FIBPlus Developers Guide
Part II

Working with BLOB fields

There can be advantages in storing non-structured data in your database, such as images, OLE-objects, sounds, etc. For this you will need to use a special data type - BLOB. SQL queries for BLOB fields do not differ from queries for standard field types; work with these fields in TpFIBDataSet does not differ from TDataSet work. The only difference from an ordinary data type is that you should use streams (special TStream descendants) to set values of the BLOB-parameter.

Before setting the BLOB-parameter value you should put the TpFIBDataSet into edit mode by calling TpFIBDataSet.edit (dsEdit).

Use the TFIBCustomDataset method for work with BLOB fields

```
function CreateBlobStream(Field: TField; Mode: TBlobStreamMode): TStream;
override;
```

If you call CreateBlobStream it creates an instance of TFIBDSBlobStream. It enables data exchange between the BLOB-parameter and the stream, which reads the image from the file. The Field parameter defines a BLOB field, on which a stream will be based. The Mode parameter defines the mode.

```
type TBlobStreamMode = (bmRead, bmWrite, bmReadWrite);
```

bmRead	the stream is used to read a BLOB field;
bmWrite	the stream is used to write a BLOB field;
bmReadWrite	the stream is used to modify a BLOB field.

In this example we are using two procedures, one saves a file into BLOB, and the other loads the BLOB field into the file.

```
procedure FileToBlob(BlobField: TField; FileName: string);
var S: TStream; FileS: TFileStream;
begin
  BlobField.DataSet.Edit;
  S := BlobField.DataSet.CreateBlobStream(BlobField, bmReadWrite);
  try
    FileS := TFileStream.Create(FileName, fmOpenRead);
    S.CopyFrom(FileS, FileS.Size);
  finally
    FileS.Free;
    S.Free;
    BlobField.DataSet.Post;
  end;
end;

procedure BlobToFile(BlobField: TField; FileName: string);
var S: TStream;
    FileS: TFileStream;
begin
  if BlobField.IsNull then Exit;
  S := BlobField.DataSet.CreateBlobStream(BlobField, bmRead);
  try
    if FileExists(FileName)
    then FileS := TFileStream.Create(FileName, fmOpenWrite)
    else FileS := TFileStream.Create(FileName, fmCreate);
    FileS.CopyFrom(S, S.Size);
  finally
    S.Free;
    FileS.Free;
  end;
end;
```

If you use TpFIBQuery with BLOB-fields you still use streams, but in contrast to TpFIBDataSet, you should not create any special streams. TFIBSQLDA has built-in SaveToStream and LoadFromStream methods. I.e. for TpFIBQuery you only need to write:

pFIBQuery1.FN('BLOB_FIELD').SaveToStream(FileS);

Using unique FIBPlus field types

FIBPlus has several unique fields: `TFIBLargeIntField`, `TFIBWideStringField`, `TFIBBooleanField`, `TFIBGuidField`.

`TFIBLargeIntField`

It is a BIGINT field in InterBase/ Firebird.

`TFIBWideStringField`

It is used for string fields in UNICODE_FSS charset. In most cases you cannot usea standard TWideString because of multiple VCL errors.

`TFIBBooleanField`

It emulates a logical field. InterBase and Firebird do not have a standard logical field, so FIBPlus enables you to easily emulate it. For this create a domain (INTEGER or SMALLINT) with a BOOLEAN substring in its name. Set psUseBooleanFlields to True in PrepareOptions in TpFIBDataSet. On creating field objects FIBPlus will check the domain name and if they haveBOOLEAN types, FIBPlus will create TFIBBooleanField instances for these fields.

```
CREATE DOMAIN FIB$BOOLEAN AS SMALLINT
DEFAULT 1 NOT NULL CHECK (VALUE IN (0,1));
```

`TFIBGuidField`

It works similar to TFIBBooleanField: a field should be declared in the domain and the domain name should have GUID. psUseGuidField should be set to True. This example illustrates how to declare a domain:

```
CREATE DOMAIN FIB$GUID AS CHAR(16) CHARACTER SET OCTETS;
```

If AutoGenerateValue is set to True, the field values will be set automatically on inserting field values.

How to work with array fields

Since its early versions InterBase enables to use multidimensional array fields and thus to store specialized data in a convenient way. InterBase array fields are not supported by the SQL standard so it’s very difficult to work with such fields using SQL queries. In practice you can use array fields item by item and only in read-only operations. To change array field values you should use special InterBase API commands. FIBPlus helps you to avoid such difficulties and handles array fields itself.

You can see how to work with array fields in the demo example in demo database EMPLOYEE supplied with the server. The LANGUAGE_REQ field in the JOB table is an array (LANGUAGE_REQ VARCHAR(15) [1:5])

The first example shows how to edit an array field using such special methods as TpFIBDataSet ArrayFieldValue and SetArrayValue, as well as GetArrayValues, SetArrayValue and AsQuad in TFIBXSQLDA. These methods enable you to work with this field as a united structure. TFIBXSQLDA.GetArrayElement helps you to get an array element value by index.

TpFIBDataSet.ArrayFieldValue and TFIBXSQLDA.GetArrayValues methods get a variant array from array field values. E.g. use the following code to get separate elements:

```
var v: Variant;
with ArrayDataSet do begin
  v := ArrayFieldValue(FieldByName('LANGUAGE_REQ'));
  Edit1.Text := VarToStr(v[1]);
end;
```

TpFIBDataSet.SetArrayValue and TFIBXSQLDA.SetArrayValue methods enable you to define all field elements as a variant array:

```
with ArrayDataSet do
  SetArrayValue(FBN('LANGUAGE_REQ'), VarArrayOf([Edit1.Text, ...]));
```

Newest FIBPlus versions have a simple solution: you can set the variant array directly:

```
  FBN('LANGUAGE_REQ').Value := VarArrayOf([Edit1.Text, ...]);
```

It's easy to set the field values in the BeforePost event. If Update or Insert operations are not successful, you need to restore the internal array identifier of the editable record. For this you need to refresh the current record by calling the Refresh method. This rule is set by InterBase API functions, so FIBPlus should take it into consideration. So the event handler is very important for working with array fields. You can place it into OnPostError.

```
procedure ArrayDataSetPostError(DataSet: TDataSet; E: EDatabaseError; var
Action: TDataAction);
begin
  Action := daAbort;
  MessageDlg('Error!', mtError, [mbOk], 0);
  ArrayDataSet.Refresh;
end;
```

The TFIBXSQLDA.AsQuad function for array fields and BLOB's has the field BLOB_ID.

Examples ArrayFields1 and ArrayFields2 demonstrate how to work with array fields. ArrayFields1 shows how to extract an array from a field, ArrayFields2 selects an array directly from SelectSQL. Both examples write a field into a database packing it into an array field

Firebird 1.5.X releases have an error, returning the wrong length for string fields. That's whu you can see a strange last symbol «|» in string array elements.

Using TDataSetsContainer containers

The TDataSetContainer component helps developers to use centralized event handling for different TpFIBDataSet components and to send messages to these components forcing them to execute some additional actions. I.e. before opening all TpFIBDataSets you can set parameters for displaying fields, save and restore sorting, etc. You can get the same results by assigning one handler to several TpFIBDataSets. But it is more convenient to use TDataSetContainer, because you can place it on a separate TDataModule as well as its handlers and thus keep the application code outside visual forms.

Besides you can use TDataSetContainer to set one function for local sorting of all connected TpFIBDataSets.

```
TKindDataSetEvent = (deBeforeOpen, deAfterOpen, deBeforeClose,
  deAfterClose, deBeforeInsert, deAfterInsert, deBeforeEdit, deAfterEdit,
  deBeforePost, deAfterPost, deBeforeCancel, deAfterCancel, deBeforeDelete,
  deAfterDelete, deBeforeScroll, deAfterScroll, deOnNewRecord, deOnCalcFields,
  deBeforeRefresh, deAfterRefresh)
```

In FIBPlus 6.4.2 upwards, the container can be global for all TpFIBDataSet instances. For this you need to set the IsGlobal property to True.

You can also build container chains by using the MasterContainer property. When you “inherit” MasterContainer behaviour you expand container behaviour.

Additional actions for TpFIBUpdateObject data modifying

Besides using standard TpFIBDataSet modifying queries you can set various additional actions by using TpFIBUpdateObject objects. TpFIBUpdateObject is a TpFIBQuery descendant, a “client trigger” which sets additional actions for TpFIBDataSet before or after Insert/Modify/Delete operations. Read more details about TpFIBQuery descendant properties and methods in the Supplement.

I.e. you have a master –detail link master(id, name) and detail(id, name, master_id) and you want to automatically delete all dependant data from detail table on deleting records from master. You should add the pFIBUpdateObject object. Then set the property SQL 'delete from deatail where master_id = :id'; set the DataSet property to the dataset for the master table; set KindUpdate to kuDelete and make it execute before the main DataSet operator ExecuteOrder oeBeforeDefault.

Now an operator from the linked pFIBUpdateObject will be executed before record deleting from MasterDataSet.

Working with shared transactions

As we have already mentioned in “Work with transactions”, you should make Update transactions as short as possible. TpFIBDataSet is unique as it can work in the context of two transactions: Transaction and UpdateTransaction. We recommend you to use this method as it is the most correct for work with InterBase/Firebird. Data are read-only in a long-running transaction, whereas all modifying queries are executed in a short-running transaction.

The reading transaction (Transaction), as a rule, is ReadCommited and read-only (in order not to retain record versions). We recommend the following parameters for Transaction: read, nowait, rec_version, read_committed. The updating transaction (UpdateTransaction) is short and concurrent, its recommended parameters are: write, nowait, concurency. If you set AutoCommit = True, each change of TpFIBDataSet will be written to a database and become available to other users.

Note: use the shared transaction mechanism and AutoCommit very carefully, especially in master-detail links. In order not to get errors you should know well when the transactions start and close.

Batch processing

TpFIBDataSet has several methods for batch processing: BatchRecordToQuery and BatchAllRecordToQuery, which execute an SQL query. This query is set in TpFIBQuery, which is transferred as a parameter.

See DatasetBatching example to get to know how to use these methods.

Centralized error handling – TpFIBErrorHandler

FIBPlus provides developers with a mechanism of centralized handling of errors and exceptions which occur with FIBPlus components. For this use the TpFIBErrorHandler component with the OnFIBErrorEvent event:

```
TOnFIBErrorEvent = procedure(Sender: TObject; ErrorValue: EFIBError;
  KindIBError: TKindIBError; var DoRaise: boolean) of object;
```

```
где
TKindIBError = (keNoError, keException, keForeignKey, keLostConnect,
    keSecurity, keCheck, keUniqueViolation, keOther);

EFIBError = class(EDatabaseError)
  //..
  property SQLCode : Long read FSQLCode ;
  property IBErrorCode: Long read FIBErrorCode ;
  property SQLMessage : string read FSQLMessage;
  property IBMessage : string read FIBMessage;
end;
```

The option DoRaise manages FIBPlus behaviour after the handler execution, i.e. it shows whether a standard exception will be generated.

This option can be used for standard handling of different error types described in KindIBError.

FIBPlus versions 6.4.2 upwards have new properties in TpFIBErrorHandler. They enable developers to work with localized server messages correctly. You should set such string properties as Index, Constraint, Exception and At.

Getting TFIBSibEventAlerter events

Use the TFIBSibEventAlerter component to get database events. Define the Database property to show which connection events will be monitored. Then define necessary event names in Events and set Active to True to activate the component.

On getting the component event the OnEventAlert event handler will be executed. It is declared as:

```
procedure (Sender: TObject; EventName: String; EventCount: Integer);
```

where EventName is an event name, EventCount is a number of events executed.

Remember that events will be sent only on committing the transaction in which context it occurred. So several events can occur before ObEventAlert.

The Events example demonstrates how to use events in FIBPlus.

Debugging FIBPlus applications

FIBPlus provides developers with a powerful mechanism for SQL control and debugging. You can use the SQL monitor and a feature for SQL statistics gathering while working with the application.

Monitoring SQL queries

TFIBSQLMonitor is responsible for SQL query monitoring. To use it you should define any information type in TraceFlags and write the OnSQL event handler. This feature is demonstrated in the SQLMonitor demo example.

Registering executable queries

The TFIBSQLLogger component logs SQL query execution and keeps SQL query execution statistics.

Properties:

property	ActiveStatistics:boolean	- shows whether the statistics is active
property	ActiveLogging:boolean	- shows whether the logging is active
property	LogFileName:string	- defines the logging file
property	StatisticsParams :TFIBStatisticsParams	- defines statistics parameters

property LogFlags: TLogFlags - defines which operations are logged

property ForceSaveLog:boolean – defines whether to save a log after every query execution

Methods

function ExistStatisticsTable:boolean; - checks whether the statistics table exists

procedure CreateStatisticsTable; - creates a statistics table

procedure SaveStatisticsToDB(ForMaxExecTime:integer=0); - saves the statistics into a table. This parameter shows statistics of certain queries which we are interested in (you should set the minimal query execution time). Statistics will be saved for queries executed longer than ForMaxExecTime or equal to ForMaxExecTime.

procedure SaveLog; - saves a log into a file (valid if ForceSaveLog is set to True).

The SQLLogger example shows how to work with the TFIBSQLLogger component.

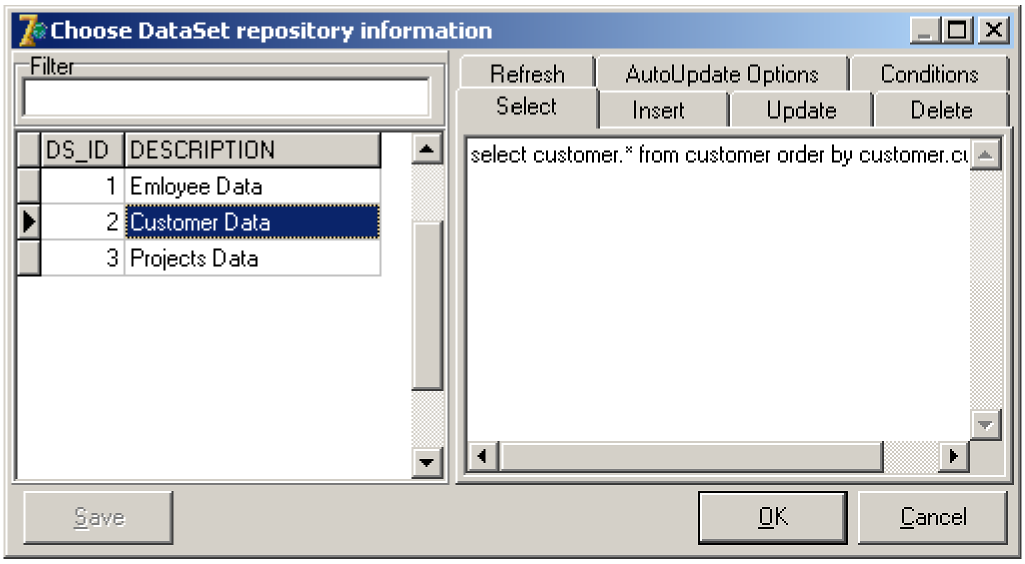
FIBPlus repositories

FIBPlus has three repositories enabling developers to save and use TpFIBDataSet settings, TFields settings and error messages. To use FIBPlus repositories you should set the UseRepositories property (urFieldsInfo, urDataSetInfo, urErrorMessagesInfo) in TpFIBDatabase (defining which repositories you need).

You can learn how to use all repositories from the DataSetRepository, ErrorMessagesRepository and FieldsRepository examples.

Dataset repository

To use the dataset repository in the TpFIBDataSet context menu use «Save to DataSets Repository table» and «Choose from dataSets Repository table». «Save to DataSets Repository table» menu item enables you to save main TpFIBDataSet properties into a special table FIB$DATASETS_INFO, whereas «Choose from dataSets Repository table» helps to load properties from the component already saved in the repository. If the table does not exist in the database, you will be asked to create it. To save the TpFIBDataSet properties into a database you should set a DataSet_ID value (not equal to zero).



DS_ID	DESCRIPTION
1	Emloyee Data
2	Customer Data
3	Projects Data

```
select customer.* from customer order by customer.cu
```

Picture 5. DataSets repository dialog

Then you should set DataSet_ID when running the application and the TpFIBDataSet properties will be loaded from the database table.

As you see in picture 5, FIBPlus saves main SQL queries as well as Conditions and AutoUpdateOptions properties.

Repository mechanism provides you with better application flexibility and enables you to change the program without recompiling it. You should only replicate repository tables.

Field repository

Get access to the field repository by opening the TpFIBDatabase context menu: «Edit field information table».

You can set such properties as DisplayLabel, DisplayWidth, Visible, DisplayFormat and EditFormat for any table field, view and Select procedure. The TRIGGERED value helps to select fields which will be filled in the trigger and which do not require user values even if they are Required (NOT NULL).

You should also set the psApplyRepository parameter in TpFIBDataSet.PrepareOptions in order to get TField settings from the repository on opening the query.

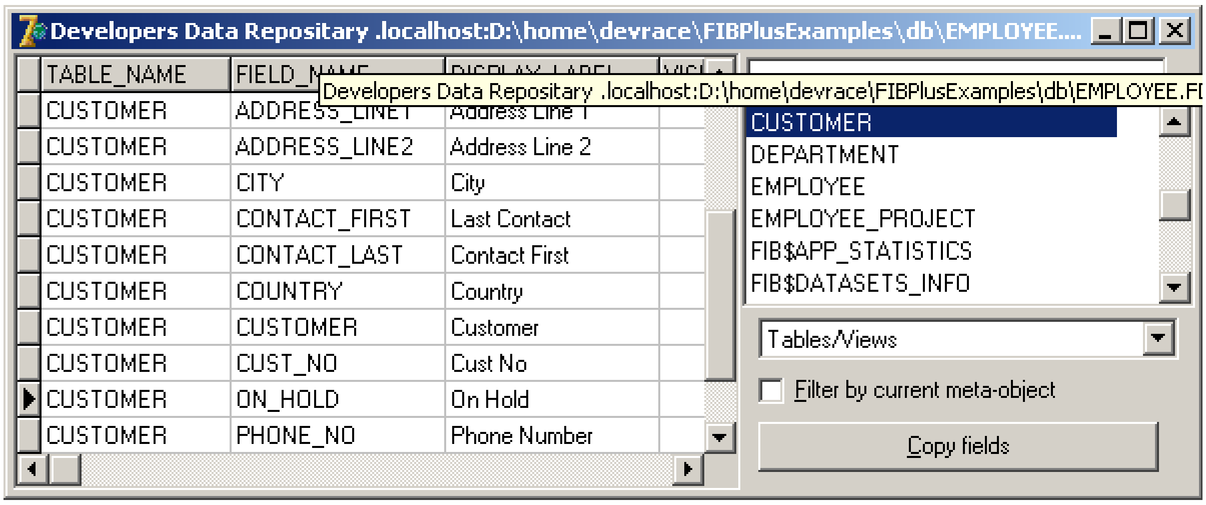
Using field aliases in SQL queries you can see that the settings are not applied for them. This is correct because physical tables do not have aliases. Nevertheless FIBPlus field repository enables you to have settings for such fields with aliases: write ALIAS instead of the table name in the repository.

In version FIBPlus 6.4.2 upwards the dataset has a standard event: TonApplyFieldRepository=**procedure**(DataSet:TDataSet;Field:TField;FieldInfo:TpFIBFieldInfo) **of object**;

It enables developers to use their own settings in the field repository. For example if you want to define the EditMask property, add the EDIT_MASK field to the repository table, define the container in the application and make it global. Then write the following code in the OnApplyFieldRepository event handler:

```
procedure TForm1.DataSetsContainer1ApplyFieldRepository(DataSet: TDataSet;
  Field: TField; FieldInfo: TpFIBFieldInfo);
begin
 Field.EditMask:=FieldInfo.OtherInfo.Values['EDIT_MASK'];
```

end;


Developers Data Repositary .localhost:D:\home\devrace\FIBPlusExamples\db\EMPLOYEE....

TABLE_NAME	FIELD_NAME	DISPLAY_LABEL
CUSTOMER	ADDRESS_LINE1	Address Line 1
CUSTOMER	ADDRESS_LINE2	Address Line 2
CUSTOMER	CITY	City
CUSTOMER	CONTACT_FIRST	Last Contact
CUSTOMER	CONTACT_LAST	Contact First
CUSTOMER	COUNTRY	Country
CUSTOMER	CUSTOMER	Customer
CUSTOMER	CUST_NO	Cust No
CUSTOMER	ON_HOLD	On Hold
CUSTOMER	PHONE_NO	Phone Number

Developers Data Repositary .localhost:D:\home\devrace\FIBPlusExamples\db\EMPLOYEE.FI

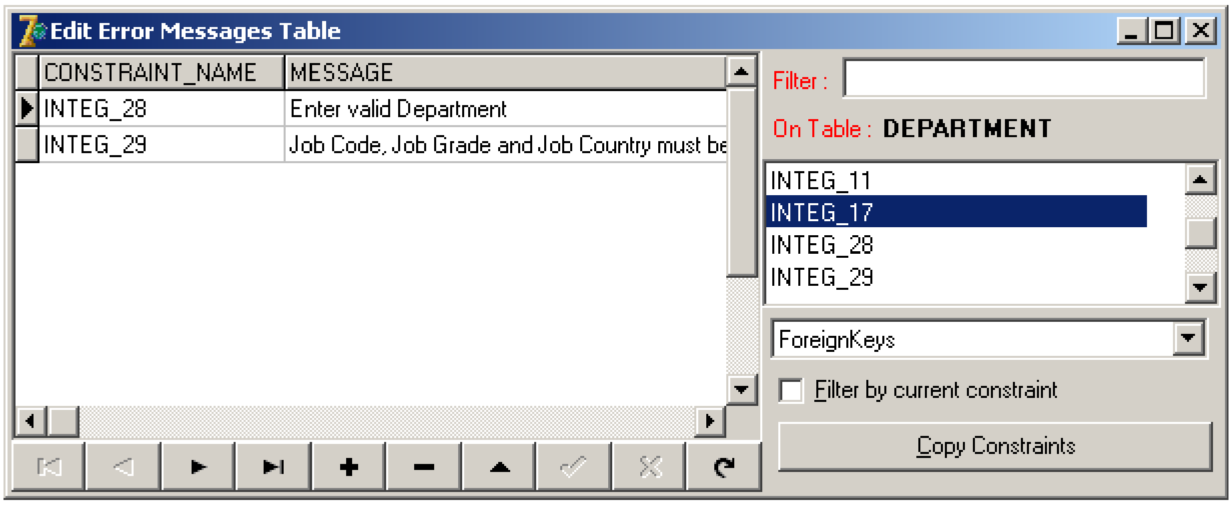
CUSTOMER
DEPARTMENT
EMPLOYEE
EMPLOYEE_PROJECT
FIB$APP_STATISTICS
FIB$DATASETS_INFO

Picture 6. Field repository

Error message repository

To get access to FIBPlus error message repository use the context menu of the TpFIBDatabase: «Edit error messages table». Using it you can create your own error messages for such cases as primary key (PK) violation, constraints (unique, FK, checks) and unique indices.

To use the error message repository place the TpFIBErrorHandler component on a form or project module and activate the repository. All texts of errors in the repository will be automatically replaced by your own error messages.


Edit Error Messages Table

CONSTRAINT_NAME	MESSAGE
INTEG_28	Enter valid Department
INTEG_29	Job Code, Job Grade and Job Country must be

Filter :
On Table : **DEPARTMENT**

INTEG_11
INTEG_17
INTEG_28
INTEG_29

Picture 7. Error repository

Supporting Firebird 2.0

FIBPlus is compatible with Firebird 2.0, it supports all Firebird 2.0 features.

Now you ca use Execute block statements in SelectSQL.

Now the *poAskRecordCount* option correctly works in all cases because it uses the statement “select count from” (where “select” is your original selection).

The TpFIBDatabase component has two new methods supporting new Firebird 2 commands: RDB$GET_CONTEXT and RDB$SET_CONTEXT:

```
function GetContextVariable (ContextSpace: TFBContextSpace; const VarName:
string): Variant;
```

```
procedure SetContextVariable (ContextSpace: TFBContextSpace; const VarName,
VarValue: string);
```

FIBPlus also supports FB2.0 insert ... into ... returning. Now you should not bother about getting generator values from the client but leave them in the trigger. You can also use RDB$DB_KEY. New possible variants of work with insert returning and RDB$DB_KEY are shown in the example "FB2InsertReturning".

If you have queries joining a table with itself:

```
Select * from Table1 t, Table1 t1
where ....
```

using Firebird 2.0 FIBPlus can understand correctly whether each field was taken from t or t1. This feature helps to generate FIBPlus modifying queries.

Additional features

Full UNICODE_FSS support

FIBPlus version 6.0 upwards correctly works with CHARSET UNICODE_FSS. For this you should use visual components supporting Unicode, e.g. TntControls.

To support Unicode FIBPlus has five new field types:

TFIBWideStringField = class(TWideStringField) – for work with VARCHAR,CHAR;
TFIBMemoField = class(TMemoField, IWideStringField) – for BLOB fields where IWideStringField is an interface for visual TNT components (http://tnt.ccci.org/delphi_unicode_controls)

The psSupportUnicodeBlobs option in TpFIBDataSet.PrepareOptions enables you to work with UNICODE_FSS BLOB fields. By default it is not active, because you need to execute additional query to know a charset for a certain BLOB field. If you do not work with UNICODE, the query will be unnecessary.

TpFIBDatabase has a new method function IsUnicodeCharSet: Boolean, which returns True if the connection uses UNICODE_FSS.

The FIBXSQLVAR class has a new property AsWideString: WideString which returns WideString.

NO_GUI compilation option

FIBPlus 6.0 upwards has a new compilation option: NO_GUI. Using it the library does not refer to standard modules with visual components. This helps you to write applications for system (not user) tasks. See {$DEFINE NO_GUI} in FIBPlus.inc.

Using SynEdit in editors

If you have installed SynEdit components, you can compile editor packages with SynEdit. Then SQL editor will have SQL operator syntax highlighting and the CodeComplete tool. See the define {$DEFINE USE_SYNEDIT} in pFIBPropEd.inc.

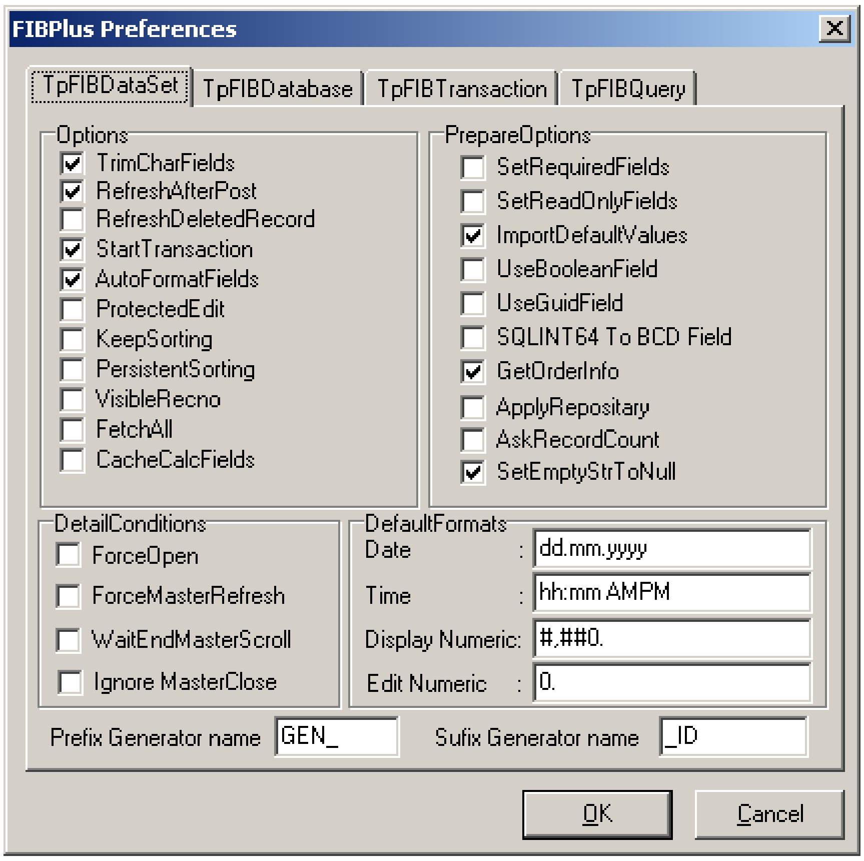
Important: you cannot change compilation defines using the trial FIBPlus version.

Unique tools: FIBPlusTools

Besides components FIBPlus has additional tools FIBPlus Tools, which extend IDE features and enable you to use FIBPlus at design-time more effectively.

Preferences

Preferences help you to set default parameters of the main components. On the first dialog page you can set default values for Options, PrepareOptions and DetailsConditions for all TpFIBDataSet components. You can also set certain keys for these properties. For example if you make SetRequiredFields active, then placing a new TpFIBDataSet component onto the form, you will see that its PrepareOptions property will have pfSetRequiredFields defined. It is important that default settings in FIBPlus Tools Preferences are valid for all new applications that you create. Notice that this concerns only initial defaults: if you change component properties after dropping the component onto the form, Preferences won’t have these changes. Besides when changing Preferences, you do not change components which properties were already defined.


FIBPlus Preferences

TpFIBDataSet | TpFIBDatabase | TpFIBTransaction | TpFIBQuery

Options
- [✓] TrimCharFields
- [✓] RefreshAfterPost
- [ ] RefreshDeletedRecord
- [✓] StartTransaction
- [✓] AutoFormatFields
- [ ] ProtectedEdit
- [ ] KeepSorting
- [ ] PersistentSorting
- [ ] VisibleRecno
- [ ] FetchAll
- [ ] CacheCalcFields

PrepareOptions
- [ ] SetRequiredFields
- [ ] SetReadOnlyFields
- [✓] ImportDefaultValues
- [ ] UseBooleanField
- [ ] UseGuidField
- [ ] SQLINT64 To BCD Field
- [✓] GetOrderInfo
- [ ] ApplyRepositary
- [ ] AskRecordCount
- [✓] SetEmptyStrToNull

DetailConditions
- [ ] ForceOpen
- [ ] ForceMasterRefresh
- [ ] WaitEndMasterScroll
- [ ] Ignore MasterClose

DefaultFormats

Date :	dd.mm.yyyy
Time :	hh:mm AMPM
Display Numeric:	#,##0.
Edit Numeric :	0.

Prefix Generator name: GEN_ Sufix Generator name: _ID

OK Cancel

Picture 8. Tools Preferences.

Prefix Generator name and Suffix Generator name fields are important. Setting them you can form generator names in the AutoUpdateOptions property in TpFIBDataSet. The generator name in AutoUpdateOptions is formed from the table name (UpdateTable), its prefix and suffix.

The following dialog pages help to define main properties for TpFIBDataBase, TpFIBTransaction and TpFIBQuery. In particular if you always work with newest InterBase/Firebird versions (e.g InterBase version 6 upwards), set SQL Dialect=3 in TpFIBDatabase, then you won't need to set it manually all the time.

SQL Navigator

It's the most interesting part of FIBPlus Tools which has no analogues in other products. This tool helps to handle SQL centrally for the whole application.

SQL Navigator enables you to have access to SQL properties of any component from one place.

The button «Scan all forms of active project» scans all application forms and selects those which contain FIBPlus components for work with SQL: TpFIBDataSet, TpFIBQuery, TpFIBUpdateObject and TpFIBStoredProc. Select any of these forms. The list in the right will be filled with components found on this form. Of you click any component you will see their corresponding properties with the SQL code. For TpFIBDataSet there will be shown such properties as SelectSQL, InsertSQL, UpdateSQL, DeleteSQL and RefreshSQL. For TpFIBQuery, TpFIBUpdateObject and TpFIBStoredProc FIBPlus will show an SQL property value.

You can change any property directly from SQLNavigator and its new value will be saved. SQLNavigator helps to work with groups of components. You only need e select corresponding components or forms.

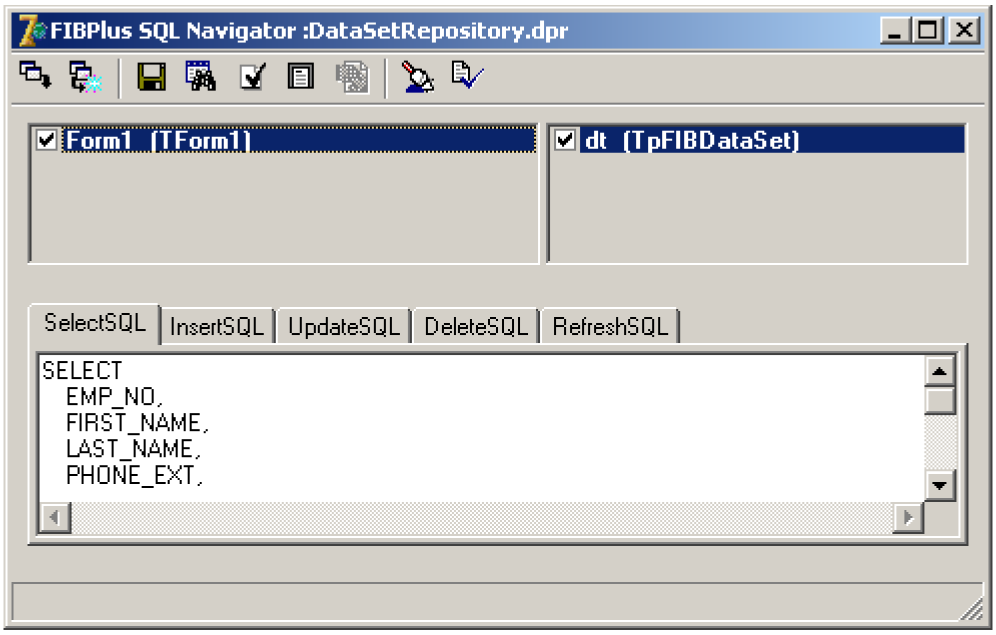
"Save selected SQLs" saves values of selected properties into an external file.

"Check selected SQLs" checks selected SQL queries on correctness in SQLNavigator.

Then you can analyze the file with selected queries by using special tools.

You can also use SQLNavigator for text searching in the SQL of the whole project.

By double clicking any found item you make SQLNavigator select a component and a property and thus can edit SQL.



```
SELECT
  EMP_NO,
  FIRST_NAME,
  LAST_NAME,
  PHONE_EXT,
```

Picture 9. Tools SQL Navigator

How to work with services

Besides the main component palette FIBPlus also has FIBPlus Services. These components are intended for getting information about InterBase/Firebird servers, user control, database parameter setting and database administration.

More details about each issue you can read in InterBase manuals (OpGuide.pdf, parts «Database Security», «Database Configuration and Maintenance», «Database Backup and Restore» and «Database and Server Statistic»).

The Services demonstration example shows how to work with services. Besides you can see a good example in Delphi/C++BuilderL: DelphiX\Demos\Db\IBX\Admin\. Though being written for IBX, this example suits well for FIBPlus.

All services components are TpFIBCustomService descendants and have such properties and methods as: ServerName – a server name used for connection; Protocol – a connection protocol; UserName – a user name; and Password – a user password.

In general work with services is the following: at first you define connection properties (server, protocol, user and password). The you connect to the server (Attach := True), execute necessary operations and disconnect (Attach := False). This sequence of actions is demonstrated in the code below and is necessary for any service.

```
with TpFIBXXXService.Create(nil) do
try
  ServerName := <server_name>;
  Protocol := <net_protocol>;
  UserName := <user_name>;
  Password := <password>;
  Atach;
  try
    //executed operations
  finally
    Deatach;
  end;
finally
  Free;
end;
```

Getting server information

Using the TpFIBServerProperties component you can get information about licenses, server configuration, a number of users connected to the database and a number of connected databases. For more details about properties, events and methods please see the DevGuide Supplement. In general you should connect to the server, define necessary information and get it using the Fetch method.

Parameters and results are described in the module IB_Services.pas. The following record keeps information about the number of server connections, the number of active databases operated by the server and their names:

```
  TDatabaseInfo = record
    NoOfAttachments: Integer;
    NoOfDatabases: Integer;
    DbName: Variant;
  end;
```

These two records have information about InterBase server licenses:

```
  TLicenseInfo = record
```

```
    Key: Variant;
    Id:     Variant;
    Desc:    Variant;
    LicensedUsers: Integer;
  end;

  TLicenseMaskInfo = record
    LicenseMask: Integer;
    CapabilityMask: Integer;
  end;

  TConfigFileData = record
    ConfigFileValue:Variant;
    ConfigFileKey:Variant;
  end;

  TConfigParams = record
    ConfigFileData: TConfigFileData;
    BaseLocation: string;
    LockFileLocation: string;
    MessageFileLocation: string;
    SecurityDatabaseLocation: string;
  end;
```

This record shows information about the server version:

```
  TVersionInfo = record
    ServerVersion: String;
    ServerImplementation: string;
    ServiceVersion: Integer;
  end;

  TPropertyOption = (Database, License, LicenseMask, ConfigParameters, Version);
  TPropertyOptions = set of TPropertyOption;

  TpFIBServerProperties = class(TpFIBCustomService)
    procedure Fetch;
    procedure FetchDatabaseInfo;
    procedure FetchLicenseInfo;
    procedure FetchLicenseMaskInfo;
    procedure FetchConfigParams;
    procedure FetchVersionInfo;
    property DatabaseInfo: TDatabaseInfo
    property LicenseInfo: TLicenseInfo
    property LicenseMaskInfo: TLicenseMaskInfo
    property ConfigParams: TConfigParams
    property Options : TPropertyOptions
  end;
```

The TpFIBLogService component is used for server logging.

Most services return text information using the OnTextNotify event of the TserviceGetTextNotify type. This type is described in IB_Services as:

```
  TServiceGetTextNotify = procedure (Sender: TObject; const Text: string) of
object;
```

TpFIBLogService uses this type of notification.

When working with these services you need to set the reaction on the OnTextNotify event (which starts the service and reads the information). The information is read as:

```
    ServiceStart;
    while not Eof do
      GetNextLine;
```

So the complete code will be the following:

```
with TpFIBXXXService.Create(nil) do
try
  ServerName := <server_name>;
  Protocol := <net_protocol>;
  UserName := <user_name>;
  Password := <password>;
  Atach;
  try
    ServiceStart;
    while not Eof do
      GetNextLine;
  finally
    Deatach;
  end;
finally
  Free;
end;
```

On working with some other similar services you should set different options.

For more details check the Services example.

Managing server users

TpFIBSecurityService is responsible for work with users. It has methods which enable you to get user info, as well as to add, modify and delete users. If you use this component you can get the following information:

A record with user information at the server:

```
  TUserInfo = class
  public
    UserName: string;
    FirstName: string;
    MiddleName: string;
    LastName: string;
    GroupID: Integer;
    UserID: Integer;
  end;
```

Available properties and methods of the component:

```
  TpFIBSecurityService = class(TpFIBControlAndQueryService)
    procedure DisplayUsers;
    procedure DisplayUser(UserName: string);
    procedure AddUser;
    procedure DeleteUser;
    procedure ModifyUser;
    procedure ClearParams;
    property  UserInfo[Index: Integer]: TUserInfo read GetUserInfo;
    property  UserInfoCount: Integer read GetUserInfoCount;

    property SQlRole : string read FSQLRole write FSQLrole;
    property UserName : string read FUserName write FUserName;
    property FirstName : string read FFirstName write SetFirstName;
    property MiddleName : string read FMiddleName write SetMiddleName;
    property LastName : string read FLastName write SetLastName;
    property UserID : Integer read FUserID write SetUserID;
    property GroupID : Integer read FGroupID write SetGroupID;
    property Password : string read FPassword write setPassword;
  end;
```

Use the DisplayUser method to get information about the server user (using the user name as a parameter). DisplayUsers is used to get information about all users. After calling DisplayUser or DisplayUsers you will see the number of users in the UserInfoCount property; the UserInfo index property will return a user record by its index.

To add a new user you should set the following properties: UserName, FirstName, MiddleName, LasName, Password; and execute the AddUser method. DeleteUser and ModifyUser methods work in the same way as AddUser; they require UserName as a minimal obligatory parameter.

Note:

- the Password property is not returned by DisplayUser and DisplayUsers methods.
- The SQLRole property cannot be used to associate a role with a user.

For these operations you should execute queries using TpFIBQuery (GRANT/REVOKE).

The Services example demonstrates all aspects of work with server users.

Configuring databases

TpFIBConfigService helps to set such parameters as:

- set an interval for the automatic garbage collection in the transaction (Sweep Interval);
- set a mode of writing changes on the disk (Async Mode);
- set the DB page size;
- set the reserved space for the DB;
- set the read-only mode;
- activate and deactivate the shadow

It also helps to:

- shutdown the database (shutdown)
- start the database (online).

Before working with this service you should set the property

```
    property DatabaseName: string
```

A path to the DB at the server

To set the interval for the automatic garbage collection in the transaction you should use the `SetSweepInterval` method. The only `SetSweepInterval` parameter is the interval, and by default it is equal to 20000.

```
  procedure SetSweepInterval (Value: Integer);
```

To set the database dialect you should use the SetDBSqlDialect method; its parameter is the DB dialect. Only three parameters are supported: 1, 2, 3.

```
  procedure SetDBSqlDialect (Value: Integer);
```

To set PageBuffers use the SetPageBuffers method; its parameter is the required buffer size.

```
  procedure SetPageBuffers (Value: Integer);
```

To activate the shadow use the ActivateShadow method. It doesn't require any parameters.

```
procedure ActivateShadow;
```

To set the asynchronous writing mode you should set the SetAsyncMode method to True; to deactivate this mode – to False.

```
procedure SetAsyncMode (Value: Boolean);
```

To set the read-only mode use the SetReadOnly method. If it is set to True, the read-only mode is active; if it is set to False, the read-only mode is deactivated. This mode is used when you prepare a database for distribution on CD or other mediums

```
procedure SetReadOnly (Value: Boolean);
```

Use SetReserveSpace to set the reserved space for the DB:

```
procedure SetReserveSpace (Value: Boolean);
```

Use the ShutdownDatabase method to shutdown the database. There are three types of shutting the database down: forced, denying new transactions and denying new connections. All the above mentioned operations should be better done under the SYSDBA administrator's connection.

```
procedure ShutdownDatabase (Options: TShutdownMode; Wait: Integer);

TShutdownMode = (Forced, DenyTransaction, DenyAttachment);
```

Use the BringDatabaseOnline method to bring the database online.

```
procedure BringDatabaseOnline;
```

Backing up and restoring databases

TpFIBBackupService and TpFIBRestoreService components help to use database backup and restore functions.

TpFIBValidationService enables you to gather garbage, check the database on errors and repair it if necessary.

Options play a very important role when you backup, restore and check the database. All these options are described in the server manual OpGuide.pdf.

Backup and restore services are simple, and you can work with them in the same way as FIBLogService. The only peculiar thing is the interpretation of their work results:

If you need to be sure there were no errors during the DB backup/restore, you should see the operation log. In case of errors the log will have a string «GBAK: ERROR»

If TpFIBValidationService did not find any errors, its log will contain one empty string.

If the database repairing caused errors (the Mend TpFIBValidationService option), they will be written to the server log.

Remember that TpFIBBackupService creates the backup file at the server; the server does the backup of the DB and keeps the file there. But you can do the backup from another server as well. Write a full path to the database in the connection path:

```
with TpFIBXXXService.Create(nil) do
try
  ServerName := <local_backup_server_name>;
  Protocol := <net_protocol>;
  UserName := <user_name>;
  Password := <password>;
  DatabaseName := <remote_db_name>;
  BackupFile.Add(<local_backup_name>);
  Atach;
  try
    ServiceStart;
    while not Eof do
      GetNextLine;
  finally
    Deatach;
  end;
finally
  Free;
end;
```

These services can work with Embedded server too (remember to set the Local protocol when working with Firebird Embedded Server).

Getting statistics about database

Use the TpFIBStatisticalService component to get important statistics about working database. It is similar to TpFIBLogService.