP are discussed in the InterBase 6.0 API Guide. In most cases, the protocol should be set to TCP, and the Server Name is the server's domain name.

The IFirebirdAPI.GetServiceManager method returns an IServiceManager interface:

```
IServiceManager = interface
    function getSPB: ISPB;
    function getServerName: AnsiString;
    procedure Attach;
    procedure Detach(Force: boolean=false);
    function IsAttached: boolean;
    function AllocateSRB: ISRB;
    function AllocateSQPB: ISQPB;
    procedure Start(Request: ISRB);
    function Query(SQPB: ISQPB; Request: ISRB) :IServiceQueryResults; overload;
    function Query(Request: ISRB) :IServiceQueryResults; overload;
end;
```

12.2.1 IServiceManager Reference

Method Name	Description
getSPB	Returns the SPB used to attached to the service manager
getServerName	Returns the attached Server Name
Attach	Reattach to the service manager
Detach	Detach from the service manager
IsAttached	Returns true if a connection exists to the service manager
AllocateSRB	Returns an empty Service Request Block (SRB)

AllocateSQPB	Returns an empty Service Query Parameter Block (SQPB)
Start	Starts the service requested by the SRB
Query	Queries an active service, or requests information from the server, or sets properties

12.3 Starting a Service

A service is started using the IServiceManager.Start method and by providing an appropriate Service Request Block (SRB). The SRB specifies the service to run and any parameters needed.

The method returns when the service is started or raises an exception if it is unable to start the requested service.

12.3.1 The Service Request Block (SRB)

An empty SRB is created by the IServiceManager.AllocateSRB method, which returns an ISRB interface to the SRB.

```
ISRB = interface
    function getCount: integer;
    function Add(ParamType: byte): ISRBItem;
    function getItems(index: integer): ISRBItem;
    function Find(ParamType: byte): ISRBItem;
    property Items[index: integer]: ISRBItem read getItems; default;
end;
```

This follows the same approach for creating and maintaining parameter blocks as used for other parameter blocks, such as the DPB (see). New SRB items are added using the Add method and can be returned by type using the find method. The SRB items may also be enumerated using the Items property. An SRB item is accessed using the ISRBItem interface, which is a subclass of the IParameterBlockItem interface (see)

```
ISRBItem = interface(IParameterBlockItem) end;
```

For example:

```
  Req := Service.AllocateSRB;
  Req.Add(isc_action_svc_backup); {Request the backup service}
  Req.Add(isc_spb_dbname).AsString := 'MyDatabase'; {this is assumed to be an
                                                     alias}
  Req.Add(isc_spb_bkp_file).AsString := '/home/backups/mydatabase.gbk';
  try
    Service.Start(Req);
```

starts a backup service and requests that “MyDatabase” on “MyServer” is backed up to the file '/home/backups/mydatabase.gbk' on the server.

12.3.2 List of Services

The following services may be started on the Service Manager. In each case, the service is requested by creating an SRB, adding the symbolic constant for the service (no value), and then adding each parameter (and parameter value) as specified for the service.

Service	Parameters	Purpose
isc_action_svc_display_user		List active users
isc_action_svc_db_stats	isc_spb_dbname, isc_spb_options	Requesting Database Statistics
isc_action_svc_backup	isc_spb_dbname, isc_spb_bkp_file, isc_spb_bkp_length, isc_spb_bkp_factor, isc_spb_options, isc_spb_verbose	Database backup
isc_action_svc_restore	isc_spb_dbname, isc_spb_bkp_file, isc_spb_res_length, isc_spb_res_buffers, isc_spb_res_page_size, isc_spb_res_access_mode, isc_spb_options, isc_spb_verbose	Database Restore

See the InterBase 6.0 API Guide for a detailed description of each parameter and parameter value

12.4 Querying a Service

The IServiceManager.Query method is used to either:

- Query a running service (e.g. a database backup)
- Request information from the service manager.
- Set Database or Server Properties.

Two variants of the IServiceManager.Query method are provided. One provides both a Service Query Parameter Block (SQRB) and a Service Request Block (SRB). The other requires only an SRB. Both variants return an interface to the Service Query Results (SQR). The SQRB is often not required and hence a variation is defined that omits it from the interface.

12.4.1 The Service Query Parameter Block (SQRB)

The Service Query Parameter Block (SQRB) follows the standard pattern for parameter blocks (see Error: Reference source not found), and is defined as:

```
ISQPB = interface
    function getCount: integer;
    function Add(ParamType: byte): ISQPBItem;
    function getItems(index: integer): ISQPBItem;
    function Find(ParamType: byte): ISQPBItem;
    property Count: integer read getCount;
    property Items[index: integer]: ISQPBItem read getItems; default;
end;

ISQPBItem = interface(IParameterBlockItem)
    function CopyFrom(source: TStream; count: integer): integer;
end;
```

Note that the ISQPBItem interface extends the standard IParameterBlockItem interface to include a CopyFrom (stream) method.

There are only two current uses for the SQPB:

1. An SQPB which includes an isc_info_svc_timeout parameter is used to provide a timeout (in seconds and encoded as an integer) to limit the response time for a Query. The query returns within this time if it cannot otherwise complete.

2. An SQPB which includes an isc_info_svc_line parameter is used to return data (in successive chunks) for the service's *stdin*. The parameter value provides the data as a long string (up to 65535 bytes), although in practice as the whole buffer is limited to 65535 bytes, the actual string length must be reduced to take account of the parameter byte, length field and any isc_info_svc_timeout parameter and value.

The latter case is used for database restore (see 12.6). The CopyFrom method provides a potentially more convenient way of setting the string value from a TStream. This method is used to set the value of an isc_info_svc_line parameter as up to “count” bytes” read from the current stream position (zero implies the maximum possible). It returns the actual number of bytes written. The CopyFrom method will always try and maximise the use of the parameter buffer and will use up all remaining space.

12.4.2 The Service Request Block (SRB)

If a service has already been started and is still active then the SRB is used to request information from the service. Otherwise, it is used to request information or to set database or server properties.

The Service Request Block (SRB) follows the standard pattern for parameter blocks (see Error: Reference source not found), and is defined as:

```
ISRB = interface
    function getCount: integer;
    function Add(ParamType: byte): ISRBItem;
    function getItems(index: integer): ISRBItem;
    function Find(ParamType: byte): ISRBItem;
    property Count: integer read getCount;
    property Items[index: integer]: ISRBItem read getItems; default;
end;
```

```
ISRBItem = interface(IParameterBlockItem) end;
```

12.4.2.1 Running Services

Service	Request	Information Requested
List Users	isc_info_svc_get_users	A list of active users
Database Statistics	isc_info_svc_line	Returns the next line of (plain text) output from the database statistics service
Database Backup	isc_info_svc_line	The next line of (plain text) output from the database backup service (backup to server file only)
	isc_info_svc_to_eof	The next chunk of the backup archive (backup to stdout only).
Database Restore	isc_info_svc_line	The next line of (plain text) output from the database restore service
	isc_info_svc_stdin	The maximum acceptable byte count to return as stdin data.

12.4.2.2 Information Requests

Information Group	Request	Information Returned
Version Information	isc_info_svc_version	The version of the Services Manager
	isc_info_svc_server_version	The version of the Firebird server
	isc_info_svc_implementation	The implementation string, or platform, of the server
Configuration Parameters	isc_info_svc_get_env_lock	The location of the Firebird lock manager file on the server
	isc_info_svc_get_config	Contents of Firebird.conf

	isc_info_svc_get_env	Location of the Firebird root directory on the server
	isc_info_svc_get_env_msg	Location of the Firebird messages file on the server.
	isc_info_svc_user_dbpath	Location of the security database on the server
Database Information	isc_info_svc_svr_db_info	The number of database attachments and databases currently active on the server
Limbo Transactions	isc_info_svc_limbo_trans	Limbo transaction information for unresolved distributed transactions

12.4.2.3 Setting Properties

These requests correspond to the options of the gfix command line utility and are described in the InterBase 6.0 API Guide Table 12.5

12.4.3 The Query Response

The Query method, on successful completion, returns an IServiceQueryResults interface to the Service Query Response. This can be enumerated to determine and process the query results. This interface is defined as:

```
IServiceQueryResults = interface
    function getCount: integer;
    function getItem(index: integer): IServiceQueryResultItem;
    function find(ItemType: byte): IServiceQueryResultItem;
    property Items[index: integer]: IServiceQueryResultItem
                                                    read getItem; default;
    property Count: integer read getCount;
end;
```

The main use of this interface is to process “Count” items in turn. However, it may also be used to locate (find) a specific response item by its symbolic constant. Each response item is return as an IServiceQueryResultItem interface.

```
IServiceQueryResultSubItem = interface
    function getItemType: byte;
    function getSize: integer;
    procedure getRawBytes(var Buffer);
    function getAsString: AnsiString;
    function getAsInteger: integer;
    function getAsByte: byte;
    function CopyTo(stream: TStream; count: integer): integer;
    property AsString: AnsiString read getAsString;
    property AsInteger: integer read getAsInteger;
    property AsByte: byte read getAsByte;
  end;

IServiceQueryResultItem = interface(IServiceQueryResultSubItem)
    function getCount: integer;
    function getItem(index: integer): IServiceQueryResultSubItem;
    function find(ItemType: byte): IServiceQueryResultSubItem;
    property Items[index: integer]: IServiceQueryResultSubItem
                                                    read getItem; default;
    property Count: integer read getCount;
end;
```

Each query response may either be a single value (for a given response type) or may itself be a list of values. This is reflected in the IServiceQueryResultItem interface where the response may either be accessed using the getters for the (sub) item type or itself enumerated for “sub items”.

The responses that may be expected for each service request are identified in the InterBase 6.0 API Guide. Appendix B. provides an example of the enumeration of a service response. The **CopyTo** method applies to string item types only and writes the contents of the string to the supplied stream.

12.5 Detaching from the Service Manager

The Service Manager connection is automatically closed when the service manager interface goes out of scope. It may also be explicitly closed by called the Detach method. In the latter case, the Attach method may be called to reconnect to the service manager.

12.6 Backup and Restore Services

Database backup and restore may be performed either to or from a file on the server, or to and from a file on the client system.

12.6.1 Backup and Restore on the Server

Backup and restore to or from a file on the server is the simple case. All that is necessary is to specify the name of the database and backup file(s) and to monitor the output of the process until completion. For example, for backup:

```
var SPB: ISPB;
    Service: IServiceManager;
    Req: ISRB;
    Results: IserviceQueryResults;
    Response: IServiceQueryResultSubItem;
    line: AnsiString;
begin
  SPB := FirebirdAPI.AllocateSPB;
  SPB.Add(isc_spb_user_name).setAsString('SYSDBA');
  SPB.Add(isc_spb_password).setAsString('masterkey');
  Service := FirebirdAPI.GetServiceManager('myserver domain name',TCP,SPB);
```

```
  Req := Service.AllocateSRB;
  Req.Add(isc_action_svc_backup);
  Req.Add(isc_spb_dbname).AsString := 'MyDatabase';
  Req.Add(isc_spb_bkp_file).AsString := 'path to backup file';
  Req.Add(isc_spb_verbose);
  try
    Service.Start(Req);
    Req := Service.AllocateSRB;
    Req.Add(isc_info_svc_line);
    repeat
      line := '';
      Results := Service.Query(Req);
      Response := Results.Find(isc_info_svc_line);
      if Response <> nil then
      begin
        line :=  Response.AsString;
        writeln(line);
      end;
    until line = '';
    writeln('Backup Complete');
  except on E: Exception do
    writeln('Backup Service Error: ',E.Message);
  end;
end;
```

The above example, starts the service and then loops round issuing queries until there is no response. Database restore follows the same pattern.

12.6.2 Backup and Restore using a File on the Client System

In principle, this is the same as the above except that the backup file is set to 'stdout' (for backup) or 'stdin' for restore. The client additionally has to process the stdout data or to provide the stdin data.

The stdout data for backup is provided by the isc_info_svc_to_eof request item. This is mutually exclusive with verbose output. The following example illustrates how backup to stdout is processed.

```
var SPB: ISPB;
    Service: IServiceManager;
    Req: ISRB;
    Results: IServiceQueryResults;
    Response: IServiceQueryResultSubItem;
    bakfile: TFileStream;
    bytesWritten: integer;
begin
  SPB := FirebirdAPI.AllocateSPB;
  SPB.Add(isc_spb_user_name).setAsString('SYSDBA');
  SPB.Add(isc_spb_password).setAsString('masterkey');
  Service := FirebirdAPI.GetServiceManager('myserver domain name',TCP,SPB);

  bakfile := TFileStream.Create('path to backup file',fmCreate);
  Req := Service.AllocateSRB;
  Req.Add(isc_action_svc_backup);
  Req.Add(isc_spb_dbname).AsString := 'MyDatabase';
  Req.Add(isc_spb_bkp_file).AsString := 'stdout';
  try
    Service.Start(Req);
    Req := Service.AllocateSRB;
    Req.Add(isc_info_svc_to_eof);
    repeat
      bytesWritten := 0;
```

```
      Results := Service.Query(Req);
      Response := Results.Find(isc_info_svc_to_eof);
      if Response <> nil then
        bytesWritten := Response.CopyTo(bakfile,0);
    until bytesWritten = 0;
    writeln('Backup Complete');
  except on E: Exception do
   begin
    writeln('Backup Service Error: ',E.Message);
    bakfile.free;
    DeleteFile('path to backup file');
   end;
  end;
  bakfile.free;
end;
```

Restore to stdin is more complex and requires use of the Service Query Parameter Block to upload the data, as shown in the following example:

```
var SPB: ISPB;
    Service: IServiceManager;
    Req: ISRB;
    Results: IServiceQueryResults;
    bakfile: TFileStream;
    bytesWritten: integer;
    bytesAvailable: integer;
    i: integer;
    ReqLength: integer;
    SQPB: ISQPB;
begin
  SPB := FirebirdAPI.AllocateSPB;
  SPB.Add(isc_spb_user_name).setAsString('SYSDBA');
  SPB.Add(isc_spb_password).setAsString('masterkey');
  Service := FirebirdAPI.GetServiceManager('myserver domain name',TCP,SPB);

  bakfile := TFileStream.Create('path to backup file',fmOpenRead);
  bytesAvailable := BakFile.Size;
  Req := Service.AllocateSRB;
  Req.Add(isc_action_svc_restore);
  Req.Add(isc_spb_dbname).AsString := 'MyDatabase';
  Req.Add(isc_spb_bkp_file).AsString := 'stdin';
  Req.Add(isc_spb_verbose);
  Req.Add(isc_spb_options).SetAsInteger(isc_spb_res_create);
  try
      ReqLength := 0;
      repeat
        SQPB := Service.AllocateSQPB;
        if ReqLength > 0 then
            bytesWritten :=
                    SQPB.Add(isc_info_svc_line).CopyFrom(BakFile,ReqLength);
        bytesAvailable -= bytesWritten;
        Req := Service.AllocateSRB;
        Req.Add(isc_info_svc_stdin);
        Req.Add(isc_info_svc_line);
        Results := Service.Query(SQPB,Req);

        {Now process the query response}
        for i := 0 to Results.Count – 1 do
        case Results[i].getItemType of
        isc_info_svc_stdin:
          ReqLength := Results[i].AsInteger;
        isc_info_svc_line:
          writeln(Results[i].AsString);
        end;
     until (ReqLength = 0) ;
      writeln('Local Restore Complete');
```

```
   except on E: Exception do
   begin
    writeln('Restore Service Error: ',E.Message);
    bakfile.free;
   end;
  end;
  bakfile.free;
end;
```

13

Deployment Guidelines

The *fbintf* package is compiled into your application and does not itself require any special procedures for deployment on operational systems. However, it does depend on the availability of the Firebird Client library or the Firebird embedded server. This chapter provides guidelines on how to distribute Firebird with your application. This necessarily differs between platforms, and between Firebird Versions.

13.1 Deployment on Windows

Probably the simplest approach is just to require the installation of Firebird from the distribution package provided on http://www.firebirdsql.org. Indeed, this is the recommended approach for a development system. However, this can be simplified and use of an embedded server requires special consideration.

13.1.1 Firebird 2.5 and Earlier

13.1.1.1 Firebird Client Only

This is the simplest case where your application will be running on a client system accessing a database on a remote server. In this case, all you need to do is to install, in the same folder as your application is installed, the following Firebird files:

- fbclient.dll
- firebird.msg
- firebird.conf

These are typically found in the “C:\Program Files\Firebird_2_x” folder and its “bin” subfolder (when installing from the standard Firebird distribution). Note: the three files must be in the same folder as the application executable if *fbintf* is to find them. The firebird.conf should be unmodifed and as originally distributed.

The advantages of this deployment are that your application distributable is minimised and avoids the stealth upgrade problem should the Firebird installation be upgraded unexpectedly and to an incompatible version.

13.1.1.2 The Embedded Firebird Server

If your application is a Personal Database Application. That is, the database resides on the same system as your application, it is single user, intended to be accessible only to its owner, and a separate database login is seen as unnecessary, then the Firebird embedded server should be used when the application is deployed.

The Firebird embedded server may be downloaded from http://www.firebirdsql.org as a single zip archive. The contents of this archive should be installed in the same folder as your application executable. *fbintf* will then automatically find and load the embedded Firebird Server.

13.1.2 Firebird 3.0 and Later

Firebird 3.0 has introduced the concept of “plugins”. Plugins can determine various capabilities and, in particular, whether an installation is client only or includes the embedded server. There is no separate distributable for the Firebird Embedded Server.

13.1.2.1 Firebird Client Only

As with Firebird 2.5 and earlier, the same basic files are required, and copied from the Firebird distribution zip to your application's installation folder:

- fbclient.dll
- firebird.msg
- firebird.conf

However, the firebird.conf file will need to be edited to reflect the plugins provided. In a minimal configuration, the “Providers” parameter line will need to be edited to ensure that the “engine12” plugin is removed i.e.

Providers = Remote,Loopback

The AuthClient parameter line should also reflect the authentication plugins installed. If the server is known to be Firebird 3, then this can be reduced to:

AuthClient = Srp

Finally, subfolder should be created for your application install folder and called “plugins” as a mimimum, this should contain the “srp.dll” file copied from the Firebird distribution zip.

13.1.2.2 Firebird Embedded Server

If your application is a Personal Database Application. That is, the database resides on the same system as your application, it is single user, intended to be accessible only to its owner, and a separate database login is seen as unnecessary, then the Firebird embedded server should be used when the application is deployed.

In Firebird 3.0, the key difference is that, in addition to what has been described above for a client installation, the “engine12.dll” should also be copied to the “plugins” subfolder and the Providers Parameter line should be:

Providers = Remote,Engine12,Loopback

However, for a working installation, additional files will be required from the Firebird Distribution zip.

- All “.conf”, “.dat” and “.dll” files should be copied from the top level zip folder to your application's installation folder.
- “udr_engine.conf” and “udr_engine.dll” should be copied from the “plugins” folder in the Firebird Distribution zip to the “plugins” subfolder in your application's installation folder.
- The “intl” and “udf” folders in the Firebird Distribution zip should be copied to your application's installation folder.

13.2 Deployment on Linux

As with Windows, probably the simplest approach is just to require the installation of Firebird from the distribution package provided on http://www.firebirdsql.org. However, Firebird usually comes as a package and as part of your Linux Distribution and use of these packages is the recommended approach.

13.2.1 Firebird 2.5 and Earlier

13.2.1.1 Firebird Client only

Your application installation should require the installation of the libfbclient2 package (debian) or the libfbclient.so.2 rpm (Fedora).

13.2.1.2 Firebird Embedded Server

Your application installation should require the installation of the libfbembed2.5 package (debian) or the firebird-libfbembed rpm (Fedora).

13.2.2 Firebird 3.0 and Later

13.2.2.1 Firebird Client Only

Your application installation should require the installation of the libfbclient2 package (debian) or the libfbclient2 rpm (Fedora).

13.2.2.2 Firebird Embedded Server

Your application installation should require the installation of the firebird3.0-server-core package (debian) or the firebird-3.0.1 rpm or later (Fedora).

Note that under Debian, the full server is not installed as this additionally requires the firebird3.0-server package.

Appendix A.Parameter Blocks

Many of the Firebird API calls require the use of parameter blocks in order to pass the various parameters and options that the user may set. These include:

- The Database Parameter Block (DPB)
- The Transaction Parameter Block (TPB)
- The Service Parameter Block (SPB)
- The Service Request Block (SRB)
- The Service Query Parameter Block (SQRB)
- The Blob Parameter Block (BPB).

Each has a slightly different format with, for example, variations in the way that integers and strings are encoded, and with no obvious pattern. The *fbintf* aims to hide these differences and to present a standard approach to the user.

In the *fbintf*, parameter blocks are managed opaquely through an interface. An API call is provided to provide an interface to an initially empty parameter block and it is then possible to add parameters to the block and where necessary give a parameter a value. The original symbolic constants defined in the InterBase 6.0 API Guide are used when adding a parameter to a parameter block. In Firebird, each parameter is said to be encoded as a “clumplet”.

It is possible to enumerate all parameters in a block or to find a parameter by its symbolic constant. The current value of a parameter can also be read.

Given the variations in encoding, each type of parameter block has its own strongly typed interface, whilst providing the same functions except for the return types. A generic is used to define the basic interface which is then specialized for each parameter block. However, in order to improve the readability of the interface, each parameter block interface described in this document is presented in its expanded form.

The Parameter Block Interface

The Database Parameter Block (DPB) is used here as an example and is defined as:

```
IDPB = interface
    function getCount: integer;
    function Add(ParamType: byte): IDPBItem;
    function getItems(index: integer): IDPBItem;
    function Find(ParamType: byte): IDPBItem;
    procedure Printbuf;
    property Count: integer read getCount;
    property Items[index: integer]: IDPBItem read getItems; default;
end;
```

- The getCount method returns the number of items in the block.
- The Add method adds a new parameter item and returns an interface to it.

- The getItems method accesses and returns an interface to a parameter by position (in the order the parameters are added.
- The Find method returns an interface to a parameter by parameter type and returns nil if the requested parameter is not found.
- PrintBuf is a debugging aid that formats and prints to stdout (using writelns) the output buffer as hex bytes.

The interface returned to a parameter block item is also strongly typed and is different for each parameter block. The IDPBItem interface is defined as:

```
IDPBItem = interface(IParameterBlockItem) end;
```

That is it's a simple subclass of the generic IParameterBlockItem interface.

The IParameterBlockItem Interface

This is the ancestor for all parameter block item interfaces as is defined as:

```
IParameterBlockItem = interface
    function getParamType: byte;
    function getAsInteger: integer;
    function getAsString: AnsiString;
    function getAsByte: byte;
    procedure setAsString(aValue: AnsiString);
    procedure setAsByte(aValue: byte);
    procedure SetAsInteger(aValue: integer);
    property AsString: AnsiString read getAsString write setAsString;
    property AsByte: byte read getAsByte write setAsByte;
    property AsInteger: integer read getAsInteger write SetAsInteger;
end;
```

The interface's methods should be intuitively understood from their names.

- The getParamType method returns the value of the symbolic constant used to “add” the parameter.

- There are “getters” and “setters” for integer, byte and string values, as well as corresponding properties.

A subclass, such as IDPBItem allows the values of the parameter to be read and written. Note that different value types are appropriate for different parameter types, although all parameter values can be read as strings. The InterBase 6.0 API Guide defines the value types for each parameter type, or even whether or not a value is necessary. The following table identifies the encoding used for each interface.

Interface	Integer Encoding	String Encoding	Byte Encoding
IDPB	No length bytes, (four bytes containing the integer value (LSB first)	Max string length 255 bytes. Encoded as one length byte plus variable number of character bytes	One length byte (always set to one), plus one byte containing the value.
ITPB	Not used	Max string length 255 bytes. Encoded as one	Not used

Interface	Integer Encoding	String Encoding	Byte Encoding
		length byte plus variable number of character bytes	
ISPB	Not used	Max string length 255 bytes. Encoded as one length byte plus variable number of character bytes	Not used
ISQRB	Two length bytes (always set to four), plus four bytes containing the integer value (LSB first)	Max string length 65535 bytes. Encoded as two length bytes plus variable number of character bytes	Not used
ISRB	No length bytes, (four bytes containing the integer value (LSB first)	Max string length 65535 bytes. Encoded as two length bytes plus variable number of character bytes	No length byte. Single byte value.
IBPB	One length byte (always set to four), plus four bytes containing the integer value (LSB first)	Not used	Not used

Example

```
var MyDPB: IDPB;
begin
  MyDPB := FirebirdAPI.AllocateDPB;
  MyDPB.Add(isc_dpb_user_name).AsString := 'SYSDBA';
  MyDPB.Add(isc_dpb_password).AsString := 'masterkey';
  MyDPB.Add(isc_dpb_lc_ctype).AsString := 'UTF8';
  MyDPB.Add(isc_dpb_set_db_SQL_dialect).AsByte := 3;
```

is a typical example of the use of IDPB to populate a DPB prior to attaching to the database. Note that the parameter to the **Add** method is one of the DPB symbolic constants defined by the Firebird API.

The following provides an example of enumerating a DPB to print out each parameter's value:

```
procedure TTestBase.PrintDPB(MyDPB: IDPB);
var i: integer;
begin
  writeln('DPB');
  writeln('Count = ', MyDPB.getCount);
  for i := 0 to MyDPB.getCount - 1 do
    writeln(MyDPB[i].getParamType,' = ', MyDPB[i].AsString);
  writeln;
end;
```

Appendix B. Example Parsing of the Service Response Block

Note: Forward declarations omitted for clarity.

```
function WriteServiceQueryResult(QueryResult: IServiceQueryResults): boolean;
var i: integer;
    line: AnsiString;
begin
  Result := true;
  for i := 0 to QueryResult.GetCount - 1 do
  with QueryResult[i] do
  case getItemType of
  isc_info_svc_version:
    writeln('Service Manager Version = ',getAsInteger);
  isc_info_svc_server_version:
    writeln('Server Version = ',getAsString);
  isc_info_svc_implementation:
    writeln('Implementation = ',getAsString);
  isc_info_svc_get_license:
    writeLicence(QueryResult[i]);
  isc_info_svc_get_license_mask:
    writeln('Licence Mask = ',getAsInteger);
  isc_info_svc_capabilities:
    writeln('Capabilities = ',getAsInteger);
  isc_info_svc_get_config:
    WriteConfig(QueryResult[i]);
  isc_info_svc_get_env:
    writeln('Root Directory = ',getAsString);
  isc_info_svc_get_env_lock:
    writeln('Lock Directory = ',getAsString);
  isc_info_svc_get_env_msg:
    writeln('Message File = ',getAsString);
  isc_info_svc_user_dbpath:
    writeln('Security File = ',getAsString);
  isc_info_svc_get_licensed_users:
    writeln('Max Licenced Users = ',getAsInteger);
  isc_info_svc_get_users:
    WriteUsers(QueryResult[i]);
  isc_info_svc_svr_db_info:
    WriteDBAttachments(QueryResult[i]);
  isc_info_svc_line:
    begin
      line := getAsString;
      writeln(line);
      Result := line <> '';
    end;
  isc_info_svc_running:
    writeln('Is Running = ',getAsInteger);
  isc_info_svc_limbo_trans:
    WriteLimboTransactions(QueryResult[i]);
  isc_info_svc_to_eof,
  isc_info_svc_timeout,
  isc_info_truncated,
```

```
  isc_info_data_not_ready,
  isc_info_svc_stdin:
    {ignore};
  else
    writeln('Unknown Service Response Item ', getItemType);
  end;
  writeln;
end;

procedure WriteDBAttachments(att: IServiceQueryResultItem);
var i: integer;
begin
  writeln('DB Attachments');
  for i := 0 to att.getCount - 1 do
  with att[i] do
  case getItemType of
  isc_spb_num_att:
    writeln('No. of Attachments = ',getAsInteger);
  isc_spb_num_db:
    writeln('Databases In Use = ',getAsInteger);
  isc_spb_dbname:
    writeln('DB Name = ',getAsString);
  end;
end;

procedure WriteLimboTransactions(limbo: IServiceQueryResultItem);
var i: integer;
begin
  writeln('Limbo Transactions');
  for i := 0 to limbo.getCount - 1 do
  with limbo[i] do
  case getItemType of
  isc_spb_single_tra_id:
    writeln('Single DB Transaction = ',getAsInteger);
  isc_spb_multi_tra_id:
    writeln('Multi DB Transaction = ',getAsInteger);
  isc_spb_tra_host_site:
    writeln('Host Name = ',getAsString);
  isc_spb_tra_advise:
    writeln('Resolution Advisory = ',getAsInteger);
  isc_spb_tra_remote_site:
    writeln('Server Name = ',getAsString);
  isc_spb_tra_db_path:
    writeln('DB Primary File Name = ',getAsString);
  isc_spb_tra_state:
    begin
      write('State = ');
      case getAsInteger of
        isc_spb_tra_state_limbo:
          writeln('limbo');
        isc_spb_tra_state_commit:
          writeln('commit');
        isc_spb_tra_state_rollback:
          writeln('rollback');
        isc_spb_tra_state_unknown:
          writeln('Unknown');
      end;
    end;
  end;
end;

procedure writeLicence(Item: IServiceQueryResultItem);
var i: integer;
begin
  for i := 0 to Item.getCount - 1 do
  with Item[i] do
  case getItemType of
```

```
    isc_spb_lic_id:
      writeln('Licence ID = ',GetAsString);
    isc_spb_lic_key:
      writeln('Licence Key = ',GetAsString);
  end;
end;

procedure WriteConfig(config: IServiceQueryResultItem);
var i: integer;
begin
  writeln('Firebird Configuration File');
  for i := 0 to config.getCount - 1 do
    writeln('Key = ',config.getItemType,', Value = ',config.getAsInteger);
  writeln;
end;

procedure WriteUsers(users: IServiceQueryResultItem);
var i: integer;
begin
  writeln('Sec. Database User');
  for i := 0 to users.getCount - 1 do
  with users[i] do
  case getItemType of
    isc_spb_sec_username:
      writeln('User Name = ',getAsString);
    isc_spb_sec_firstname:
      writeln('First Name = ',getAsString);
    isc_spb_sec_middlename:
      writeln('Middle Name = ',getAsString);
    isc_spb_sec_lastname:
      writeln('Last Name = ',getAsString);
    isc_spb_sec_userid:
      writeln('User ID = ',getAsInteger);
    isc_spb_sec_groupid:
      writeln('Group ID = ',getAsInteger);
    else
      writeln('Unknown user info ', getItemType);
  end;
  writeln;
end;
```