Microsoft Business Solutions

C/FRONT Reference Guide

MICROSOFT BUSINESS SOLUTIONS–NAVISION

C/FRONT REFERENCE GUIDE

Published by Microsoft Business Solutions ApS, Denmark.

Published in Denmark 2003.

DocID: NA-370-DVG-003-v01.00-W1W1

PREFACE

This book is a manual for the C/FRONT application programming interface that can be used to access a C/SIDE® database. This manual describes how to install and use C/FRONT. This book is part of a comprehensive set of documentation and Help materials for Microsoft® Business Solutions–Navision®.

You should also be familiar with the symbols and typographical conventions used in the Navision manuals. In the list below, you can see how various elements of the program are distinguished by special typefaces and symbols:

Appearance	Element
Ctrl	Keys on the keyboard. They are written in small capitals.
Design	Menu items and buttons in windows. They always start with a capital letter, and the access key is underlined.
Address	Field names. They appear in medium bold and start with a capital letter.
Department	Names of windows, boxes and tabs. They appear in medium bold italics and start with a capital letter.
Hansen	Text that you must enter, for example: "...enter Yes in this field." It is written in italics.
`fin.flf`	File names. They are written with the Courier font and lowercase letters.
⬆ ⬇ ▼ ✳ ▸ ...	The special symbols that can be seen in the windows on the screen.

TABLE OF CONTENTS

Table of Contents

Chapter 1
Introduction to C/FRONT

This is an introduction to C/FRONT and describes the contents of C/FRONT, the system requirements, standby/hibernation and multilanguage.

This chapter contains:

- Introduction to C/FRONT

1.1 Introduction to C/FRONT

C/FRONT is an application programming interface that can be used to access a C/SIDE® database. C/FRONT facilitates high-level interaction with the C/SIDE database manager, and allows C developers to manipulate any C/SIDE database.

The central component of C/FRONT is a library of C functions. These functions give you access to every aspect of data storage and maintenance, and allow you to integrate both standard and custom applications with your C/SIDE database.

This Manual

This manual explains how to use the library functions to access the features and facilities of a C/SIDE database. It consists of five chapters:

- **Chapter 1 – Introduction to C/FRONT**:

  Lists the contents of C/FRONT and explains how to configure C/FRONT.

- **Chapter 2 – Review of Standard Operations**:

  Explains and demonstrates how to use some of the library functions in frequent operations.

- **Chapter 3 – A Sample Application**:

  Explains how to run the sample application that comes with C/FRONT. The source code is available on the Microsoft® Business Solutions–Navision® product CD in the file called `sample.c`.

- **Chapter 4 – Moving From the C-Toolkit to C/FRONT**:

  Describes the differences between the C-Toolkit and C/FRONT and explains how to upgrade to C/FRONT.

- **Chapter 5** – **The Library Functions**:

  All of the C/FRONT functions are listed and described.

  Appendix A lists the type and constant definitions.

  Appendix B describes nonstandard data formats.

It is assumed that the reader has a good knowledge of C and C/SIDE. For further information about these subjects, refer to your C manuals and the *Application Designer's Guide*.

The Contents of C/FRONT

C/FRONT provides an interface to the C language, and is distributed on the Navision product CD. The following files are distributed:

File name	Used for
`cfront.dll`	C/FRONT C-API library

File name	Used for
cfrontsql.dll	C/FRONT C-API library
cf.h	C/FRONT header file
libload.c	Source file containing functions to load and unload cfront.dll
dberror.txt	Database error/return codes
cfront.ocx	C/FRONT OCX

In addition to these files, C/FRONT contains a sample application – see Building and Running the Sample Application on page 22 for details.

Before you can use C/FRONT, you must copy the following files from Navision to the directory where you have installed C/FRONT:

```
dbm.dll
nc_netb.dll
nc_tcp.dll
slave.exe
fin.etx
fin.stx
fin.flf
```

You must also have a database. These files are all part of a standard Navision installation. Last minute changes to C/FRONT are documented in the file `Readme.txt` that is distributed on the Navision product CD. Please read this file before beginning the installation.

Installation

The `Readme.txt` file on the product CD contains detailed and up-to-date information on installing C/FRONT.

System Requirements

C/FRONT can be used on the Windows XP, Windows 2000, Windows 98 and Windows NT platforms. It has been tested with the Watcom C compiler, version 10.5a and with Microsoft Visual C++, version 5.00. It can also be used with any other compiler that can load and use DLLs correctly, but note that the functions in the C/FRONT DLL module are called with the _CDECL calling convention.

Standby and Hibernation

C/FRONT supports the standby and hibernation facilities provided by Windows XP and Windows 2000.

Putting your computer on standby means that the entire computer switches to a low power state. When on standby all devices, such as the monitor and hard disks, turn off and your computer uses less power. When you want to use the computer again, it comes out of standby quickly, and your desktop is restored exactly as you left it.

Standby is particularly useful for conserving battery power in portable computers. Because standby does not save your desktop state to disk, a power failure while on standby can cause unsaved information to be lost.

Putting your computer in hibernation means that before shutting down your computer saves everything that is currently in memory to disk, turns off your monitor and hard disk, and then turns off your computer. When you reactivate your computer, your desktop is restored exactly as you left it. It takes longer to bring your computer out of hibernation than out of standby.

Shutting Down

The individual workstations can specify that their computer should go to standby or hibernate after being idle for a certain length of time. It is also possible to make the computer go to standby from the Shut Down dialog box in Windows.

Windows will not go to standby or hibernation if there is an open server connection from C/FRONT.

If you attempt to make the computer go to standby from the Windows Shut Down dialog box, a window will appear informing you that C/FRONT is busy and that shutting down is not yet possible.

If you click Cancel in this window, the hibernation or standby procedure will be postponed. Alternatively, you can ignore this window and the computer will shut down when C/FRONT has completed its task.

Restarting

When you restart your computer after it has gone to standby or is in hibernation it will restart with the desktop exactly as it was when you left it. However, the information displayed will also be the same and will therefore not necessarily be up to date.

The window will not be updated until you use the program and actively update the window in question.

Multilanguage

Navision 3.70 is multilanguage enabled, allowing users to change application language on the fly. This is achieved by adding captions to the objects in the database. These captions contain the names of the database objects in the languages that are available in your application. However, C/FRONT does not have any table or field caption functions that you can use to identify these different names.

Any programs that use C/FRONT will therefore not be able to identify the names of the database objects in the various application languages that are available. To obtain this information you can generate a text file listing all of the objects in the database and the captions that they contain.

To generate this text file:

1 Open the Object Designer and select all the objects.

2 Click Tools, Translate, Export.

This file will list all the objects in the database and each object will be listed once for each language that it used in your application.

Here is an example:

T3-F2-P8629-A1033-L999:Due Date Calculation

Table 3, Field 2, Property 8629(Caption), Language ID 1033 (US English), Max. Length 999, the name of the object.

You can now use the information contained in this text file in the program that uses C/FRONT.

Chapter 2
Review of Standard Operations

C/FRONT contains functions that allow you to perform all the standard operations that are used to maintain a C/SIDE database.

The chapter contains the following topics:

- The Standard Operations

2.1 The Standard Operations

Maintaining a C/SIDE database involves a number of operations:

- Determining which DLL to Use
- Initializing the Library
- Connecting to a Server and Opening a Database
- Opening a Company
- Opening a Table
- Using Filters
- Using Keys
- Finding a Record
- Inserting a Record
- Modifying a Record
- Deleting a Record
- Editing a Field in a Record
- Handling Errors and Exceptions

These operations are illustrated with the help of small sample routines. All of the functions are fully explained in chapter 5, "The Library Functions".

Determining which DLL to Use

Navision can run on two different servers and C/FRONT therefore comes with two different DLLs: `CFRONT.DLL` and `CFRONTSQL.DLL`. They will both be installed when you install C/FRONT.

If you are running on Navision Database Server you must use `CFRONT.DLL`. If you are running on Microsoft SQL Server you must use `CFRONTSQL.DLL`.

Initializing the Library

To initialize the library and configure the environment, call DBL_Init. This function initializes internal data structures, creates buffers, and loads the dynamic link libraries required by the library.

As shown below, this operation is usually performed in the main( ) function of the C program:

```
void main(int argc, char* argv[])

{
DBL_Init();
/* ... */
DBL_Exit();
exit(0);
}
```

Connecting to a Server and Opening a Database

C/FRONT now contains a single function that you can use to connect with a server and open a database. This new function works with both Navision Database Server and the Microsoft SQL Server Option.

SQL Server Option

With the SQL Server Option there is one function that enables you to connect to a server, open a database and specify the kind of authentication to use.

```
#define DRIVERNAME "NDBCS"
#define SERVERNAME "SQL 1"
#define NETTYPE      "Named Pipes"
#define DATABASENAME "My Database.mdf"
#define CACHESIZE    0
#define USECOMMITCACHE    0
#define USENTAUTHENTICATION 1
#define USERID ""
#define PASSWORD ""

DBL_ConnectServerandOpenDatabase(DRIVERNAME, SERVERNAME, NETTYPE,
DATABASENAME, CACHESIZE, USECOMMITCACHE, USENTAUTHENTICATION, USERID,
PASSWORD);
/* ... access the database ... */


 DBL_DisconnectServer();

/* ... */
```

If you are using SQL Server you still have to enter a zero value for CacheSize or UseCommitCache even though they only apply to the Navision Database Server. If you select NT Authentication (`UseNTAuthentication=1`) then you do not have to supply a user ID or a password. However you must enter two empty sets of quotes ("") in order to comply with the syntax.

You must run this function again if you want to open another database. This database can be on the same server or on another server.

Both Servers

This routine works with both servers and we recommend that it be used with both server options.

Navision Database Server

The functions described here still work with the Navision Database Server, even though we recommend that you use the function described above.

The library can either run in 'local mode' or be connected to a remote server in a network. When the user connects to a Navision Database Server in a network a database is opened automatically by the server. When the library is run in 'local mode' (the database is stored on the same computer as your application), a database must be opened explicitly. The following routine works in both environments:

```
#define DRIVERNAME "NDBCN"
#define SERVERNAME "accounting"
#define NETTYPE      "tcp"
#define DATABASENAME "Database.fdb"
#define CACHESIZE    1000
#define USECOMMITCACHE    0

DBL_BOOL RemoteMode = 0;

/* ... */

if (RemoteMode)
DBL_ConnectServer(DRIVERNAME, SERVERNAME, NETTYPE);
else
DBL_OpenDatabase(DATABASENAME, CACHESIZE, USECOMMITCACHE);

/* ... access the database ... */

if (RemoteMode)
 DBL_DisconnectServer();
else
 DBL_CloseDatabase();

/* ... */
```

If the open/connect operation fails, the function raises an exception that terminates the application.

Only one database/server connection can be open at a time.

For more information about exceptions, see Handling Errors and Exceptions on page 18.

Note

C/FRONT does not support the use of extended characters in directory and database names.

Opening a Company

A database consists of one or more companies. A company is a "subdatabase," whose primary use is to separate and group data within one database. A company "bundles" one or more tables together into a logical structure that is identified by the company name. The tables within a company do not need not have anything in common other than the shared company name.

Only one company can be open at a time. You must open a company before the application can access the data in the database tables. By opening a company, you specify which data tables can be opened by DBL_OpenTable.

```
#define COMPANYNAME "Test Company"
```

```
DBL_OpenCompany(COMPANYNAME);

/* ... open tables ... */

DBL_CloseCompany();
```

If the company that you want to open does not exist, the function raises an exception that terminates the application.

For more information about exceptions, see Handling Errors and Exceptions on page 18.

A database maintains a list of the companies that it contains. To retrieve the names of all the companies contained in a database, execute the DBL_NextCompany function in a loop:

```
DBL_U8 *CompName;

for (CompName = NULL; CompName = DBL_NextCompany(CompName);)
{
/* ... */
}
```

For more information about companies, see the *Application Designer's Guide*.

Opening a Table

A database contains a number of tables that can be manipulated with the functions in the library. You can have any number of tables, within a single company, open at the same time. A table is identified by a unique number. When a table is opened, the database manager returns a unique handle, which remains valid until the table is closed. This handle must be passed to all operations that are carried out on the table.

If you do not know the number of the table that you want to open, you can look it up with the library function DBL_TableNo. This function requires the name of the table as a parameter. Table names are also unique.

Manipulating a table often requires the use of filters and keys. Filters and keys are bound to a handle and not to a table. This allows you to open various "views" of a single table, each with its own key and set of filters, to suit the needs of individual applications.

The following example illustrates how a table is represented by its handle, with the symbolic name hTable.

```
#define TABLENAME "MyTable"

DBL_HTABLE hTable;
DBL_S32    TableNo;

TableNo = DBL_TableNo(TABLENAME);
```

```
if (TableNo != 0)
{
DBL_OpenTable(&hTable, TableNo);
/* ... access data in the table ... */
DBL_CloseTable(hTable);
}
```

Filters and keys are discussed in the next section. For more information about tables, see the *Application Designer's Guide*.

Using Filters

Filters limit the number of records that are being manipulated. Filters limit the number of records that are selected for calculating column sums, in record searches, and in other activities. The library contains two functions that can set a filter: DBL_SetFilter and DBL_SetRange. DBL_SetRange is a subfunction of DBL_SetFilter.

In the following example, the operations will only affect records with Field Number 2 that are within the ranges 100..200 and 300..400.

```
#define TABLENUMBER 15

DBL_HTABLE hTable;
DBL_S32    FieldNo;
/* ... */
DBL_OpenTable(&hTable, TABLENUMBER);
FieldNo = 2;
DBL_SetFilter(hTable, FieldNo, "100..200&300..400", NULL);
/* ... retrieve data within the specified range ... */
DBL_CloseTable(hTable);
/* ... */
```

Setting a Range

The DBL_SetRange function is used in the following way:

```
DBL_HTABLE  hTable;
DBL_S32     FieldNo  = 2;
DBL_S32     MinValue = 100;
DBL_S32     MaxValue = 200;

/* ... */
DBL_SetRange(hTable, FieldNo, &MinValue, &MaxValue);
/* ... retrieve data within the specified range ... */
/* ... */
```

DBL_SetFilter can use the & or | operators to specify a complex interval as the condition for a field, but DBL_SetRange can specify only a single interval. A filter can be retrieved by the function DBL_GetFilter, and the range by DBL_GetRange.

For more information about filters, see the *Application Designer's Guide*.

Using Keys

You can use keys to sort the records in a table according to the values in specified fields, for example, in ascending order. You can find a field that contains a specific value much faster when the records are sorted (using a key). A key maintains the relationship between the records, in a structure called an index.

A key is composed of one or more fields. You specify how these fields are ordered in C/SIDE. Keys cannot be defined or modified in C/FRONT.

A table can contain up to 20 keys, each with its own index. The first key in a table is the primary key and the data it contains must always be unique. The index for each key is maintained by the C/SIDE database manager.

All of the other keys are secondary keys. Multiple secondary keys may be active simultaneously. The secondary keys do not have to contain unique data. Records containing identical data in secondary key fields are "sub-sorted" once again, according to the value of the primary key because the primary key is always in effect with the current secondary key.

Keys can be active or inactive. Keys can be activated or deactivated in C/SIDE, not in C/FRONT. Only active keys are available to library functions.

When you insert, delete or modify a record, the indexes that are maintained by all of the active keys are automatically updated to reflect any changes that are made to the table. The indexes of inactive keys are not updated. If at some point you reactivate a key (in C/SIDE) that has been inactive, you must allow some time for the application to rebuild the index structure. This may require some additional disk space if the space that has been allocated to the database is nearly full, because each key occupies space in the database.

The Current Key

Although keys can only be defined in C/SIDE, the key needed for the current application can be selected in the library. This key is then called the current key. The current key is always attached to the table handle, not to the table itself.

The following example shows how to select the current key:

```
DBL_HTABLE hTable;
DBL_S32    Key[DBL_MaxFieldsPerKey+1];

/* ... */

Key[0] = 3;
Key[1] = 0; /* Zero-terminated list */
DBL_SetCurrentKey(hTable, Key);

/* ... scan table sorted by field number 3 ... */

/* ... */
```

The primary key is selected when you call DBL_SetCurrentKey with the Key parameter equal to NULL.

Each table maintains a list of all its keys. To retrieve this list, execute the DBL_NextKey function in a loop:

```
DBL_HTABLE hTable;
DBL_S32    *Key;
DBL_S32    *Field;

/* ... */

for (Key = NULL; Key = DBL_NextKey(hTable, Key); )
{
printf("Key contains the following field number(s):\n");

for (Field = Key; *Field; Field++)
printf("%d\n", *Field);
}

/* ... */
```

Other key functions are:

Function	Purpose
DBL_GetCurrentKey	Retrieves the key that is currently selected.
DBL_KeySumFields	Returns a list of SumIndexField® numbers for a specified key.

For more information about keys and SumIndexFields, see the *Application Designer's Guide*.

Finding a Record

Two functions are used to retrieve records:

Function	Purpose
`DBL_FindRec`	Locates a record that contains given values in the fields of the current key, and copies it to a buffer. The current filters are used when searching.
`DBL_NextRec`	Uses the current sorting sequence to retrieve a record in the table relative to a specified record, and copies it to a buffer. The current filters are used when searching.

To create a record buffer, call DBL_AllocRec. This function allocates an area of memory equal to the size of the record, including virtual fields (FlowField® and FlowFilter®). You must remove the record buffer when you are finished using it, by calling DBL_FreeRec.

In the following routine, DBL_FindRec searches for the first record in the table, copies it to a buffer and performs some operations on it. Control is then passed to DBL_NextRec, which steps through the entire file performing the same operations on each record.

```
DBL_HTABLE hTable;
DBL_HREC   hRec;

/* ... */

/* Allocate memory for the record buffer */
hRec = DBL_AllocRec(hTable);

/* Let DBL_FindRec return error if not present */
DBL_Allow(DBL_Err_RecordNotFound);

/* Find the first record */
if (DBL_FindRec(hTable, hRec, "-"))
{
do
{
/* ... */

/* Get the next record */
} while (DBL_NextRec(hTable, hRec, 1));
}
DBL_FreeRec(hRec);
/* ... */
```

For more information about records and FlowFields, see the *Application Designer's Guide*.

Inserting a Record

Inserting a record is a three-step operation:

1. Call DBL_InitRec to initialize the record. This function assigns the default field values that were defined when the record layout was created in C/SIDE.
2. Modify the field values.

   Be aware that the library does not provide range and validity checks. You must verify that the inserted data is valid.
3. After you have verified the contents of the fields, call DBL_InsertRec to insert the record into the table.

   The database manager will ensure that the new record is automatically inserted into the correct place. The correct place is determined by the values of the fields in the primary key.

In the following example, an initialized record is inserted into the table:

```
DBL_HTABLE hTable;
DBL_HREC   hRec;

/* ... */

/* Allocate memory for the record buffer */
hRec = DBL_AllocRec(hTable);

/* Initialize record */
DBL_InitRec(hTable, hRec);

/* ... assign values to fields in hRec ... */

DBL_BWT();
DBL_Allow(DBL_Err_RecordExists);
if (DBL_InsertRec(hTable, hRec))
{
printf("Record inserted.\n");
DBL_EWT();
}
else
{
  printf("Record NOT inserted. Record already exists.\n");
DBL_AWT();
}

/* ... */
```

The transaction functions DBL_BWT, DBL_AWT and DBL_EWT ensure that all the database updates that are grouped in a series are either committed (DBL_EWT) or rolled back (DBL_AWT).

Modifying a Record

Use the DBL_ModifyRec function to change the field values in a record. The record structure itself cannot be changed by using any of the library functions. The record structure can only be changed in C/SIDE.

Be aware that the library does not provide range and validity checks.You must verify that the data being inserted is valid.

The following example below shows how a modified record is copied from its buffer and written to the original record in the table. The fields are changed while the record is in the buffer, but this is not shown in this example. To see the steps in detail, refer to Editing a Field in a Record on page 18.

```
DBL_HTABLE hTable;
DBL_HREC   hRec;

/* ... */
```

```
/* Allocate memory for the record buffer */
hRec = DBL_AllocRec(hTable);

/* ... retrieve record to modify from the table ... */

/* ... assign values to fields in hRec ... */

DBL_BWT();
DBL_Allow(DBL_Err_RecordNotFound);
if (DBL_ModifyRec(hTable, hRec))
{
printf("Record modified\n");
DBL_EWT();
}
else
{
printf("Record NOT modified. Record not found\n");
DBL_AWT();
}

/* ... */
```

The transaction functions DBL_BWT, DBL_AWT and DBL_EWT ensure that all database updates grouped in a series are either committed (DBL_EWT) or rolled back (DBL_AWT).

Deleting a Record

Use DBL_DeleteRec to delete a record from a table. This function can remove any or all records in a table, but the table description itself can only be deleted in C/SIDE.

```
DBL_HTABLE hTable;
DBL_HREC   hRec;

/* ... */

/* Allocate memory for the record buffer */
hRec = DBL_AllocRec(hTable);

/* ... retrieve record to delete from the table ... */

DBL_BWT();
DBL_Allow(DBL_Err_RecordNotFound);
if (DBL_DeleteRec(hTable, hRec))
printf("Record deleted\n");
else
{
printf("Record NOT deleted. Record not found\n");
DBL_AWT();
}
DBL_EWT();

/* ... */
```

The transaction functions DBL_BWT, DBL_AWT and DBL_EWT ensure that all database updates that are grouped in a series are either committed (DBL_EWT) or rolled back (DBL_AWT).

Editing a Field in a Record

Before you can edit a record, you must retrieve it from the table and have it copied to a buffer, where you can access it by using a record handle. You can then use the DBL_AssignField function to assign new values to (edit) the fields in the record.

In the following example, DBL_FindRec is first used to retrieve the record and copy it to a buffer. Then it is accessed with the hRec handle, and a new value is assigned to field number 3.

```
DBL_HTABLE hTable;
DBL_HREC hRec;

/* ... */

DBL_FindRec(hTable,hRec,"=");
DBL_AssignField(hTable,hRec,3,DBL_FieldType(hTable,3),"RAW",
strlen("RAW"));

DBL_BWT();
DBL_ModifyRec(hTable,hRec);
DBL_EWT()

/* ... */
```

Handling Errors and Exceptions

Errors can occur if you use the library functions incorrectly. If an error occurs, the library terminates your application. However, you can specify that some errors are allowed by using the DBL_Allow function. For more information, see DBL_Allow() on page 42.

The library function DBL_Allow enables the programmer to allow the following errors:

- DBL_Err_TableNotFound
- DBL_Err_RecordNotFound
- DBL_Err_RecordExists
- DBL_Err_KeyNotFound

These allowable errors can occur in the following library functions:

- DBL_OpenTable
- DBL_FindRec
- DBL_InsertRec
- DBL_DeleteRec
- DBL_ModifyRec

- DBL_SetCurrentKey

If the error that causes the function to fail has been allowed by DBL_Allow, the function will return `0`. If it has not been allowed, the library will do the following:

1. Fetch the corresponding error message and call the message handler, passing the error message as a parameter. If no message handler has been set, the error message is written to standard output.
2. Call the exception handler.

The following routine demonstrates how to allow an unsuccessful result for a record retrieval:

```
/* ... */

/* Allow DBL_FindRec to fail */
DBL_Allow(DBL_Err_RecordNotFound);

if (DBL_FindRec(hTable, hRec, "="))
  printf("Record was found\n");
else
  printf("Record was NOT found\n");

/* ... */
```

When a library function detects an error or sends a warning, it fetches the corresponding error message and calls the default message handler, passing the message, the message type and the message code as arguments. The default message handler writes the error message to standard output.

However, you can choose to install your own message handler by using the library function DBL_SetMessageShowHandler. Your custom message handler can do such tasks as:

- hide particular messages
- map some messages to your own versions
- improve the formatting of the messages
- log messages to a file

Error messages passed to the message handler (and displayed by the default message handler) are retrieved from the `fin.etx` and `fin.stx` ASCII files. You can inspect the contents of these files with an ASCII editor.

After the library has detected an error and the message handler has been called, the default exception handler is called. The default exception handler simply terminates the application. You can install your own exception handler to prevent the application being terminated. Use the library function DBL_SetExceptionHandler to install your own exception handler.

The exception handler receives two parameters: the error code and a fatal error flag. The error code identifies the error (this has the same value as the message code in the message handler) and the fatal error flag tells you whether you are able to continue to use the library functions or if you must terminate your application.

If the error is nonfatal and you decide not to terminate your application in your exception handler, the execution is continued immediately after the library call that raised the exception.

If the error is fatal, the library will terminate the application, when you return from your exception handler.

In short the exception handler can be used to:

- save data before the application is terminated.
- continue when the errors are nonfatal.

Chapter 3
A Sample Application

C/FRONT comes with a sample application.

The chapter contains:

- Building and Running the Sample Application

3.1 Building and Running the Sample Application

In the C/FRONT folder on the product CD you will find a sample C/FRONT C-API program named `sample.c` (source file) and `sample.exe` (executable). You can either use the executable, or build your own by using the source file. This sample program tests the most basic C/FRONT functions and can be used as a source of inspiration for your own programs. It can connect to a database directly or via a server.

The Sample Application

The sample application consists of the following files:

File	Contents
`sample.c`	C/FRONT sample application source file.
`sample.exe`	C/FRONT sample application executable.
`sample.fbk`	Navision sample database backup.
`sample.fdb`	Navision sample database.
`sample.txt`	Text file describing how to build and run the sample application. Any last-minute changes are also described here.

You will also need the `libload.c` file – the file that contains the functions that load and unload the `cfront.dll`/`cfrontsql.dll`.

Building the C/FRONT Sample Application

1. Copy `libload.c` and `sample.c` to your compiler directory and `cf.h` to your include directory.
2. Edit `sample.c` :

   Delete `#include <cf/dbl_type.h>` and `#include <cf/dbl_td.h>`. Replace them with `#include <cf.h>`
3. Compile `sample.c` and `libload.c`.
4. Link `sample.obj` and `libload.obj` to `sample.exe`.

Running the Sample Application on SQL Server

If you are running on Microsoft SQL Server you must also perform the following tasks before you can run the sample application:

1. Open the SQL Server Option for Navision and create a new database called *sample*.
2. Restore the backup of the sample database `sample.fbk` into this new database.

Running the Sample Application

1. Copy the sample application `sample.exe` to the directory where `cfront.dll`/`cfrontsql.dll` and `sample.fdb` are placed. This is normally the directory where C/FRONT is installed.

2. Normally, the C/FRONT library (`cfront.dll/cfrontsql.dll`) reads the registry in order to locate the Navision DBMS system. However, if multiple Navision systems are installed or if Navision is not present on the system, the function SetNavisionPath in the `cfront.dll/cfrontsql.dll` library must be called specifying the path to the directory where Navision is installed or to a directory containing the following files from a Navision installation:

   ```
   dbm.dll
   nc_netb.dll
   nc_tcp.dll
   slave.exe
   fin.etx
   fin.stx
   fin.flf
   ```

   Alternatively these files can also be copied to the directory where the sample application and the database are stored.

3. You run the sample application by entering the command `sample`. But you may also enter one or more of the following parameters:

Parameter	Meaning
`-d`	Database name
`-t`	Run the exception handler test
`-s`	Server name
`-p`	Set Navision path
`-n`	Nettype – Navision Database Server: tcp, netb – SQL Server: Default, Named Piped, TCP/IP Sockets, Multiprotocol.

Note

If you are running on SQL Server, you must have a Windows login in order to connect to the sample application.

Example

If you have installed the sample application on your client computer, enter the following command to run it:

```
sample
```

If the program is started by entering the `sample` command only, the following appears on the screen during startup:

```
Company present C/FRONT Sample Company
Testing some table functions ..OK
Verifying create table functions ..OK
Verifying the table layout ..OK
Deleting all records ..OK
Creating test data ..OK
Verifying test data ..OK
Verifying modified test data ..OK
Testing string to/from type conversion  ..OK
Testing filter functions ..OK
Testing key functions ..OK
Testing bcd functions ..OK
Testing sum functions ..OK
Database test ended
```

If the program is started with a parameter (in the case illustrated below, using `sample -t`), the text will be a little different:

```
Company present C/FRONT Sample Company
Testing some table functions ..OK
Verifying create table functions ..OK
Verifying the table layout ..OK
Deleting all records ..OK
Creating test data ..OK
Verifying test data ..OK
Verifying modified test data ..OK
Testing string to/from type conversion  ..OK
Testing filter functions ..OK
Testing key functions ..OK
Testing bcd functions ..OK
Testing sum functions ..OK
Exception Handler test
TableData 2000 does not exist.
Exception Handler called with Database Error: 1001.
Database test ended
```

If you have installed the sample application on a Navision Database Server you can enter the following code when you want to run the sample application:

```
sample -s"My Server" -ntcp -penter the path
```

To use CFRONT to access a database on a Navision Database Server enter the following code:

```
sample -s"My Server" -ntcp -d"My database" -t -penter the path
```

Chapter 4
Moving From the C-Toolkit to C/FRONT

There are some important differences between the C-Toolkit for Navision 3.XX (the old text-based version) and C/FRONT for Navision.

The chapter contains:

- Overview
- New and Changed Functions
- Changes to Constants

4.1 OVERVIEW

C/FRONT differs from the C-Toolkit in a number ways. Basically, these differences reflect changes in the underlying database management system – especially the fact that key elements of the database management system have been changed to new data types. For example, field numbers were eight-bit entities in the old text -based Navision 3.XX, but now they are 32-bit entities.

A number of new functions have been added to the library, and a few have been removed. The new functions provide a higher-level of access to the database, that is: the programmer does not have to manage the physical layout of records and tables.

Finally, some of the constants in the `cf.h` header file have been changed.

4.2 New and Changed Functions

This section lists all of the new functions and the functions that have been changed in the library. The list of changed functions only contains those functions where the syntax or semantics have been changed. There is also a set of general differences that have been caused by the changes to the data types in the database management system. These changes are summarized at the end of the section.

New Functions

The following functions have been added to the library:

Function	Purpose
DBL_AddKey	Adds keys and SumIndexFields to a table
DBL_AddTableField	Adds a field to a table
DBL_AssignField	Assigns a value to a record field
DBL_CheckLicenseFile	Checks permissions in license file for object
DBL_CmpRec	Compares records
DBL_ConnectServerandOpenDatabase	Connects with a server and opens a database
DBL_CopyRec	Copies a record
DBL_CreateTable	Creates a database table
DBL_CreateTableBegin	Creates a create table handle
DBL_CreateTableEnd	Closes a create table handle
DBL_Date_2_Str	Converts date to string
DBL_DeleteTable	Deletes a table from a database
DBL_FieldDataOffset	Gets offset of field data
DBL_GetFieldData	Retrieves field data
DBL_GetFieldDataAddr	Gets address of field data
DBL_GetFieldDataSize	Gets size of field data
DBL_GetLastErrorCode	Retrieves last error code
DBL_GetVersion	Gets C/FRONT-API version number
DBL_LoadLicenseFile	Loads a license file
DBL_SetMessageShowHandler	Installs a message handing routine
DBL_SetNavisionPath	Sets path to Navision files
DBL_Str_2_Date	Converts a string to date
DBL_Str_2_Time	Converts a string to time
DBL_Time_2_Str	Converts time to a string
DBL_UseCodeUnitsPermissions	Allows you to use the permissions of a codeunit

Changed functions

The following functions have been changed syntactically and/or semantically:

Function	Purpose
DBL_AllocRec	Returns a record handle of type DBL_HREC (was: takes a record pointer as an argument)
DBL_ConnectServer	Takes a server name as an argument (was: a server number)
DBL_Init	Takes no arguments (was: took three arguments)
DBL_Login	Allows any number of login attempts (was: three attempts)
DBL_OpenDatabase	Third argument means UseCommitCache? (was: third argument was size of lazy cache)
DBL_SetExceptionHandler	Read the new description carefully

Removed Functions

The following functions have been removed or replaced:

Function	Replaced By
DBL_FieldOffset	Replaced by DBL_FieldDataOffset (read this description carefully).
DBL_RecSize	Removed without replacement.

General Changes

These general changes should be considered when moving an application from the C-Toolkit to the C/FRONT-API:

Referencing Records
: Records are no longer referenced by a pointer, but by a record handle of type DBL_HREC.

Table Numbers
: Table numbers are now of type DBL_S32.

Field Numbers
: Field numbers are now of type DBL_S32.

Keys
: Keys are now of type DBL_S32.

Date and Time
: The components of dates (year, month, date) and times (hours, minutes, seconds) are now of type DBL_S32.

Options
: Options are now of type DBL_O32.

DBL_BOOL
: The DBL_BOOL type is now defined as a DBL_U32.

DBL_DATE
: The DBL_DATE type is now defined as a DBL_U32.

4.3 Changes to Constants

The table below lists those constants in the `cf.h` file that are different in the C/FRONT-API:

Constant	New Value	Old Value
DBL_MaxRecSize	4000	1000
DBL_MaxCompanyNameLen	30	15
DBL_MaxFieldNameLen	30	20
DBL_MaxNoOfKeys	40	20
DBL_MaxFieldsPerKey	20	10
DBL_MaxSumFieldsPerKey	20	10

Field Class Constants

The three field class constants (as used in, for example, the DBL_FieldClass function) have been changed:

New	Old	Value
DBL_Class_Normal	DBL_Class_Ordinary	0
DBL_Class_FlowField	DBL_Class_Calculated	1
DBL_Class_FlowFilter	DBL_Class_CalcFilter	2

Chapter 5
The Library Functions

The functions in the C/FRONT function library are listed and described in this chapter. The functions are grouped according to use and then they are grouped alphabetically.

The chapter contains:

- Library Functions Grouped by Use
- Library Functions in Alphabetical Order

5.1 Library Functions Grouped by Use

Initialization and Finalization Functions

Function	Purpose
`DBL_Init()`	Initializes the library
`DBL_Exit()`	Closes the library
`DBL_SetNavisionPath()`	Sets the path to the Navision files

Database Functions

Function	Purpose
`DBL_ConnectServerandOpenDatabase()`	Connects to a server and opens a database
`DBL_ConnectServer()`	Connects to a server
`DBL_DisconnectServer()`	Disconnects from a server
`DBL_ReleaseAllObjects()`	Frees all resources in C/FRONT
`DBL_OpenDatabase()`	Opens a database
`DBL_CloseDatabase()`	Closes a database
`DBL_OpenCompany()`	Opens a company
`DBL_CloseCompany()`	Closes a company
`DBL_CompanyName()`	Retrieves the current company name
`DBL_NextCompany()`	Scans company names
`DBL_GetDatabaseName()`	Tests whether a database is open and – if it is – returns its name.
`DBL_CheckLicenseFile()`	Checks user permissions against license file
`DBL_LoadLicenseFile()`	Loads a license file.
`DBL_GetVersion()`	Gets version number of C/FRONT library
`DBL_AddKey()`	Adds keys and SumIndexFields to a table
`DBL_AddTableField()`	Adds a field to a table
`DBL_CreateTable()`	Creates a database table
`DBL_CreateTableBegin()`	Acquires a create table handle
`DBL_CreateTableEnd()`	Releases a create table handle
`DBL_DeleteTable()`	Deletes a table from a database

Security Functions

Function	Purpose
`DBL_Login()`	Authorizes entry to a database

Function	Purpose
`DBL_UserID()`	Retrieves current user ID
`DBL_UserCount()`	Counts users in a database
`DBL_CryptPassword()`	Encrypts password
`DBL_UseCodeUnitsPermissions`	Allows you to use the permissions of a codeunit

Table Functions

Function	Purpose
`DBL_OpenTable()`	Opens a table
`DBL_CloseTable()`	Closes a table
`DBL_OpenTemporaryTable()`	Creates a temporary table
`DBL_LockTable()`	Locks a table
`DBL_TableName()`	Retrieves a table name
`DBL_TableNo()`	Retrieves a table number
`DBL_NextTable()`	Scans table numbers
`DBL_CalcSums()`	Accumulates the sums of specified columns

Record Functions

Function	Purpose
`DBL_AllocRec()`	Creates a record buffer
`DBL_FreeRec()`	Removes a record buffer
`DBL_FindRec()`	Finds a record
`DBL_NextRec()`	Scans records
`DBL_InsertRec()`	Inserts a record
`DBL_DeleteRec()`	Deletes a record
`DBL_DeleteRecs()`	Deletes all records in a table
`DBL_ModifyRec()`	Modifies a record
`DBL_CopyRec()`	Copies a record
`DBL_CmpRec()`	Compares two records
`DBL_InitRec()`	Initializes fields in a record
`DBL_RecCount()`	Counts records
`DBL_CalcFields()`	Updates FlowFields in a record

Transaction Functions

Function	Purpose
`DBL_BWT()`	Begins a write transaction
`DBL_EWT()`	Ends a write transaction
`DBL_AWT()`	Aborts a write transaction
`DBL_SelectLatestVersion()`	Selects the latest data version

Field Functions

Function	Purpose
`DBL_FieldCount()`	Counts the number of fields in record
`DBL_NextField()`	Scans the fields in a table
`DBL_FieldLen()`	Retrieves the declared length of a field
`DBL_FieldNo()`	Retrieves a field number
`DBL_FieldName()`	Retrieves a field name
`DBL_FieldType()`	Retrieves a field type
`DBL_FieldSize()`	Retrieves the field size in bytes
`DBL_AssignField()`	Assigns a value to a field in a record
`DBL_GetFieldData()`	Retrieves data from a field
`DBL_GetFieldDataAddr()`	Retrieves the address of field data
`DBL_GetFieldDataSize()`	Retrieves the size of field data
`DBL_FieldDataOffset()`	Retrieves the offset of a field
`DBL_FieldOptionStr()`	Retrieves the option string of a field
`DBL_FieldClass()`	Retrieves the class of a field

Key Functions

Function	Purpose
`DBL_SetCurrentKey()`	Sets the current key for a table
`DBL_GetCurrentKey()`	Retrieves the current key
`DBL_KeyCount()`	Counts the keys
`DBL_NextKey()`	Scans the keys of a table
`DBL_KeySumFields()`	Retrieves the SumIndexFields of a key

Filter Functions

Function	Purpose
`DBL_SetFilter()`	Sets a filter for a field
`DBL_GetFilter()`	Retrieves the current filter
`DBL_SetRange()`	Sets a range filter for a field
`DBL_GetRange()`	Retrieves the current range filter

Conversion Functions

Function	Purpose
`DBL_Field_2_Str()`	Converts a value to a string
`DBL_YMD_2_Date()`	Converts date elements to DATE type
`DBL_Date_2_YMD()`	Converts DATE type to date units
`DBL_HMST_2_Time()`	Converts time elements to TIME type
`DBL_Time_2_HMST()`	Converts TIME type to time units
`DBL_Alpha_2_Str()`	Converts ALPHA type to string
`DBL_Str_2_Alpha( )`	Converts string to ALPHA type
`DBL_Date_2_Str()`	Converts DATE to string
`DBL_Str_2_Date()`	Converts string to DATE
`DBL_Time_2_Str()`	Converts TIME to string
`DBL_Str_2_Time()`	Converts string to TIME
`DBL_Ansi2OemBuff()`	Converts string from ANSI to OEM
`DBL_Oem2AnsiBuff()`	Converts string from OEM to ANSI

BCD Functions

Function	Purpose
`DBL_BCD_2_Str()`	Converts a BCD number to a string
`DBL_Str_2_BCD()`	Converts a string to a BCD number
`DBL_BCD_2_Double()`	Converts a BCD number to a double
`DBL_Double_2_BCD()`	Converts a double to a BCD number
`DBL_BCD_2_S32()`	Converts a BCD number to an S32
`DBL_S32_2_BCD()`	Converts an S32 to a BCD number
`DBL_BCD_IsZero()`	Tests if a BCD number has a value of 0
`DBL_BCD_IsNegative()`	Tests if a BCD number is negative
`DBL_BCD_IsPositive()`	Tests if a BCD number is positive
`DBL_BCD_Div()`	Divides one BCD number with another BCD number
`DBL_BCD_Mul()`	Multiplies one BCD number by another BCD number
`DBL_BCD_Add()`	Adds two BCD numbers together
`DBL_BCD_Sub()`	Subtracts one BCD number from another BCD number
`DBL_BCD_Abs()`	Returns the absolute value of a BCD number
`DBL_BCD_Neg()`	Reverses the sign of a BCD number
`DBL_BCD_Power()`	Raises a BCD number to a power.
`DBL_BCD_Sgn()`	Returns the sign of a BCD number
`DBL_BCD_Cmp()`	Compares one BCD number to another BCD number
`DBL_BCD_Trunc()`	Truncates a BCD number
`DBL_BCD_Round()`	Rounds a BCD number
`DBL_BCD_RoundUnit()`	Rounds a BCD number to a unit
`DBL_BCD_Make()`	Returns a BCD number

Error-Handling and Exception-Handling Functions

Function	Purpose
`DBL_Allow()`	Specifies the error to be allowed
`DBL_SetExceptionHandler()`	Installs a custom exception handler
`DBL_SetMessageShowHandler`	Installs custom message handler
`DBL_GetLastErrorCode`	Retrieves code of last error

5.2 Library Functions in Alphabetical Order

This section lists all the functions in the C/FRONT library in alphabetical order.

DBL_AddKey()

Function

Adds keys and SumIndexFields to a table.

Category

Database function.

Syntax

```
void DBL_AddKey(DBL_HCREATE_TABLE hCreateTable, DBL_S32 *Key, DBL_S32
*SumIndexFields);

hCreateTable: A create table handle (see DBL_CreateTableBegin)
Key: Zero-terminated array of the field numbers that you want to constitute
the key
SumIndexFields:Zero-terminated array of the field numbers for which you want
SumIndexFields to be maintained
```

Remarks

The first time you call DBL_ AddKey you create the primary key and the SumIndexFields that are associated with the primary key. You can then add more keys and SumIndexFields by making more AddKey calls.

The fields (and the table itself) are not created in the database until you call DBL_CreateTable.

Example

```
DBL_HCREATE_TABLE hCT;
DBL_S32     TableNo = 50000;
DBL_U8     *TableName = (DBL_U8*)"Sample Table";
DBL_S32     Key[15];
DBL_S32 SumIndexFields[15];

DBL_CreateTableBegin(&hCT, TableNo, TableName, 1);

/* create table fields */

Key[0] = 10;
Key[1] = 0;
SumIndexFields[0] = 40;
SumIndexFields[1] = 0;
DBL_AddKey(hCT, Key, SumIndexFields);
```

DBL_AddTableField()

Function

Adds a field to a table.

Category

Database function.

Syntax

```
void DBL_AddTableField(DBL_HCREATE_TABLE hCreateTable, DBL_S32 FieldNo,
DBL_U8 *FieldName, DBL_U16 FieldType, DBL_S16 FieldLen, DBL_U8 *OptionStr,
DBL_S16 FieldClass);

hCreateTable: A create table handle (see CreateTableBegin)
FieldNo: Number of the field to add
FieldName: Name of the field to add
FieldType: Type of the field to add
FieldLen: The length of the field
OptionStr: A comma-separated string of option values
FieldClass: The class of the field
```

Remarks

The FieldType can be one of these constants:

```
DBL_Type_STR
DBL_Type_DATE
DBL_Type_TIME
DBL_Type_BLOB
DBL_Type_BOOL
DBL_Type_S32
DBL_Type_ALPHA
DBL_Type_O32
DBL_Type_BCD
```

FieldLen is only relevant for fields of type DBL_Type_STR and DBL_Type_ALPHA. FieldLen corresponds to the *declared* length, not the actual length of a field.

The OptionStr should be a string like "a,b,c,d".

FieldClass can be one of these constants:

```
DBL_Class_Normal
DBL_Class_FlowField
DBL_Class_FlowFilter
```

DBL_AddTableField adds a field to a table. You must already have acquired a create table handle by calling DBL_CreateTableBegin. The table will not be created in the database before you call DBL_CreateTable on the create table handle.

You can create keys and SumIndexFields by using DBL_AddKey.

Example

```
DBL_HCREATE_TABLE hCT;
DBL_S32 TableNo = 50000;
DBL_U8 *TableName = (DBL_U8*)"Sample Table";

DBL_CreateTableBegin(&hCT, TableNo, TableName, 1);
DBL_AddTableField(hCT, 10, (DBL_U8*)"Decimal Field", DBL_Type_BCD, 0, NULL,
```

```
0);
DBL_AddTableField(hCT, 20, (DBL_U8*)"Date Field", DBL_Type_DATE, 0, NULL, 0);
DBL_AddTableField(hCT, 30, (DBL_U8*)"Time Field", DBL_Type_TIME, 0, NULL, 0);
```

DBL_AllocRec()

Function

Creates a record buffer.

Category

Record function.

Syntax

```
DBL_HREC DBL_AllocRec(DBL_HTABLE hTable);
hTable:  Handle to the table
```

Remarks

DBL_AllocRec creates a record buffer for the specified table handle. If it is successful, a handle for the record is returned. A record buffer is just an area of memory that has the same size as a record (including virtual fields – FlowFields and FlowFilters) in the specified table. Changes to the buffer do not affect the table. You change or add table records by using DBL_InsertRec, DBL_ModifyRec or DBL_DeleteRec. If sufficient memory is not available, the function will raise an exception. Always use DBL_FreeRec to remove the record buffer created by DBL_AllocRec. Never use a C run-time function for record allocation and deallocation.

Example

```
DBL_HTABLE hTable;
DBL_HREC hRec;
DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);

/* Allocate memory for record buffer */
hRec = DBL_AllocRec(hTable);

/* ... */

/* Free memory occupied by record buffer */
DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_Allow()

Function

Specifies that an error is allowed.

Category

Error function.

Syntax

```
void DBL_Allow(DBL_S32 ErrorCode);
```

ErrorCode can be one of these constants:

```
DBL_Err_TableNotFound
DBL_Err_RecordNotFound
DBL_Err_RecordExists
DBL_Err_KeyNotFound
```

Remarks

DBL_Allow permits certain library functions to be executed after a specified error has occurred that would otherwise raise an exception. These functions all have a boolean return value. If a call to such a function is successful, the function returns 1. If the function fails, one of two things can happen:

1. If you have used DBL_Allow to allow the error that causes the function to fail, the function will return 0.
2. If the error is not allowed, the function will raise an exception and call the exception handler.

For more information about exceptions, see Handling Errors and Exceptions on page 18.

DBL_Allow stores the number of the allowed error in a global variable, which is reset when the next library function is called. Always place DBL_Allow immediately before the function call in which the error is to be allowed. DBL_Allow must be invoked again if the same error is to be permitted in a subsequent function call.

Example

```
DBL_HTABLE hTable;
DBL_HREC hRec;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

/* Allow DBL_FindRec to fail */
DBL_Allow(DBL_Err_RecordNotFound);

if (DBL_FindRec(hTable,hRec,(DBL_U8*)"-"))
printf("Record was found.\n");
else
printf("Record was NOT found.\n");
```

```
DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_Alpha_2_Str()

Function — Converts an ALPHA variable to a string.

Category — Conversion function.

Syntax

```
void DBL_Alpha_2_Str(DBL_U8* Str, DBL_S16 StrSize, DBL_U8* Alpha);
Str: Variable to receive the converted string
StrSize: Size of Str in bytes, including the terminating zero
Alpha: ALPHA string to be converted
```

Remarks — DBL_Alpha_2_Str converts an Alpha variable to a string and stores it in Str. If Str is not long enough to contain the converted variable, the function raises an exception.

For more information about the ALPHA variable type, see Appendix B.

Example

```
DBL_U8 Alpha[12], Str[11];

DBL_Init();

DBL_Str_2_Alpha(Alpha, sizeof(Alpha), "Number10");
DBL_Alpha_2_Str(Str, sizeof(Str), Alpha);
printf("Variable Str now contains the string value %s\n", Str);

DBL_Exit();
```

DBL_Ansi2OemBuff

Function — Converts characters from ANSI to OEM.

Category — Conversion function.

Syntax

```
void DBL_Ansi2OemBuff(const DBL_U8 *Src,DBL_U8 *Dst,DBL_S32 DstSize)

Src: the source
Dst: the destination
DstSize: the number of characters to be converted
```

Remarks — DBL_Ansi2OemBuff converts the character buffer from ANSI to OEM. You must specify the source buffer the destination buffer and the number of characters. This function should be used in conjuction with DBL_Oem2AnsiBuff because it can

successfully convert the characters from ANSI to OEM and back again. The comparable Windows function does not always perform this conversion successfully.

Example

```
DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);

/* Ansi buffer is allocated room for 10 characters */
DBL_U8 Ansibuff[10];

/* Oem buffer is allocated room for 5 characters */
DBL_U8 Oembuff[5];

/* Copy the string "Hi" to the Ansi buffer */
strcpy(Ansibuff, "Hi")

/* Convert the two character string from ANSI to OEM */
DBL_Ansi2OemBuff(Ansibuff, Oembuff, 2);

DBL_CloseDatabase();
DBL_Exit();
```

DBL_AssignField()

Function

Assigns a value to a field in a record.

Category

Field function.

Syntax

```
void DBL_AssignField(DBL_HTABLE hTable, DBL_HREC hRec, DBL_S32 FieldNo,
DBL_U16 Type, void *Data, DBL_S32 DataSize);

hTable: handle to the table
hRec: handle to the record
FieldNo: the field number
Type: the type of the field
Data: pointer to the data
DataSize: size of the data
```

Remarks

DBL_AssignField places the data that is pointed to by the Data pointer in the FieldNo field of hRec. Note that Data is designed as a void pointer – a pointer to any kind of data type. DataSize must be set to the size of the data in bytes (for example, as returned by the C function sizeof).

Example

```
DBL_HTABLE hTable;
DBL_HREC hRec;
char s[100];

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
```

```
hRec = DBL_AllocRec(hTable);
sprintf(s,”This is some test data”);

DBL_BWT();
DBL_InitRec(hTable,hRec);
DBL_AssignField(hTable,hRec,10,DBL_FieldType(hTable,10),
s,sizeof(s));
DBL_InsertRec(hTable,hRec);
DBL_EWT();

DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_AWT()

Function

Aborts a write transaction.

Category

Transaction function.

Syntax

```
void DBL_AWT(void);
```

Remarks

DBL_AWT signals the interruption of a write transaction and undoes all of the changes that have been made in the database since DBL_BWT was issued and unlocks any locked tables. DBL_BWT must be used before DBL_AWT is called.

For more information about write transactions, see the *Application Designer’s Guide*.

Example

```
DBL_HTABLE hTable;
DBL_HREC hRec;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

/* Start write transaction */
DBL_BWT();

/* Insert and modify records using hRec and hTable */

/* Abort write transaction */
DBL_AWT();
printf("Table 15 is unchanged\n");

DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
```

```
DBL_CloseDatabase();
DBL_Exit();
```

DBL_BCD_2_Double()

Function — Converts a BCD (decimal number) to a double.

Category — BCD function.

Syntax

```
void DBL_BCD_2_Double(DBL_DOUBLE* Dest,const DBL_BCD* Source)

Dest: Variable in which the converted BCD is placed
Source: BCD variable to be converted
```

Remarks — DBL_BCD_2_Double converts Source to a double and places the result in Dest.

Example

```
DBL_BCD b1, b2;
double  d1, d2;

DBL_Init();

d1 = 12.56789;
DBL_Double_2_BCD(&b1, d1);
DBL_Double_2_BCD(&b2, d1);
DBL_BCD_Round(&b1, 2);
DBL_BCD_Trunc(&b2, 2);
DBL_BCD_2_Double(&d1, &b1);
DBL_BCD_2_Double(&d2, &b2);
if ((d1 != 12.57) || (d2 != 12.56))
return(-1);

DBL_Exit();
```

DBL_BCD_2_S32()

Function — Converts a BCD (decimal number) to an S32.

Category — BCD function.

Syntax

```
DBL_S32 DBL_BCD_2_S32(const DBL_BCD *Source);

Source: BCD variable to be converted
```

Remarks — DBL_BCD_2_S32 converts the BCD variable Source to an S32 value, and returns this value.

Example

```
DBL_BCD b1;
DBL_S32 s1, s2;

DBL_Init();

s1 = 31415;
DBL_S32_2_BCD(&b1, s1);
s2 = DBL_BCD_2_S32(&b1);

if (s1 != s2)
    return(-1);

DBL_Exit();
```

DBL_BCD_2_Str()

Function

Converts a BCD (decimal number) to a string.

Category

BCD function.

Syntax

```
void DBL_BCD_2_Str(DBL_U8* Str, DBL_S16 StrSize, DBL_BCD* Bcd);

Str:Variable in which the converted BCD is placed
StrSize:Size of Str in bytes, including the terminating zero
Bcd: BCD variable to be converted
```

Remarks

DBL_BCD_2_Str converts Bcd to a string and stores it in Str. The returned string is formatted according to the format specified in the Windows setup. This means that the following examples may yield different results, depending on the Windows setup.

Value	String
1234	"1234"
-1234	"-1234"
1234.00	"1234"
1234.050	"1234.05"
11234.56	"11234.56"
.005	"0.005"

If Str cannot hold the converted BCD, then Str is filled with '*' characters.

Example

```
DBL_BCD Bcd;
DBL_U8  Str[11];

DBL_Init();

DBL_Str_2_BCD(&Bcd, "-1.2345");
```

```
/* Variable Bcd now contains the value -1.2345 */

DBL_BCD_2_Str(Str, sizeof(Str), &Bcd);
/* Variable Str now contains the string value "-1.2345" */
printf("Variable Str now contains the string value %s\n", Str);

DBL_BCD_2_Str(Str, 6+1, &Bcd);
/* Variable Str now contains the string value "******" */
printf("Variable Str now contains the string value %s\n", Str);

DBL_Exit();
```

DBL_BCD_Abs()

Function — Converts a BCD number to the absolute value of the number.

Category — BCD function.

Syntax

```
void DBL_BCD_Abs(DBL_BCD *Dest);

Dest: the BCD number to convert
```

Remarks — DBL_BCD_Abs converts the BCD number Dest to the absolute value of that number.

DBL_BCD_Add()

Function — Adds two BCD numbers together.

Category — BCD function.

Syntax

```
DBL_BCD* DBL_BCD_Add(DBL_BCD *Dest,const DBL_BCD *Source)

Dest: destination BCD number
Source: source BCD number
```

Remarks — DBL_BCD_Add adds the BCD number Source to the BCD number Dest, and places the result in the BCD number Dest.

The function also returns the Dest BCD number. This is useful in situations where the result is only used to call yet another function, as in the following example:

h = DBL_BCD_Add(DBL_BCD_Add(dst, src),src1);

h and dst are now the same number, and that number has the value: (dst + src) + src1.

DBL_BCD_Cmp()

Function — Compares two BCD numbers.

Category — BCD function.

Syntax

```
DBL_S32 DBL_BCD_Cmp(const DBL_BCD *Left, const DBL_BCD *Right)

Left: first (left) BCD number to compare
Right: second (right) BCD number to compare
```

Remarks — DBL_BCD_Cmp compares the Left and Right BCD numbers. The return values are:

If	DBL_BCD_Cmp returns
Left > Right	1
Left < Right	-1
Left = Right	0

DBL_BCD_Div()

Function — Divides one BCD number with another BCD number.

Category — BCD function.

Syntax

```
DBL_BCD* DBL_BCD_Div(DBL_BCD *Dest,const DBL_BCD *Source)

Dest: destination BCD number
Source: source BCD number
```

Remarks — DBL_BCD_Div divides the BCD number Dest with the BCD number Source, and places the result in the BCD number Dest.

The function also returns the Dest BCD number. This is useful in situations where the result is only used to call yet another function, as in the following example:

h = DBL_BCD_Div(DBL_BCD_Div(dst, src),src1);

h and dst are now the same number, and that number has the value: (dst / src) / src1.

DBL_BCD_IsNegative()

Function — Tests whether a BCD number is negative.

Category — BCD function.

Syntax

```
DBL_BOOL DBL_BCD_IsNegative(const DBL_BCD *Source)

Source: the BCD number to test
```

Remarks — DBL_BCD_IsNegative tests if the BCD number Source is negative. DBL_BCD_IsNegative returns TRUE if the BCD number referenced by hBcd has a value less than 0 (zero). If it does not, DBL_BCD_IsNegative returns FALSE

DBL_BCD_IsPositive()

Function — Tests whether a BCD number is positive.

Category — BCD function.

Syntax

```
DBL_BOOL DBL_BCD_IsPositive(const DBL_BCD *Source)

Source: the BCD number to test
```

Remarks — DBL_BCD_IsPositive tests if the BCD number Source is positive. DBL_BCD_IsPositive returns TRUE if the BCD number referenced by hBcd has a value larger than 0 (zero). If it does not, DBL_BCD_IsPositive returns FALSE

DBL_BCD_IsZero()

Function — Tests if a BCD number has a value of 0 (zero).

Category — BCD function.

Syntax

```
DBL_BOOL DBL_BCD_IsZero(const DBL_BCD *Source)

Source: the BCD number to test
```

Remarks — DBL_BCD_IsZero tests if the BCD number Source has a value of zero. If Source has a value of zero, DBL_BCD_IsZero returns TRUE. If it does not, DBL_BCD_IsZero returns FALSE.

DBL_BCD_Make()

Function — Returns a BCD number.

Category — BCD function.

Syntax

```
DBL_BCD* DBL_BCD_Make(DBL_BCD *Dest, DBL_S32 Kind);

Dest: the BCD to make
Kind: what BCD number to make
```

Remarks — DBL_BCD_Make creates a valid BCD number, according to the Kind parameter. The created BCD number is placed in Dest and returned by the function.

The Kind parameter can have the following values:

Kind	Created BCD number
DBL_Make_Bcd_0	0
DBL_Make_Bcd_1	1
DBL_Make_Bcd_2	2
DBL_Make_Bcd_10	10
DBL_Make_Bcd_100	100
DBL_Make_Bcd_1024	1024
DBL_Make_Bcd_MIN	The BCD with the minimum value (lowest possible)
DBL_Make_Bcd_MAX	The BCD number maximum value (highest possible)

DBL_BCD_Mul()

Function — Multiplies two BCD numbers.

Category — BCD function.

Syntax

```
DBL_BCD* DBL_BCD_Mul(DBL_BCD *Dest,const DBL_BCD *Source)

Dest: destination BCD number
Source: source BCD number
```

Remarks — DBL_BCD_Mul multiples the BCD number Dest with the BCD number Source, and places the result in the BCD number Dest.

The function also returns the Dest BCD number. This is useful in situations where the result is only used to call yet another function, as in the following example:

h = DBL_BCD_Mul(DBL_BCD_Mul(dst, src),src1);

h and dst are now the same number, and that number has the value: (dst * src) * src1.

DBL_BCD_Neg()

Function — Reverses the sign of a BCD number.

Category — BCD function.

Syntax

```
void DBL_BCD_Neg(DBL_BCD *Dest);

Dest: BCD number to be converted
```

Remarks — DBL_BCD_neg reverses the sign of the BCD number Dest. For example, -3 becomes 3, 4 becomes -4 and 0 remains 0.

DBL_BCD_Power()

Function — Raises a BCD number to a specified power.

Category — BCD function.

Syntax

```
DBL_BCD* DBL_BCD_Power(DBL_BCD *Dest,const DBL_BCD *Power);

Dest: the BCD number to raise to a power
Power: the power to raise Dest to
```

Remarks — DBL_BCD_Power raises the BCD number Dest to the power Power, and puts the result in Dest.

The function also returns the Dest BCD number. This is useful in situations where the result is only used to call yet another function, as in the following example:

h = DBL_BCD_Power(DBL_BCD_Power(dst, src),src1);

h and dst are now the same number, and that number has the value: (dst ** src) ** src1.

DBL_BCD_Round()

Function — Rounds a BCD number.

Category — BCD function.

Syntax

```
void DBL_BCD_Round(DBL_BCD *Dest, DBL_S32 Cnt);

Dest: the BCD number to truncate
Cnt: the number of digits to round Dest to
```

Remarks — DBL_BCD_Round rounds the BCD number Dest to contain Cnt digits.

Examples:

Original BCD number	Cnt	Result
123.45	1	123.5
123.45	0	123
126.45	-1	130
153.45	-2	200
123.45	-40	0
-123.45	1	-123.5

DBL_BCD_RoundUnit()

Function — Rounds a BCD number to a unit.

Category — BCD function.

Syntax

```
void DBL_BCD_RoundUnit(DBL_BCD *Dest,const DBL_BCD *Unit,DBL_S32 How);

Dest: the BCD number to round
Unit: the unit to use
How: how to round
```

Remarks — DBL_BCD_RoundUnit rounds Dest according to Unit. The How parameter is used to set how rounding should occur:

If How is	DBL_BCD_RoundUnit rounds
DBL_BCD_Up	Always Up
DBL_BCD_Near	To the closest value, Up or Down
DBL_BCD_Down	Always Down

The algorithm of DBL_BCD_RoundUnit can be described like this:

1 The sign of Dest is saved

2 Dest is converted to its absolute value

3 Dest is divided by Unit.

If Unit is Null, the default unit of 0.01 is used. This means that Dest is rounded to 2 decimals.

4 The result is rounded according to How.

5 Dest is multiplied by Unit.

6 The saved sign of Dest is put back in place.

The following table shows some examples of this function:

Dest	Unit	How	Result
80	12	DBL_BCD_Up	84
80	12	DBL_BCD_Down	72
80	12	DBL_BCD_Near	84
12.5	10	DBL_BCD_Up	20

DBL_BCD_Sgn()

Function — Returns the sign of a BCD number.

Category — BCD function.

Syntax

```
DBL_S32 DBL_BCD_Sgn(const DBL_BCD *Source);

Source: the BCD number to return the sign of
```

Remarks — DBL_BCD_Sgn return the sign of the Source BCD number:

If Source is	DBL_BCD_sgn returns
> 0	1
< 0	-1
0	0

DBL_BCD_Sub()

Function — Subtracts one BCD number from another BCD number

Category — BCD function.

Syntax

```
DBL_BCD* DBL_BCD_Sub(DBL_BCD *Dest,const DBL_BCD *Source)

Dest: destination BCD number
Source: source BCD number
```

Remarks — DBL_BCD_Sub subtracts the BCD number Source from the BCD number Dest, and puts the result in the BCD number Dest.

The function also returns the Dest BCD number. This is useful in situations where the result is only used to call yet another function, as in the following example:

```
h = DBL_BCD_Sub(DBL_BCD_Sub(dst, src),src1);
```

h and dst are now the same number, and that number has the value: (dst - src) - src1.

DBL_BCD_Trunc()

Function — Truncates a BCD number.

Category — BCD function.

Syntax

```
void DBL_BCD_Trunc(DBL_BCD *Dest, DBL_S32 Cnt);

Dest: the BCD number to truncate
Cnt: the number of digits to truncate Dest to
```

Remarks — DBL_BCD_trunc truncates the BCD number Dest to contain Cnt digits.

Examples:

Original BCD number	Cnt	Result
123.45	1	123.4
123.45	0	123
123.45	-1	120
123.45	-2	100
123.45	-40	0
-123.45	1	-123.4

DBL_BWT()

Function — Begins a write transaction.

Category — Transaction function.

Syntax

```
void DBL_BWT(void);
```

Remarks

DBL_BWT marks the beginning of a set of logically related table operations (start of a transaction). The end of the transaction is signaled by a DBL_EWT or aborted with a DBL_AWT. Transactions are useful when you need to ensure that tables are not left in an inconsistent state if an operation in a set of operations fails. Multiple operations can be performed automatically with transactions.

By placing a set of operations between Begin and End transaction functions, you ensure that none of the operations are permanently recorded unless all of the operations are completed successfully.

After calling DBL_BWT, an application is allowed to modify data, using DBL_ModifyRec, etc. Calling DBL_BWT does not in itself lock the table; other users still have write access. It is only when your application begins making changes to the table or calls DBL_LockTable that the table is locked and other applications are refused access. Locked tables remain locked until either DBL_AWT or DBL_EWT is called.

For more information about write transactions, see the *Application Designer's Guide*.

Example

```
DBL_HTABLE hTable;
DBL_HREC hRec;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

/* Start write transaction */
DBL_BWT();

/* Process hRec and hTable */

/* Commit write transaction */
DBL_EWT();

DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_CalcFields()

Function

Updates FlowFields in a record.

Category

Record function.

Syntax

```
void DBL_CalcFields(DBL_HTABLE hTable, DBL_HREC hRec, DBL_S32* FieldList);

hTable:Handle to the table
hRec:Handle to the record whose FlowFields are to be updated
FieldList:Zero-terminated array containing the numbers of the fields in hRec
to be updated
```

Remarks

DBL_CalcFields updates any FlowFields that exist in a record. FlowFields are a special C/SIDE feature that provide information from other tables in the database.

They are also known as virtual fields because their values are not saved with the table. FlowFields are only updated when DBL_CalcFields is called. For example, FlowFields in records that are accessed with DBL_FindRec and DBL_NextRec are set to zero. DBL_CalcFields must be called to update the values in the FlowFields.

A detailed explanation and illustration of FlowFields is available in the *Application Designer's Guide*.

Example

```
DBL_HTABLE hTable;
DBL_HREC hRec;
DBL_S32 Fields[11];

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

/* Update value in field 5 (a FlowField) in hRec     */
/* Causes an exception if field 5 is not a FlowField */
Fields[0] = 5;
Fields[1] = 0; /* Remember to zero-terminate strings */
DBL_CalcFields(hTable, hRec, Fields);

DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_CalcSums()

Function

Sums up totals for selected fields.

Category

Table function.

Syntax

```
void DBL_CalcSums(DBL_HTABLE hTable, DBL_HREC hRec, DBL_S32* FieldList);

hTable: Handle to the table
hRec: Handle to the record buffer in which sums are to be placed
FieldList: Zero-terminated array containing the numbers of the fields to be
added up
```

Remarks

DBL_CalcSums adds up totals for specific fields (columns) in a table. The function operates only on those records that meet the conditions specified in any filters that are associated with the table handle.

All of the fields listed in FieldList must be designated in the current key as SumIndexFields. If any of them do not meet this criteria, an exception is raised. To retrieve a list of the SumIndexFields for a given key, call DBL_KeySumFields.

SumIndexFields are a special C/SIDE feature that give speedy access to numeric totals. SumIndexFields also work in tables that contain many thousands of records.

For more information about SumIndexFields, see the *Application Designer's Guide*.

Example

```
DBL_HTABLE hTable;
DBL_HREC hRec;
DBL_S32 Fields[DBL_MaxSumFieldsPerKey+1];

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

Fields[0] = 4;
Fields[1] = 0;
DBL_CalcSums(hTable, hRec, Fields);

DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_CheckLicenseFile()

Function — Checks whether the license file that is currently in use contains permissions for a specified object.

Category — Database function.

Syntax

```
void DBL_CheckLicenseFile(DBL_S32 ObjectNo);

ObjectNo: number of the object to check permissions for
```

Remarks — The permissions for the object specified by ObjectNo are checked against the license file. If the required permissions are not specified in the license file, an error (which you will have to handle) occurs.

Example

```
DBL_CheckLicenseFile(9110); /* C/FRONT license check */
```

DBL_CloseCompany()

Function — Closes the company that is currently open.

Category — Database function.

Syntax

```
void DBL_CloseCompany(void);
```

Remarks — DBL_CloseCompany closes the company that was opened by DBL_OpenCompany. You must close any tables that are open before calling this function.

Example

```
DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);

/* Open company.                                          */
/* Causes an exception if "Test Company" does not exist */
DBL_OpenCompany("Test Company");

printf("Current company is %s\n", DBL_CompanyName());

/* ... */

/* Close company */
DBL_CloseCompany();

DBL_CloseDatabase();
DBL_Exit();
```

DBL_CloseDatabase()

Function — Closes the database that is open.

Category — Database function.

Syntax

```
void DBL_CloseDatabase(void);
```

Remarks — Closes the database that was opened by DBL_OpenDatabase. DBL_CloseDatabase will raise an exception if there are any open tables or allocated records (see DBL_ReleaseAllObjects).

Example

```
DBL_Init();

/* Open database using 2000 Kb cache                        */
/* Causes an exception if database MY DB does not exist     */
/* Causes an exception if 2000 Kb cache cannot be allocated */
DBL_OpenDatabase("MY DB.fdb", 2000, 0);

/* ... */

/* Close database */
DBL_CloseDatabase();

DBL_Exit();
```

DBL_CloseTable()

Function — Closes the specified table.

Category — Table function.

Syntax

```
void DBL_CloseTable(DBL_HTABLE hTable);

hTable: Handle to the table
```

Remarks — DBL_CloseTable deletes a handle to a table that was opened by DBL_OpenTable and frees the memory occupied by such things as filters. hTable is no longer a valid handle after this operation has been called.

For more information about tables, see the *Application Designer's Guide*.

Example

```
DBL_HTABLE hTable;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
```

```
/* Open table                                        */
/* Causes an exception if table 15 does not exist */
DBL_OpenTable(&hTable, 15);

/* ... */

/* Close table */
DBL_CloseTable(hTable);

/* Open table */
DBL_Allow(DBL_Err_TableNotFound);
if (DBL_OpenTable(&hTable, 16))
{
printf("Table opened\n");

/* ... */

/* Close table */
DBL_CloseTable(hTable);
}
else
printf("Table does not exist\n");

DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_CmpRec()

Function — Compares two records.

Category — Record function.

Syntax

```
DBL_BOOL DBL_CmpRec(DBL_HTABLE hTable, const DBL_HREC hDstRec, const DBL_HREC
hSrcRec);

hTable: handle to the table
hDstRec: handle to a record
hSrcRec: handle to the record to compare hDstRec against
```

Remarks — hDstRec and hSrcRec must be handles to records from the same table. If the contents of the records are the same, the function returns TRUE. If the contents are not the same, it returns FALSE.

Example

```
DBL_HTABLE hTable;
DBL_HREC hDstRec;
DBL_HREC hSrcRec;
char a[100];
char b[100];
```

```
DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hDstRec = DBL_AllocRec(hTable);
hSrcRec = DBL_AllocRec(hTable);
sprintf(a,"Some data");
sprintf(b,"Some other data");

/* Assign values to the fields of the two records. */
DBL_InitRec(hTable,hDstRec);
DBL_InitRec(hTable,hSrcRec);
DBL_AssignField(hTable,hDstRec,10,DBL_Type_STR,a,sizeof(a));
DBL_AssignField(hTable,hSrcRec,10,DBL_Type_STR,b,sizeof(b));

/* Then, compare the records: */
if (DBL_CmpRec(hTable,hDstRec,hSrcRec))
printf("The records are identical\n");
else
printf("The records are NOT identical\n");

DBL_FreeRec(hDstRec);
DBL_FreeRec(hSrcRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_CompanyName()

Function — Retrieves the name of the company that is currently open.

Category — Database function.

Syntax

```
const DBL_U8* DBL_CompanyName(void);
```

Remarks — DBL_CompanyName returns the CompanyName of the company that was opened by DBL_OpenCompany. The returned value is a pointer to a string that contains the current CompanyName. If DBL_OpenCompany has not been called, the function returns a pointer to an empty string.

You cannot select another CompanyName with this function. You must use DBL_CloseCompany and DBL_OpenCompany to change to another company. Companies can only be created and deleted in C/SIDE.

Example

```
DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);

/* Open company                                          */
/* Causes an exception if "Test Company" does not exist */
```

```
DBL_OpenCompany("Test Company");

printf("Current company is %s\n", DBL_CompanyName());

/* Close company */
DBL_CloseCompany();

DBL_CloseDatabase();
DBL_Exit();
```

DBL_ConnectServerandOpenDatabase()

Function

Connects to a server and opens a database.

Category

Database function.

Syntax

```
void DBL_ConnectServerandOpenDatabase(DBL_U8* NDBCDriverName, DBL_U8*
ServerName, DBL_U8* NetType, DBL_U8* DatabaseName, DBL_S32 CacheSize DBL_BOOL
UseCommitCache, DBL_Bool UseNTAuthentication, DBL_U8* UserID, DBL_U8*
PassWord);

NDBCDriverName: "NDBCS": The Microsoft SQL Server driver
     "NDBCN": The Navision Database Server driver

ServerName: Name of the server

NetType: "Default": use the default Net Type for network communication
"Named Pipes": use Named Pipes for network communication
"TCP/IP Sockets": use TCP/IP Sockets for network communication
"Multiprotocol": use Multiprotocol for network communication

DatabaseName: Name of the database to open

CacheSize: Size of cache in KB

UseCommitCache: Whether to use CommitCache or not

UseNTAuthentication: Whether to use NT Authentication or not

UserID: Login name

PassWord: Password belonging to UserID
```

Remarks

You must use this function with the SQL Server Option for Navision and we recommend that you also use it with Navision Database Server. It must be used with the SQL Server Option for Navision because SQL Server demands that you open the server, the database and provide authentication at the same time.

The Net Type you select is dependent on the server you are using:

Server:	Value:	Net Type:
SQL Server	Named Pipes	Named Pipes
	TCP/IP Sockets	TCP/IP Sockets
	Multiprotocol	Multiprotocol
	Default	Default Net Type
Navision Database Server	tcp	TCP/IP
	netb	NetBIOS

The driver you select is dependent on the server you are using.

SQL Server

The server must be running when you issue this call. All succeeding calls to the database will be passed to the server, which will execute the operations.

If you are using SQL Server you still have to enter a zero value for CacheSize and UseCommitCache even though they only apply to the Navision Database Server.

If you select NT Authentication (`UseNTAuthentication=1`) then you do not have to supply a user ID or a password. However, you do have to enter two empty sets of quotes ("") in order to comply with the syntax.

Issue this call again to open another database. You can only open the databases to which you have been granted permission.

Navision Database Server

This function connects with a server and opens a database with a cache of the CacheSize you have specified. The server must be running when you issue this call. All succeeding calls to access the database are passed to the server, which will execute the operations.

A cache is an area of RAM that holds the results of recent disk accesses. CacheSize specifies the amount of memory assigned to the disk cache. The size depends upon which operating system is in use. As a general rule, the larger the cache, the better the performance. For more information, see the *Installation and System Management: Microsoft Business Solutions–Navision Database Server* manual.

If you are using the Navision Database Server and do not specify a server name (`ServerName=0`) the application will search for the database file on your local computer and open it, if it can be found.

If you select NT Authentication (`UseNTAuthentication=1`) then you do not have to supply a user ID or a password. However you do have to enter two empty sets of quotes ("") in order to comply with the syntax.

To close the connection to a database server, call DisconnectServer.

An application can only be connected to one server at a time.

Example

SQL Server

```
DBL_Init();
/* Connect to TestServer using Named Pipes and NT Authentication */

DBL_ConnectServerandOpenDatabase("NDBCS", "TestServer",
"Named Pipes", "test.fdb", 0, 0, 1, "", "");

/* Causes an exception if TestServer is not connectable*/
/* Causes an exception if database test.fdb does not exist*/
/* Causes an exception if the NT login does not give the user access to this
server*/
/*CacheSize and UseCommitCache do not apply to SQL Server*/
/*Using NT Authentication means that you do not need to enter a UserID and
Password*/

/*...*/
/* Disconnect from server */
DBL_DisconnectServer();
DBL_Exit();
```

Navision Database Server

```
DBL_Init();
/* Connect to the Navision Database Server "TestServer" using NetBIOS and
open testdb using 3000Kb cache, commitcache, not using NT Authentication and
therefore providing a user ID and password*/
DBL_ConnectServerandOpenDatabase("NDBCN", "TestServer", "netb", "test.fdb",
3000, 1, 0, "MyUserID", "MyPassword");

/* Causes an exception if TestServer is not connectable*/
/* Causes an exception if database test.fdb does not exist*/
/* Causes an exception if 3000 KB cannot be allocated*/
/* Causes an exception if UserID is not correct*/
/* Causes an exception if password is not correct*/

/* ...*/
/* Disconnect from server */
DBL_DisconnectServer();

DBL_Exit();
```

DBL_ConnectServer()

Function — Connects to a database server.

Category — Database function.

Syntax

```
void DBL_ConnectServer(DBL_U8* ServerName, DBL_U8* NetType);

ServerName: name of the server to connect to
```

```
NetType: "netb": use NetBIOS for network communication
"tcp":  use TCP/IP for network communication
```

Remarks

The server must be running when you issue this call. All succeeding calls to access the database will be passed to the server, which will execute the operations.

This function is applicable only in a multiuser configuration.

To close the connections to a database server, call DBL_DisconnectServer.

An application can be connected to only one server at a time; use DBL_DisconnectServer before connecting to another. Applications can switch between a server connection and a locally-opened database (you can alternate between DBL_ConnectServer and DBL_OpenDatabase); remember to close the existing connection before making the switch.

If an error occurs, the function will raise an exception and call the exception handler.

For more information about exceptions, see Handling Errors and Exceptions on page 18.

Example

```
DBL_Init();

/* Connect to TestServer using NetBIOS                  */
/* Causes an exception if TestServer is not connectable */
DBL_ConnectServer("TestServer", "netb");

/* ...*/
/* Disconnect from server */
DBL_DisconnectServer();

DBL_Exit();
```

DBL_CopyRec()

Function

Copies a record.

Category

Record function.

Syntax

```
void DBL_CopyRec(DBL_HTABLE hTable, DBL_HREC hDstRec, const DBL_HREC
hSrcRec);

hTable: handle to the table
hDstRec: handle to the record to copy to
hSrcRec: handle to the record to copy from
```

Remarks

DBL_CopyRec copies the contents of one record to another record. hSrcRec and hDstRec must be handles to records from the same table.

Example

```
DBL_HTABLE hTable;
DBL_HREC hDstRec;
DBL_HREC hSrcRec;
char s[100];
DBL_S32 num;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hDstRec = DBL_AllocRec(hTable);
hSrcRec = DBL_AllocRec(hTable);
sprintf(s,"New data");

DBL_BWT();
/* Retrieve a record: */
DBL_FindRec(hTable,hSrcRec,(DBL_U8*)"-")
/* And copy it: */
DBL_CopyRec(hTable,hDstRec,hSrcRec);
/* Process hDstRec, and insert it into the table: */
num = 4711;
DBL_AssignField(hTable,hDstRec,1,DBL_FieldType(hTable,1),
num,sizeof(DBL_S32));
DBL_AssignField(hTable,hDstRec,10,DBL_FieldType(hTable,10),
s,sizeof(s));
DBL_InsertRec(hTable,hDstRec);
DBL_EWT();

DBL_FreeRec(hDstRec);
DBL_FreeRec(hSrcRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_CreateTable()

Function

Creates a database table.

Category

Database function

Syntax

```
DBL_BOOL DBL_CreateTable(DBL_HCREATE_TABLE hCreateTable)

hCreateTable: Create table handle
```

Remarks

DBL_CreateTable creates a table in a Navision database. Before using DBL_CreateTable you have to acquire a create table handle by using DBL_CreateTableBegin, and add fields and keys with DBL_AddTableField and DBL_AddKey.

DBL_CreateTable returns TRUE if the table is created successfully and FALSE if it is not created. When creating the table fails, the error handler is called.

If you use DBL_CreateTableEnd without using DBL_CreateTable, the table will not be created.

Example

```
DBL_HCREATE_TABLE hCT;
DBL_S32 TableNo = 50000;
DBL_U8 *TableName = (DBL_U8*)"Sample Table";

DBL_CreateTableBegin(&hCT, TableNo, TableName, 1);

/* add table fields, keys and SumIndexFields */

DBL_CreateTable(hCT);
DBL_CreateTableEnd(hCT);
```

DBL_CreateTableBegin()

Function

Creates a create table handle.

Category

Database function.

Syntax

```
DBL_CreateTableBegin(DBL_HCREATE_TABLE *phCreateTableRef, DBL_S32 TableNo,
DBL_U8 *TableName, DBL_BOOL DataPerCompany);

phCreateTableRef: create table handle
TableNo: number of the table
TableName: name of the table
DataPerCompany: if FALSE, data in the table will be available to all
companies in the database; if TRUE, it will only be available to the currently
selected company.
```

Remarks

DBL_CreateTableBegin creates a create table handle. After you have created this handle, you can use DBL_AddTableField and DBL_AddKey to add fields and keys to the table. When you have finished defining fields and keys, you use DBL_CreateTable to create the table in C/SIDE.

When the table has been created with DBL_CreateTable, you must close the create table handle with DBL_CreateTableEnd. If you use DBL_CreateTableEnd without using DBL_CreateTable first, the table will not be created.

Example

```
DBL_HCREATE_TABLE hCT;
DBL_S32     TableNo = 50000;
DBL_U8     *TableName = (DBL_U8*)"Sample Table";

DBL_CreateTableBegin(&hCT, TableNo, TableName, 1);

/* add table fields, keys and SumIndexFields */
```

```
DBL_CreateTable(hCT);
DBL_CreateTableEnd(hCT);
```

DBL_CreateTableEnd()

Function — Closes a create table handle.

Category — Database function.

Syntax

```
DBL_CreateTableEnd(DBL_HCREATE_TABLE hCreateTable);

hCreateTable: Create table handle.
```

Remarks — DBL_CreateTableEnd closes the create table handle that hCreateTable acquired by calling DBL_CreateTableBegin.

If you have not called DBL_CreateTable, any fields and keys you may have defined with DBL_AddTableField and DBL_AddKey will be lost.

Example

```
DBL_HCREATE_TABLE hCT;
DBL_S32     TableNo = 50000;
DBL_U8     *TableName = (DBL_U8*)"Sample Table";

DBL_CreateTableBegin(&hCT, TableNo, TableName, 1);

/* add table fields, keys and SumIndexFields */

DBL_CreateTable(hCT);
DBL_CreateTableEnd(hCT);
```

DBL_CryptPassword()

Function — Encrypts the password.

Category — Password function

Syntax

```
DBL_CryptPassword(const DBL_U8 *UserID, DBL_U8 *PassWord)

UserId: the user ID
PassWord: the clear-text password
```

Remarks — DBL_CryptPassword encrypts a password, if you have supplied a user ID and a clear-text version of the password. The result is placed in PassWord.

Example

```
DBL_HTABLE  hTable;
DBL_HREC    hRec;
DBL_U8     *UserID = (DBL_U8*)"UID";
```

```
DBL_U8      UserIDCode[12];
DBL_U8      Password[11];
DBL_U8     *EncryptedPassword;

printf("Testing login functions ..");
strcpy((char*)Password,"Password");
DBL_Login(UserID, Password);
DBL_OpenTable(&hTable, 2000000002);   /* 'User' table */
hRec = DBL_AllocRec(hTable);
DBL_InitRec(hTable, hRec);
DBL_BWT();
DBL_DeleteRecs(hTable);
DBL_EWT();

DBL_Str_2_Alpha(UserIDCode, sizeof(UserIDCode), UserID);

DBL_AssignField(hTable,hRec,1,DBL_FieldType(hTable,1),UserIDCode,strlen((cha
r*)UserID)+2);

DBL_AssignField(hTable,hRec,2,DBL_FieldType(hTable,2),Password,
strlen((char*)Password)+1);

DBL_BWT();
DBL_InsertRec(hTable, hRec);
DBL_EWT();
DBL_InitRec(hTable, hRec);
DBL_FindRec(hTable, hRec, (DBL_U8*)"-");

EncryptedPassword =
(DBL_U8*)DBL_GetFieldDataAddr(hTable, hRec, 2);
DBL_CryptPassword(UserIDCode, Password);

if (0 != memcmp(EncryptedPassword, Password, strlen((char*)Password)))
{
DBL_CloseTable(hTable);
printf("Login functions \nError !");
return(-1);
}

DBL_BWT();
DBL_DeleteRecs(hTable);
DBL_EWT();
DBL_CloseTable(hTable);
printf("OK\n");
```

DBL_Date_2_Str()

Function	Converts a DATE element to a string.
Category	Conversion function
Syntax	`void DBL_Date_2_Str(DBL_U8 *Str, DBL_S16 StrSize, DBL_DATE Date);`

```
Str: the string in which to place the converted DATE
StrSize: the size of the destination string
Date: the DATE element to convert
```

Remarks

DBL_Date_2_Str converts the DATE value in Date to a string. StrSize is the size of the destination string, in effect, the number of bytes to place in Str. It is your own responsibility to ensure that the converted value will not be truncated.

Example

```
DBL_HTABLE hTable;
DBL_HREC hRec;
DBL_U8 resStr[50];
DBL_DATE *pDate;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

/* Retrieve a record: */
DBL_FindRec(hTable,hSrcRec,(DBL_U8*)"-")
/* Get pDate from field 20: */
pDate = (DBL_DATE*)DBL_GetFieldDataAddr(hTable,hRec,20);
/* Convert pDate to a string: */
DBL_Date_2_Str(resStr,sizeof(resStr),*pDate);
/* Print out the string: */
printf("Date as string: %s\n", resStr);

DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_Date_2_YMD()

Function

Converts a DATE variable to year, month, day.

Category

Conversion function.

Syntax

```
void DBL_Date_2_YMD(DBL_S32* y, DBL_S32* m, DBL_S32* d,
DBL_BOOL* Closing, DBL_DATE Date);

y: Variable to receive the value for year
m: Variable to receive the value for month
d: Variable to receive the value for day
Closing: 1 if the Date value is designated as a closing date,
otherwise 0
Date: DATE variable to be converted
```

Remarks

DBL_Date_2_YMD dismantles a DATE variable to create separate values for year, month and day.

Any of the four output variables (y, m, d or Closing) can be set to NULL if they are not needed.

If this function is called with Date=zero (undefined), an exception is raised. To prevent this occurring, test the value of the Date variable before making this call.

Example

```
DBL_DATE Date;
DBL_S32 y,m,d;
DBL_BOOL c;

DBL_Init();

DBL_YMD_2_Date(&Date, 1996, 5, 17, 0);
/* Variable Date now contains the date May 17, 1996 */

DBL_Date_2_YMD(&y, &m, &d, &c, Date);
printf("y,m,d and c now contain %d, %d, %d and %d\n",y,m,d,c);

DBL_Exit();
```

DBL_DeleteRec()

Function

Deletes a record from a table.

Category

Record function.

Syntax

```
DBL_BOOL DBL_DeleteRec(DBL_HTABLE hTable, DBL_HREC hRec);

hTable: Handle to the table
hRec: Handle to the record to be deleted. hRec itself does not change.
```

Remarks

DBL_DeleteRec deletes a record from an open table. The current key and any filters bound to the table handle have no effect on this operation. The record to be deleted is identified only by the values in its primary key.

In a multiuser environment, another application can delete the record from the table in the interval between your reading the record and your attempt to delete it. The C/SIDE database system automatically detects such an event, causing DBL_DeleteRec to raise an exception.

To prevent this from happening, use DBL_LockTable to lock the table before reading the record. Remember, however, that the table will be locked for the time that elapses between reading and deleting the record, and that other users will therefore be unable to access it.

For more information about table locking, see the *Application Designer's Guide*.

If the record is successfully deleted, 1 is returned. If the record is not found in the table, two things can happen:

1 If this result has been allowed by the function: DBL_Allow(DBL_Err_RecordNotFound), 0 is returned.

2 If the result is not allowed, the function will raise an exception and call the exception handler with an error.

For more information about exceptions, see Handling Errors and Exceptions on page 18.

Example

```
DBL_HTABLE hTable;
DBL_HREC hRec;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

DBL_BWT();

DBL_FindRec(hTable,hRec,(DBL_U8*)"-")

DBL_DeleteRec(hTable, hRec);

DBL_EWT();

DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_DeleteRecs()

Function — Deletes all of the records in a table.

Category — Record function.

Syntax

```
DBL_DeleteRecs(DBL_HTABLE hTable)

hTable: handle to the table
```

Remarks — DBL_DeleteRecs deletes all of the records in the table referenced by hTable. DBL_DeleteRecs must be used inside a transaction.

Example

```
DBL_HTABLE  hTable;
```

```
DBL_OpenTable(&hTable, 123456);
DBL_BWT();
DBL_DeleteRecs(hTable);
DBL_EWT();
DBL_CloseTable(hTable);
```

DBL_DeleteTable()

Function — Deletes a table from a database.

Category — Database function.

Syntax

```
DBL_BOOL DBL_DeleteTable(DBL_S32 TableNo);

TableNo: the number of the table to delete
```

Remarks — DBL_DeleteTable deletes the table with table number TableNo from the database. If the table was found and it could be deleted, DBL_DeleteTable returns TRUE.

If DBL_DeleteTable returns FALSE, an error has occurred. This means that either the table could not be found or that the table was locked. When DBL_DeleteTable returns FALSE, the error handler is called.

DBL_DisconnectServer()

Function — Disconnects from a database server.

Category — Database function.

Syntax

```
void DBL_DisconnectServer(void);
```

Remarks — DBL_DisconnectServer disconnects an application from a database server. The connection must have been established by calling DBL_ConnectServer.

Example

```
DBL_Init();

/* Connect to TestServer using NetBIOS                  */
/* Causes an exception if TestServer is not connectable */
DBL_ConnectServer("TestServer", "netb");

/* Disconnect from server */
DBL_DisconnectServer();

DBL_Exit();
```

DBL_Double_2_BCD()

Function — Converts a BCD (decimal number) to a double.

Category — BCD function.

Syntax

```
void DBL_Double_2_BCD(DBL_BCD *Dest,DBL_DOUBLE Source)

Dest: Variable in which the converted double is placed
Source: double variable to be converted
```

Remarks — DBL_Double_2_BCD converts Source to a BCD and places the result in Dest.

Example

```
DBL_BCD b1, b2;
double  d1, d2;

DBL_Init();

d1 = 12.56789;
DBL_Double_2_BCD(&b1, d1);
DBL_Double_2_BCD(&b2, d1);
DBL_BCD_Round(&b1, 2);
DBL_BCD_Trunc(&b2, 2);
DBL_BCD_2_Double(&d1, &b1);
DBL_BCD_2_Double(&d2, &b2);
if ((d1 != 12.57) || (d2 != 12.56))
return(-1);

DBL_Exit();
```

DBL_EWT()

Function — Ends a write transaction.

Category — Transaction function.

Syntax

```
void DBL_EWT(void);
```

Remarks — DBL_EWT signals the end of a transaction. It completes the ongoing transaction and makes the appropriate changes to the tables. All modifications that have been made to the database since DBL_BWT was called are committed, and any locked tables are unlocked.

A call to DBL_BWT must precede a call to DBL_EWT. An application cannot abort a transaction after a DBL_EWT operation.

For more information about write transactions, see the *Application Designer's Guide*.

Example

```
DBL_HTABLE hTable;
DBL_HREC hRec;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

/* Start write transaction */
DBL_BWT();

/* Insert and modify records */

/* Commit write transaction */
DBL_EWT();

DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_Exit()

Function

Closes the C/FRONT environment.

Category

Initializing function.

Syntax

```
void DBL_Exit(void);
```

Remarks

DBL_Exit closes the single-user or network C/FRONT environment that was previously opened with DBL_Init. DBL_Exit aborts any outstanding write transactions, removes any table locks, closes any open tables and frees internal buffer areas and internal tables.

DBL_Init must have been called before this function is used.

Example

```
void main(int argc, char* argv[], char* envp[])
{
DBL_Init();

/* A database can now be opened or      */
/* a connection to a server established */

DBL_Exit();
}
```

DBL_Field_2_Str()

Function

Converts a value in a field to a string.

Category

Conversion function.

Syntax

```
void DBL_Field_2_Str(DBL_U8* Str, DBL_S16 StrSize, DBL_HTABLE hTable,
DBL_HREC hRec, DBL_S32 FieldNo);

Str: Variable in which the converted field will be placed
StrSize:Size of Str in bytes, including the terminating zero
hTable: Handle to the table
hRec: Handle to the record containing the field to be converted
FieldNo: Number of the field containing the value to be converted
```

Remarks

DBL_Field_2_Str converts the contents of FieldNo in hRec to a zero-terminated ASCII string and places it in Str. The field represented by FieldNo may be of any type.

If Str is too short to hold the converted string, the converted string is truncated from the right. If FieldNo is of type integer, large integer or decimal, Str is filled with '*' characters.

Example

```
DBL_HTABLE hTable;
DBL_HREC hRec;
DBL_U8     Str[11];

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

DBL_FindRec(hTable, hRec, (DBL_U8*)"-");

/* Let's assume that:                                    */
/* Field 10 in hRec is a BCD type field containing 1.23456 */
/* Field 11 in hRec is a TEXT type field containing      */
/* "This is a test"                                      */

DBL_Field_2_Str(Str, 6+1, hTable, hRec, 10);
/* Variable Str now contains the string value "******" */
printf("Variable Str now contains the string value %s\n", Str);

DBL_Field_2_Str(Str, 7+1, hTable, hRec, 11);
/* Variable Str now contains the string value "This is" */
printf("Variable Str now contains the string value %s\n", Str);

DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_FieldClass()

Function

Retrieves the class of a specified field.

Category

Field function.

Syntax

```
DBL_S16 DBL_FieldClass(DBL_HTABLE hTable, DBL_S32 FieldNo);

hTable: Handle to the table
FieldNo: Number of the field whose class is to be determined
```

Remarks

There are three different classes of fields: Normal, FlowField or FlowFilter. DBL_FieldClass returns the class of the field with the number specified in FieldNo.

If the number specified in FieldNo does not exist, the function will raise an exception.

The value returned by DBL_FieldClass is one of the three constants:

```
DBL_Class_Normal
DBL_Class_FlowField
DBL_Class_FlowFilter
```

These constants are listed in Appendix A.

While ordinary fields are stored in the database, FlowFields are virtual fields that contain information about other tables in the database and are not saved with the table. To update the FlowFields you must call DBL_CalcFields.

FlowFilter fields are also virtual fields. The values in FlowFilter fields are used as parameters for calculating FlowFields.

For more information about field classes see the *Application Designer's Guide*.

Example

```
DBL_HTABLE hTable;
DBL_S16    Class;

DBL_Init();
DBL_OpenDatabase("test-fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);

/* Retrieve class of field 3 */
Class = DBL_FieldClass(hTable, 3);

if (Class == DBL_Class_Normal)
printf("Field 3 is a field stored in the database\n");
else
{
printf("Field 3 is NOT a field stored in the database\n");

if (Class == DBL_Class_FlowField)
printf("because it is a FlowField\n");
else
```

```
printf("because it is a FlowFilter field\n");
}

DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_FieldCount()

Function

Counts the number of fields (columns) in a table.

Category

Field function.

Syntax

```
DBL_S16 DBL_FieldCount(DBL_HTABLE hTable);

hTable: Handle to the table
```

Remarks

DBL_FieldCount retrieves the number of fields in a record in a given table. Only active fields are counted.

Example

```
DBL_HTABLE hTable;
DBL_S16    FieldNo;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);

Init(("Table 15 contains %d active field(s)\n",
 DBL_FieldCount(hTable));

printf("These fields are numbered as follows:\n");
for (FieldNo = 0; FieldNo = DBL_NextField(hTable, FieldNo;)
printf("%d\n", FieldNo);


DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_FieldDataOffset()

Function — Returns the offset of a specified field in a record.

Category — Field function.

Syntax

```
DBL_S16 DBL_FieldDataOffset(DBL_HTABLE hTable, DBL_U8 FieldNo);

hTable: Handle to the table
FieldNo: Number of the field whose offset is to be retrieved
```

Remarks — DBL_FieldDataOffset returns the offset of the field with field number FieldNo in bytes, thereby indicating its position in relation to the first byte of the record. Generally, you should not use this function to assign values to or retrieve values from fields – use the DBL_AssignField and DBL_GetFieldData functions instead. This function is not expected to exist in future versions of the C/FRONT API, and it is only included for compatibility reasons for now.

If FieldNo does not exist, the function will raise an exception.

DBL_FieldLen()

Function — Retrieves the declared length of a specified field.

Category — Field function.

Syntax

```
DBL_S16 DBL_FieldLen(DBL_HTABLE hTable, DBL_S32 FieldNo);

hTable: Handle to the table
FieldNo: Number of the field whose size will be retrieved
```

Remarks — DBL_FieldLen returns the declared length of FieldNo. If FieldNo does not exist, the function will raise an exception.

While DBL_FieldSize returns the actual length of the data in a field, DBL_FieldLen returns the declared length. A text field could, for example, be declared with length 30, while the actual size of the data in the field is 8 bytes.

For more information about C/SIDE field types and their sizes, see the *Application Designer's Guide* and Appendix A.

DBL_FieldName()

Function — Retrieves the name of a specified field.

Category — Field function.

Syntax

```
const DBL_U8* DBL_FieldName(DBL_HTABLE hTable, DBL_S32 FieldNo);

hTable: Handle to the table
FieldNo: Number of the field whose name will be retrieved
```

Remarks — DBL_FieldName returns the name of the field identified by FieldNo. This function cannot be used to change the name of the field. Field names are defined in C/SIDE.

If no field exists with the number specified in FieldNo, the function will raise an exception. Because the field name is returned as a pointer, it (the pointer) is only valid while the handle to the table is open.

For more information about fields, see the *Application Designer's Guide*.

Example

```
DBL_HTABLE hTable;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);

/* Retrieve information about field 3 in table 15            */
/* Causes an exception if field 3 does not exist in table 15 */
printf("Field name   is %s\n", DBL_FieldName(hTable, 3));
printf("Field type   is %d\n", DBL_FieldType(hTable, 3));
printf("Field size   is %d\n", DBL_FieldSize(hTable, 3));

DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_FieldNo()

Function — Retrieves the number of a specified field that is identified by name.

Category — Field function.

Syntax

```
DBL_S32 DBL_FieldNo(DBL_HTABLE hTable, DBL_U8* FieldName);

hTable: Handle to the table
FieldName: Name of the field whose number is to be retrieved
```

Remarks

DBL_FieldNo returns the field number of FieldName.

Each field is uniquely identified by both a number and a name. Two fields in the same table cannot have the same number or name – this is checked when fields are created in C/SIDE.

Fields are normally accessed by number because the random access used for numbers is faster than the sequential scan used for names. Therefore, you should only call DBL_FieldNo to look up the number of a field if your system does not support field numbers, and there is no alternative.

If no field exists with the name specified in FieldName, the function will raise an exception.

Example

```
DBL_HTABLE hTable;
DBL_S32    FieldNo;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);

/* Look up MyField in table 15 */
FieldNo = DBL_FieldNo(hTable, "TestField");

printf("TestField in table 15 has the number %d\n", FieldNo);

DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_FieldOptionStr()

Function

Retrieves the option string of a field.

Category

Field function.

Syntax

```
DBL_U8* DBL_FieldOptionStr(DBL_HTABLE hTable, DBL_S32 FieldNo);

hTable: Handle to the table
FieldNo: Option field whose option string is to be retrieved
```

Remarks

DBL_FieldOptionStr returns a string that contains the options for a specified option field. Only fields of type DBL_Type_O32 have an option string. The string consists of a comma-separated list of the valid options for the field that match the valid values of the option field.

If no field exists with the number specified in FieldNo or if the field is not of the DBL_Type_O32 type, the function will raise an exception.

This function cannot be used to change the option string of the field, only to retrieve it. Option strings can only be modified in C/SIDE.

For more information about fields and option strings, see the *Application Designer's Guide*.

Example

```
DBL_HTABLE hTable;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);

/* Retrieve the option string for field 6 in table 15         */
/* Causes an exception if field 6 does not exist in table 15  */
/* Causes an exception if field 6 is not of type DBL_Type_O32 */
printf("Option string is %s\n", DBL_FieldOptionStr(hTable, 6));

DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_FieldSize()

Function

Retrieves the size of a specified field.

Category

Field function.

Syntax

```
DBL_S16 DBL_FieldSize(DBL_HTABLE hTable, DBL_S32 FieldNo);

hTable: Handle to the table
FieldNo: Number of the field whose size will be retrieved
```

Remarks

DBL_FieldSize returns the size of the field specified in FieldNo in bytes. If no field exists with the number specified in FieldNo, the function will raise an exception.

For more information about the C/SIDE field types and their sizes, see the *Application Designer's Guide* and Appendix A.

Example

```
DBL_HTABLE hTable;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);

/* Retrieve information about field 3 in table 15           */
/* Causes an exception if field 3 does not exist in table 15 */
printf("Field name   is %s\n", DBL_FieldName(hTable, 3));
```

```
printf("Field type   is %d\n", DBL_FieldType(hTable, 3));
printf("Field size   is %d\n", DBL_FieldSize(hTable, 3));

DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_FieldType()

Function

Retrieves the type of a specified field.

Category

Field function.

Syntax

```
DBL_U16 DBL_FieldType(DBL_HTABLE hTable, DBL_S32 FieldNo);

hTable: Handle to the table
FieldNo: Number of the field whose type will be retrieved
```

Remarks

DBL_FieldType returns the type of the field specified in FieldNo. If no field exists with the number specified in FieldNo, the function will raise an exception.

C/SIDE supports the following data types: option, boolean, integer, biginteger, decimal, text, code, date, time, BLOB, datetime, binary, dateformula, duration, GUID.

For more information about the C/SIDE data types, see the *Application Designer's Guide*. The sizes of the C/SIDE data types are listed in Appendix A.

Example

```
DBL_HTABLE hTable;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);

/* Retrieve information about field 3 in table 15           */
/* Causes an exception if field 3 does not exist in table 15 */
printf("Field name   is %s\n", DBL_FieldName(hTable, 3));
printf("Field type   is %d\n", DBL_FieldType(hTable, 3));
printf("Field size   is %d\n", DBL_FieldSize(hTable, 3));

DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_FindRec()

Function

Locates a record and copies it to a buffer.

Category

Record function.

Syntax

```
DBL_BOOL DBL_FindRec(DBL_HTABLE hTable, DBL_HREC hRec, DBL_U8* SearchMethod);

hTable: Handle to the table.
hRec: As input: Record from which the search will begin.
As output: Record that is found. Any FlowFields
associated with the record are set to zero; use
DBL_CalcFields to update these fields.

SearchMethod: A string with one or more of these operators:
=  find record equal to hRec
<  find record less than hRec
>  find record greater than hRec
-  find first record in table
+  find last record in table
NULL means the same as =
```

An operator can only occur once. The operators + and - must be used alone.

If SearchMethod contains any of the operators =, > or <, values must be assigned to all the fields of the current key and the primary key in hRec before making this call.

Remarks

DBL_FindRec retrieves the first record that meets the criteria set by SearchMethod and the scope of any filters associated with the table handle (set by DBL_SetFilter/Range). The order in which the records are scanned is determined by the current key of the table handle (set by DBL_SetCurrentKey).

The search starts from the values in the current key fields in hRec. If the current key is not the primary key, there is a chance that several records will have the same values in their current key fields. In such case, the values in the primary key fields of hRec are also used in the search.

If a record is found, 1 is returned. If a record is not found, two things can happen:

1. If this result is allowed by DBL_Allow(DBL_Err_RecordNotFound), 0 is returned.
2. If the result is not allowed, the function will raise an exception and call the exception handler with an error.

For more information about exceptions, see Handling Errors and Exceptions on page 18.

Example

```
DBL_HTABLE hTable;
DBL_HREC   hRec;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
```

```
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

/* Look up record matching hRec                          */
/* Causes an exception if hRec does not exist in table 15 */
DBL_AssignField(hTable,hRec,1,DBL_FieldType(hTable,1),"100",
strlen("100"));
DBL_FindRec(hTable, hRec, (DBL_U8*)"=");

/* Look up record equal to or greater than hRec        */
/* Causes an exception if such a record does not exist */
DBL_AssignField(hTable,hRec,1,DBL_FieldType(hTable,1),"100",
strlen("100"));
DBL_FindRec(hTable, hRec, (DBL_U8*)"=>");

/* Look up first record                    */
/* Causes an exception if table 15 is empty */
DBL_FindRec(hTable, hRec, (DBL_U8*)"-");

/* Look up last record                     */
/* Causes an exception if table 15 is empty */
DBL_FindRec(hTable, hRec, (DBL_U8*)"+");

DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_FreeRec()

Function

Removes a record buffer.

Category

Record function.

Syntax

```
void DBL_FreeRec(DBL_HREC hRec);
hRec: Record buffer to remove
```

Remarks

DBL_FreeRec frees the memory occupied by the specified record buffer that was previously allocated by a call to DBL_AllocRec. After this operation, hRec is no longer a valid buffer.

A call to DBL_Exit will automatically remove all the record buffers.

Example

```
DBL_HTABLE hTable;
DBL_HREC   hRec;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
```

```
DBL_OpenTable(&hTable, 15);

 /* Allocate memory for record buffer */
hRec = DBL_AllocRec(hTable);

/* ... */

/* Free memory occupied by record buffer */
DBL_FreeRec(hRec);

DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_GetCurrentKey()

Function — Retrieves the key that is currently assigned to a table.

Category — Key function.

Syntax

```
DBL_S32* DBL_GetCurrentKey(DBL_HTABLE hTable);

hTable: Handle to the table
```

Remarks

DBL_GetCurrentKey returns a pointer to a list of the fields that make up the current key. The field list will be the same as the key that has been set by a prior call to DBL_SetCurrentKey. If DBL_SetCurrentKey has not been called, the current key is the primary key.

For more information about keys, see the *Application Designer's Guide*.

Example

```
DBL_HTABLE hTable;
DBL_HREC   hRec;
DBL_S32    Key[DBL_MaxFieldsPerKey+1];
DBL_S32    *Field;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

/* Select key on field 2 as the current key for the hTable */
/* Causes an exception if there is no key on field 2       */
Key[0] = 2;
Key[1] = 0;
DBL_SetCurrentKey(hTable, Key);

printf("The current key on hTable contains these fields:\n");
for (Field = DBL_GetCurrentKey(hTable); *Field; Field++)
```

```
printf("%d\n", *Field);

/* Scan all records sorted by field 2 in ascending order */
DBL_Allow(DBL_Err_RecordNotFound);
if (DBL_FindRec(hTable, hRec, (DBL_U8*)"-"))
do {
/* Process records */
} while (DBL_NextRec(hTable, hRec, 1));

DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_GetDatabaseName()

Function — Returns information about whether or not a database is open, and if it is open, its name.

Category — Database function.

Syntax

```
DBL_BOOL DBL_GetDatabaseName(DBL_U8* DatabaseName);

DatabaseName: Name of the database, if a database is open
```

Remarks — DBL_GetDatabaseName is called before opening a database to test if it is already open. This function is used with DBL_OpenDatabase, DBL_ConnectServer or DBL_ConnectServerandOpenDatabase.

Example 1

```
DBL_U8 DatabaseName[256];

DBL_Init();

/* Connect to a running database if there is one. */
/* If no database is running, use 2000 KB cache.  */
DBL_OpenDatabase(NULL, 2000, 0);

if (!DBL_GetDatabaseName(DatabaseName))
{
strcpy(DatabaseName,"test.fdb");
/* There is no database running. */
/* Connect to database.          */
DBL_OpenDatabase(DatabaseName, 0, 0);
}

printf("Connected to database: %s\n",DatabaseName);

/* Close database.*/
DBL_CloseDatabase();
```

```
DBL_Exit();
```

Example 2

```
DBL_U8       DatabaseName[256];

DBL_Init();

/* Connect to a running server if there is one. */
/* If no server is running, use 2000 KB cache.  */
DBL_ConnectServer("TestServer", "netb");

if !DBL_GetDatabaseName(DatabaseName)
{
strcpy(DatabaseName,"test.fdb");
/* There is no database running*. */
/* Connect to database.           */
DBL_OpenDatabase(DatabaseName, 0, 0);
}

printf("Connected to database: %s\n",DatabaseName);

/* Disconnect Server */
DBL_DisconnectServer();

DBL_Exit();
```

DBL_GetFieldData()

Function

Retrieves data from a field.

Category

Field function

Syntax

```
DBL_S32 DBL_GetFieldData(void *Dst, DBL_S32 DstSize, DBL_HTABLE hTable,
DBL_HREC hRec, DBL_S32 FieldNo);

Dst: pointer to the destination of the field data
DstSize: amount of data to retrieve, in bytes
hTable: handle to the table
hRec: handle to the record to retrieve field data from
FieldNo: number of the field to retrieve data from
```

Remarks

DBL_GetFieldData is the preferred way to retrieve data from a field in a record, in order to store it in a variable for further processing. Dst is designed as a void pointer, and can therefore be a pointer to any data type.

If the value of DstSize is less than the size of the data in the field specified with FieldNo in the record of the hTable table referenced by the hRec handle, the function will raise an exception. Therefore, you must be sure that Dst has enough space to contain the data it will receive.

DBL_GetFieldData returns the number of bytes actually retrieved – which may be less than DstSize.

Example

```
DBL_HTABLE hTable;
DBL_HREC hRec;
char s[100];
int i;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

/* Retrieve a record: */
DBL_FindRec(hTable,hRec,(DBL_U8*)"-")
/* Retrieve contents of field 10: */
i = DBL_GetFieldData(s,100,hTable,hRec,10);
/* Print out the result of the operation: */
printf("Contents of field 10: %s (%d bytes)\n",s,i);

DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_GetFieldDataAddr()

Function

Retrieves a pointer to the data in a field.

Category

Field function

Syntax

```
void* DBL_GetFieldDataAddr(DBL_HTABLE hTable, DBL_HREC hRec, DBL_S32
FieldNo);

hTable: handle to the table
hRec: handle to the record of the field
FieldNo: the number of the field to retrieve the data address for
```

Remarks

DBL_GetFieldDataAddr is designed to return a void pointer to the data contained in the field with the number specified in FieldNo, in the record specified by hRec, and where hTable is a handle to the table. In the application you must cast the pointer that is returned by DBL_GetFieldDataAddr to be of the data type defined for that field.

Example

```
DBL_HTABLE hTable;
DBL_HREC hRec;
DBL_DATE *pDate;

DBL_Init();
```

```
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

/* Retrieve a record: */
DBL_FindRec(hTable,hRec,(DBL_U8*)"-")
/* Retrieve a pointer to data of field 5 */
pDate = (DBL_DATE*)DBL_GetFieldDataAddr(hTable,hRec,5);
/* ... */

DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_GetFieldDataSize()

Function

Retrieves the data size of a field.

Category

Field function.

Syntax

```
DBL_S32 DBL_GetFieldDataSize(DBL_HTABLE hTable, DBL_S32 FieldNo, const void
*FieldVal);

hTable: handle to the table
FieldNo: number of the field to retrieve the data size for
FieldVal: NULL pointer, or pointer to data in field
```

Remarks

DBL_GetFieldDataSize returns the data size of a field. In most cases, this will be the same as the size of the field that was defined in C/SIDE. However, for fields with the (C/SIDE) data types Text and Code, the actual size of the data in a field may be less than the defined size, and the actual size of a BLOB field will always be different from the defined field size – which is 0(zero) – when the BLOB field has a content (BLOB fields are stored in a dedicated area of the database).

For these three data types, the number returned depends upon the FieldVal parameter. If FieldVal is a NULL pointer, the defined size is returned. If FieldVal is a pointer to the data, the actual size is returned (as retrievable by DBL_GetFieldDataAddr). For all the other data types, it doesn't matter whether FieldVal is a NULL pointer or a pointer to the field data.

Example 1

```
DBL_HTABLE hTable;
DBL_S32 i;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
```

```
/* Get the defined size of field 5: */
i = DBL_GetFieldDataSize(hTable,5,NULL);
/* Print out the size: */
printf("The defined size of field 5 is %d\n",i);

DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

Example 2

```
DBL_HTABLE hTable;
DBL_HREC hRec;
DBL_S32 i, j;
DBL_U8 *Text;
DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

/* Get the defined size of field 10: */
i = DBL_GetFieldDataSize(hTable,10,NULL);
/* Retrieve a specific record from the table: */
DBL_FindRec(hTable,hRec,(DBL_U8*)"-")
/* Retrieve a pointer to the data of field 10 */
Text = (DBL_U8*)DBL_GetFieldDataAddr(hTable,hRec,10);
/* Get the actual size of the data in field 10: */
j = DBL_GetFieldDataSize(hTable,10,Text);

/* Print out the sizes: */
printf("Sizes of field 10 are:\n");
printf("Defined: %d\n",i);
printf("Actual:  %d\n",j);

DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_GetFilter()

Function — Retrieves the filter that has been set for a given field.

Category — Filter function.

Syntax

```
DBL_S32 DBL_GetFilter(DBL_HTABLE hTable, DBL_S32 FieldNo,
DBL_U8* FilterStr, DBL_S16 FilterStrSize);

hTable: Handle to the table
```

```
FieldNo: Number of the field whose filter will be retrieved
FilterStr: A string variable to hold filter information
FilterStrSize: Size of FilterStr, including the terminating zero
```

If the contents of the filter cannot fit in FilterStr, an exception is raised.

Remarks

DBL_GetFilter returns the filter expression that has been set by DBL_SetFilter or DBL_SetRange for the field specified by FieldNo.

Example

```
DBL_HTABLE hTable;
DBL_HREC   hRec;
DBL_S16    MaxValue;
DBL_U8     FilterStr[100];

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

/* Set filter on field 3 */
MaxValue = 500;
DBL_SetFilter(hTable, 3, ">=200&<=%1", &MaxValue, NULL);

/* Retrieve filter on field 3 */
DBL_GetFilter(hTable, 3, FilterStr, sizeof(FilterStr));
/* Variable FilterStr now contains the string ">=200&<=500" */

printf("Current filter on field 3 is %s\n", FilterStr);

/* Scan records with a value in field 3 in the range 200 - 500 */
DBL_Allow(DBL_Err_RecordNotFound);
if (DBL_FindRec(hTable, hRec, (DBL_U8*)"-"))
do {
/* Process records */
} while (DBL_NextRec(hTable, hRec, 1) != 0);

DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_GetLastErrorCode

Function

Retrieves the code of the last error.

Category

Error function.

Syntax

```
DBL_S32 DBL_GetLastErrorCode(void);
```

Remarks

DBL_GetLastErrorCode returns the code number of the last error that you recieved.

Example

```
DBL_HTABLE hTable;
DBL_HREC hRec;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

DBL_BWT();

DBL_FindRec(hTable,hRec,(DBL_U8*)"-")

DBL_DeleteRec(hTable, hRec);

/* Delete record again to provoke error */
DBL_DeleteRec(hTable, hRec);

if(DBL_GetLastErrorCode()!=0)
{
/*...record not found...*/
}

DBL_EWT();

DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_GetRange()

Function

Retrieves the values of a range filter for a field.

Category

Filter function.

Syntax

```
void DBL_GetRange(DBL_HTABLE hTable, DBL_S32 FieldNo,
void* MinValue, void* MaxValue);

hTable: Handle to the table
FieldNo:Number of the field whose range filter is to be
retrieved
MinValue:If set to NULL, then MinValue is not returned
MaxValue:If set to NULL, then MaxValue is not returned
```

Remarks

DBL_GetRange returns the start and end values of the range filter for the field specified by FieldNo. The values are returned in MinValue and/or MaxValue, respectively. These variables must have the same size as FieldNo.

If either value is undefined, any attempt to return them will cause an exception. Set the undefined value to NULL to prevent an exception being caused.

DBL_GetRange can retrieve only a single interval, for example:

```
>=5 & <=8
```

Setting a range that is more complex than this will raise an exception.

Example

```
DBL_HTABLE hTable;
DBL_S16    MinValue;
DBL_S16    MaxValue;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);

/* Set range filter on field 3                    */
/* Equals DBL_SetFilter(hTable, 3, ">=200&<=500") */
MinValue = 200;
MaxValue = 500;
DBL_SetRange(hTable, 3, &MinValue, &MaxValue);

/* Retrieve range limits from current filter on field 3 */
DBL_GetRange(hTable, 3, &MinValue, &MaxValue);
/* Retrieve range minimum limit from current filter on field 3* /
DBL_GetRange(hTable, 3, &MinValue, NULL);
/* Retrieve range maximum limit from current filter on field 3 */
DBL_GetRange(hTable, 3, NULL, &MaxValue);

/**********/

/* Set one value filter on field 3         */
/* Equals DBL_SetFilter(hTable, 3, "=200") */
MinValue = 200;
DBL_SetRange(hTable, 3, &MinValue, NULL);

/* Retrieve range limits from current filter on field 3 */
DBL_GetRange(hTable, 3, &MinValue, &MaxValue);
printf("MinValue and MaxValue now both contain the value 200\n");

/**********/

/* Remove filter on field 3            */
/* Equals DBL_SetFilter(hTable, 3, "") */
DBL_SetRange(hTable, 3, NULL, NULL);

/* Any call to DBL_GetRange(hTable, 3, ...) now causes an  */
/* exception                                               */

DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_GetVersion()

Function — Retrieves the version number of the C/FRONT library functions.

Category — Database function

Syntax

```
DBL_S32 DBL_GetVersion(void);
```

Remarks — This function returns the version number of the C/FRONT library.

Example

```
printf("The C/FRONT library version is %d\n", DBL_GetVersion());
```

DBL_HMST_2_Time()

Function — Converts units of time to a TIME variable.

Category — Conversion function.

Syntax

```
void DBL_HMST_2_TIME(DBL_TIME* Time, DBL_S32 h, DBL_S32 m,
 DBL_S32 s, DBL_S32 t);

Time: TIME variable that will receive the converted value
h: Value for the hours
m: Value for the minutes
s: Value for the seconds
t: Value for the thousandths of a second
```

Remarks — DBL_HMST_2_TIME combines discrete values for hours, minutes, seconds and thousandths of a second to create a TIME variable. The range of the input values is not checked – this is your responsibility!

Example

```
DBL_TIME Time;
DBL_S32 h,m,s,t;

DBL_Init();

DBL_HMST_2_TIME(&Time, 14, 23, 30, 1);
/* Variable Time now contains the time 14:23:30.1 */

DBL_TIME_2_HMST(&h, &m, &s, &t, Time);
/* Variables h,m,s and t now contain 14, 23, 30 and 1 */
printf("h, m, s and t now contain %d, %d, %d and %d\n",h,m,s,t);

DBL_Exit();
```

DBL_Init()

Function — Initializes the library.

Category — Initializing function.

Syntax

```
DBL_S32 DBL_Init(void);
```

Remarks — DBL_Init initializes (opens) the library by creating and initializing internal buffers and tables. Initialization includes loading the Navision DLL-modules used by the library. An application must therefore call DBL_Init before it calls any other library function.

If the library is successfully initialized, 0 is returned. If initialization fails, an error code is returned. This is the only library function that returns an error code.

For more information about errors and exceptions, see Handling Errors and Exceptions on page 18.

Once it has been successfully called, DBL_Init cannot be called again until you have closed the library by calling DBL_Exit.

Example

```
void main(int argc, char* argv[], char* envp[])
{
DBL_Init();

/* A database can now be opened or      */
/* a connection to a server established */

DBL_Exit();
```

DBL_InitRec()

Function — Initializes a record.

Category — Record function.

Syntax

```
void DBL_InitRec(DBL_HTABLE hTable, DBL_HREC hRec);

hTable: Handle to the table
hRec: Handle to the record to be initialized
```

Remarks — DBL_InitRec assigns default values to each field in the record buffer. The values correspond to those that were defined when the table was created in C/SIDE. Fields for which no values were defined are assigned zero values.

After this operation has been performed, you are free to change the values in any or all of the fields before calling DBL_InsertRec to enter the record in the table. Ensure

that the field(s) which make up the primary key contain values that make the contents of the total primary key unique. If the contents of the total primary key are not unique, the database manager will reject the record.

Example

```
DBL_HTABLE hTable;
DBL_HREC   hRec;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

DBL_BWT();

/* Initialize all fields in hRec */
DBL_InitRec(hTable, hRec);

/* Assign value to field 1 in hRec */
DBL_AssignField(hTable,hRec,1,DBL_FieldType(hTable,1),"100",
strlen("100"));

/* Insert hRec into table 15 */
DBL_InsertRec(hTable, hRec);

DBL_EWT();

DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_InsertRec()

Function

Inserts a record into a table.

Category

Record function.

Syntax

```
DBL_BOOL DBL_InsertRec(DBL_HTABLE hTable, DBL_HREC hRec);

hTable: Handle to the table
hRec: Handle to the record to be inserted. hRec itself does not change.
```

Remarks

DBL_InsertRec inserts a record into an open table. Use DBL_InitRec to initialize the record before assigning values to the fields. The current key and any filters that have been placed on the table handle do not effect this operation.

A record is uniquely identified by the values of the fields in the primary key. The C/SIDE database manager inspects the primary keys in the table before inserting the new record.

If the record is successfully inserted, 1 is returned. If a record with the same value in the primary key already exists in the table, the insertion will fail and two things can happen:

1 If this result is allowed by DBL_Allow(DBL_Err_RecordExists), 0 is returned.

2 If the result is not allowed, the function will raise an exception and call the exception handler with an error.

For more information about errors and exceptions, see Handling Errors and Exceptions on page 18.

Example

```
DBL_HTABLE hTable;
DBL_HREC   hRec;

DBL_Init();
DBL_OpenDatabase("test.db", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

DBL_BWT();

/* Initialize all fields in hRec */
DBL_InitRec(hTable, hRec);

/* Assign value to field 1 in hRec */
DBL_AssignField(hTable,hRec,1,DBL_FieldType(hTable,1),"100",
strlen("100"));

/* Insert hRec into table 15 */
DBL_InsertRec(hTable, hRec);

DBL_EWT();

DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_KeyCount()

Function — Counts the keys that are available for a table.

Category — Key function.

Syntax

```
DBL_S16 DBL_KeyCount(DBL_HTABLE hTable);

hTable: Handle to the table
```

Remarks — DBL_KeyCount returns the number of keys that have been defined for a table. A table always has one primary key and can have one or more secondary keys. Therefore, DBL_KeyCount always returns a number greater than or equal to one. Only active keys are counted.

For more information about keys, see the *Application Designer's Guide*.

Example

```
DBL_HTABLE hTable;
DBL_S32   *Key,Field;
DBL_S16    i;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);

printf("Table 15 contains %d active key(s)\n",
DBL_KeyCount(hTable));

i = 0;
for (Key = NULL; Key = DBL_NextKey(hTable, Key); )
{
i++;
printf("Key %d contains these field number(s):\n", i);

for (Field = Key; *Field; Field++)
printf("%d\n", *Field);
}

DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_KeySumFields()

Function — Returns the SumIndexFields of a specified table key.

Category — Key function.

Syntax

```
DBL_S32* DBL_KeySumFields(DBL_HTABLE hTable, DBL_S32* Key);

hTable: Handle to the table
Key:Key whose list of SumIndexFields is to be retrieved; this
parameter may represent either a primary or secondary
key
```

Remarks — DBL_KeySumFields retrieves a list of the SumIndexFields for a given key and for a given table. SumIndexFields are a special C/SIDE feature that permits speedy access to numeric amounts, even in tables that contain thousands of records.

For more information about keys and SumIndexFields, see the *Application Designer's Guide*.

Example

```
DBL_HTABLE hTable;
DBL_S32    Key[DBL_MaxFieldsPerKey+1];
DBL_S32    *Field;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);

Key[0] = 2; /* Field number 2 */
Key[1] = 0; /* Key terminator */

printf("The key on field 2 contains the following SumIndexFields:\n");
for (Field = DBL_KeySumFields(hTable, Key); *Field; Field++)
printf("%d\n", *Field);

DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_LoadLicenseFile()

Function — Loads a license file.

Category — Database function.

Syntax

```
void DBL_LoadLicenseFile(DBL_U8* FileName);
```

```
FileName: license file to load.
```

Remarks

DBL_LoadLicenseFile loads the license file specified by FileName. This function must be called before establishing a connection to a database – otherwise, the current connection will be closed.

DBL_LockTable()

Function

Locks a table.

Category

Table function.

Syntax

```
void DBL_LockTable(DBL_HTABLE hTable, DBL_U32 Mode);

hTable: Handle to the table
Mode: DBL_LockWait or DBL_LockNoWait
```

Remarks

DBL_LockTable locks a table to prevent conflicting write operations. You can specify either a Wait or NoWait lock for the Mode parameter:

- DBL_LockWait

  If another application is carrying out a transaction on the table when you issue this lock, the function suspends your operations and waits until the table is available before returning.

- DBL_LockNoWait

  If another application is carrying out a transaction on the table when you issue this lock, the function raises an exception and calls the exception handler.

The C/SIDE database system uses table locking to ensure data integrity. Whenever an application begins to change data in a table (with InsertRec / ModifyRec / DeleteRec), the table is automatically locked. The lock prevents all other applications from changing data in the same table and remains active until the write transaction is ended (or aborted) with DBL_EWT or DBL_AWT.

Table locking does not prevent any authorized users from gaining read access to the table.

Because all write operations automatically lock the table in question, a call to DBL_LockTable would seem to be unnecessary. Imagine, however, a transaction in which an application wants to inspect data before possibly (though not necessarily) changing it – and have a guarantee that the data it changes has not been altered since it was read. The solution is to lock the table before reading, thereby ensuring that no other application can change the data between your reading the data and performing the possible write transaction.

The C/SIDE database system provides deadlock detection. Let us say that two applications, A and B, simultaneously want to lock the same two tables but in reverse order. Thus A locks Table 1 and waits to lock Table 2, while B locks Table 2 and waits to lock Table 1. This potentially fatal situation is called a deadlock and is automatically detected by the C/SIDE database manager. One of the applications will raise an exception and be terminated, while the other will be allowed to continue.

DBL_LockTable is only allowed within a DBL_BWT/DBL_EXT construction.

Example

```
DBL_HTABLE hTable;
DBL_HREC   hRec;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

DBL_BWT();

/* Prevent other users from modifying table 15 */
DBL_LockTable(hTable, DBL_LockWait);

/* Retrieve hRec from table 15 */
DBL_FindRec(hTable, hRec, (DBL_U8*)"-");

/* Assign value to field 2 in hRec */
DBL_AssignField(hTable,hRec,2,DBL_FieldType(hTable,2),"Name",
strlen("Name"));

/* Modify hRec in table 15 */
DBL_ModifyRec(hTable, hRec);

/* Commit write transaction and remove table lock */
DBL_EWT();

DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_Login()

Function

Logs a user into a database.

Category

Database function.

Syntax

```
DBL_BOOL DBL_Login(DBL_U8* UserID, DBL_U8* PassWord);

UserID: Login name (see DBL_MaxUserIDLen in cf.h)
PassWord: Password belonging to UserID
```

```
(see DBL_MaxPassWordLen in cf.h)
```

Remarks

DBL_Login plays a double role: it controls access to tables in a multiuser environment, and it provides password protection for user verification. The function must be called if more than one user (DBL_UserCount > 0) is permitted to open the database tables.

A successful login, however, does not ensure access to the data. Anyone can open a database and then, after logging in, inspect table descriptions, but gaining access to data tables requires further verification. The security mechanisms in C/SIDE allow any or all tables to be protected against unauthorized users. A typical user, for example, may only have read access to the tables in one company, may be able to read and write to the tables in another company and have no access at all to the tables in a third company. DBL_Login returns 1 for a successful login and 0 for an unsuccessful login.

User IDs are stored in an internal table in the database. The contents of this table cannot be changed by any of the functions in the library, although the current User ID can be retrieved by DBL_UserID. User IDs can only be created, modified and deleted within C/SIDE.

Example

```
DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);

/* Test whether login is required */
if (DBL_UserCount() > 0)
{
/* Log into database */
DBL_Login("MyUserID", "MyPassWord");

printf("Current userid is %s\n", DBL_UserID());
}

/* ... */

DBL_CloseDatabase();
DBL_Exit();
```

DBL_ModifyRec()

Function

Modifies a record in a table.

Category

Record function.

Syntax

```
DBL_BOOL DBL_ModifyRec(DBL_HTABLE hTable, DBL_HREC hRec);

hTable: The handle to the table
hRec: The handle to the record that is to be modified. hRec itself does not
change.
```

Remarks

DBL_ModifyRec modifies a record in the table. The record to be modified is the one identified by the values in the primary key fields in hRec. The current key and any filters that have been placed on the table handle have no effect on this operation.

In a multiuser environment, another application can modify the record in the table in the interval between your reading the record and your attempting to modify it. The C/SIDE database system automatically detects such an event, causing DBL_ModifyRec to raise an exception.

To prevent this from happening, use DBL_LockTable to lock the table before reading the record. Remember, however, that the table will be locked for the entire time that elapses between reading the the record and modifying it, and other users will therefore be unable to access it.

For more information about table locking, see the *Application Designer's Guide*.

Note that the library does not support range and validity checks – it is your responsibility to verify that the data you are inserting is valid.

If the record is successfully modified, 1 is returned. If the record is not found in the table, two things can happen:

1. If this result is allowed by the function BL_Allow(DBL_Err_RecordNotFound), 0 is returned.
2. If the result is not allowed, the function will raise an exception and call the exception handler with an error.

For more information about exceptions and errors, see Handling Errors and Exceptions on page 18.

Example

```
DBL_HTABLE hTable;
DBL_HREC   hRec;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

DBL_BWT();

/* Prevent other users from modifying table 15 */
DBL_LockTable(hTable, DBL_LockWait);

/* Retrieve hRec from table 15 */
DBL_FindRec(hTable, hRec, (DBL_U8*)"-");

/* Assign value to field 2 in hRec */
DBL_AssignField(hTable,hRec,2,DBL_FieldType(hTable,2),"Name",
strlen("Name"));

/* Modify hRec in table 15 */
DBL_ModifyRec(hTable, hRec);
```

```
/* Commit write transaction and remove table lock */
DBL_EWT();

DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_NextCompany()

Function

Retrieves the company name that comes after a specified company name.

Category

Database function.

Syntax

```
DBL_U8* DBL_NextCompany(DBL_U8* CompanyName);

CompanyName: Name of a company, or a NULL pointer. Both this
parameter and the result returned by the function are pointers
to strings.
```

Remarks

DBL_NextCompany returns the company name that follows CompanyName. You can scan all the company names in a database, by executing DBL_NextCompany in a loop. If you call this function with CompanyName set to NULL, the first company name in the database is returned. If you then call DBL_NextCompany using this result as an argument, the function returns the second company name, and so on, until the entire list has been scanned. When the end of the list is reached, the function returns a NULL pointer.

The company is neither opened or closed when this function is called.

Example

```
DBL_U8* CompName;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);

printf("The database contains the following companies:\n");
for (CompName = NULL; CompName = DBL_NextCompany(CompName);)
{
printf("%s\n", CompanyName);
}

DBL_CloseDatabase();
DBL_Exit();
```

DBL_NextField()

Function — Retrieves the field number that comes after the specified field number in a table.

Category — Field function.

Syntax

```
DBL_S32 DBL_NextField(DBL_HTABLE hTable, DBL_S32 FieldNo);

hTable: Handle to the table
FieldNo: A field number. If set to zero, the first field number in the record
is retrieved.
```

Remarks

DBL_NextField returns the field number of the field that comes after FieldNo. If the function reaches the end of the field number list, zero is returned.

DBL_NextField can scan the entire list of fields in a table. The scan is restricted to the active fields and excludes the inactive ones.

For more information about active and inactive fields, see the *Application Designer's Guide*.

Example

```
DBL_HTABLE hTable;
DBL_S32    FieldNo;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("test Company");
DBL_OpenTable(&hTable, 15);

printf("Table 15 contains %d active field(s)\n",
DBL_FieldCount(hTable));

printf("These fields are numbered as follows:\n");
for (FieldNo = 0; FieldNo = DBL_NextField(hTable, FieldNo);)
printf("%d\n", FieldNo);

DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_NextKey()

Function — Retrieves the table key that comes after a specified table key.

Category — Key function.

Syntax

```
DBL_S32* DBL_NextKey(DBL_HTABLE hTable, DBL_S32* Key);
```

```
hTable: Handle to the table
Key: A table key or a NULL pointer
```

Remarks

DBL_NextKey returns the key that comes after the specified key in the table. Both the parameter "Key" and the function result are pointers to zero-terminated char arrays that contain the numbers of the fields comprising a key.

This function does not influence the definition or selection of the keys. Table keys are defined in C/SIDE.

If Key is set to NULL, the first key for the table is returned. The first key is the primary key. If DBL_NextKey is then called using the first key as the argument, the second key is returned, and so on. When the function reaches the end of the key list, it returns a NULL pointer. In this way, DBL_NextKey can scan the entire list of table keys. The scan is restricted to the active keys and excludes the inactive ones.

For more information about keys, see the *Application Designer's Guide*.

Example

```
DBL_HTABLE hTable;
DBL_S32    *Key;
DBL_S32    *Field;
DBL_S16    i;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);

printf("Table 15 contains %d active key(s)\n",
DBL_KeyCount(hTable));

i = 0;
for (Key = NULL; Key = DBL_NextKey(hTable, Key); )
{
i++;
printf("Key %d contains the following field number(s):\n", i);
for (Field = Key; *Field; Field++)
printf("%d\n", *Field);
}

DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_NextRec()

Function

Steps through a specified number of records to retrieve a record.

Category

Record function.

Syntax

```
DBL_S16 DBL_NextRec(DBL_HTABLE hTable, DBL_HREC hRec, DBL_S16 Step);

hTable: Handle to the table
hRec: As input: Record from which the search will begin.
As output: Record that is found. Any FlowFields
associated with the record are set to zero; use
DBL_CalcFields to update these fields.
Step: Number of steps. If Step=0, the function has no effect.
For backward movement through the table, use a negative
number.
```

Remarks

DBL_NextRec locates a record that is positioned a given number of steps before or after hRec. Movement through the table is governed by the filters and by the current key that is associated with the table handle. All of the fields in hRec which will be compared to the current key must contain relevant values before this function is called.

The function returns the number of records that have been scanned which meet the criteria set by any filters, given the current key. This value can be closer to zero than Step, depending upon the number of records in the table. If the table is empty, zero is returned and hRec remains unchanged.

Example

```
DBL_HTABLE hTable;
DBL_HREC   hRec;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

/* Scan all records in table 15 in ascending order */
DBL_Allow(DBL_Err_RecordNotFound);
if (DBL_FindRec(hTable, hRec, (DBL_U8*)"-"))
do
{

} while (DBL_NextRec(hTable, hRec, 1) != 0);

/* Scan all records in table 15 in descending order */
DBL_Allow(DBL_Err_RecordNotFound);
if (DBL_FindRec(hTable, hRec, (DBL_U8*)"+"))
do
{

} while (DBL_NextRec(hTable, hRec, -1) != 0);
```

```
DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_NextTable()

Function — Returns the table number of the table that comes after a given table number.

Category — Table function.

Syntax

```
DBL_S32 DBL_NextTable(DBL_S32 TableNo);

TableNo: A number of a table. If set to zero, the first table number is
retrieved.
```

Remarks — DBL_NextTable allows you to scan all of the table numbers within a database. It returns the table number that comes after TableNo – or zero when it reaches the end of the list of table numbers.

For more information about tables, see the *Application Designer's Guide*.

Example

```
DBL_S32 TableNo;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company")

printf("The database contains the following table numbers:\n");
for (TableNo = 0; TableNo = DBL_NextTable(TableNo); )
{
printf("%d\n", TableNo);
}

DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_Oem2AnsiBuff

Function — Converts characters from OEM to ANSI.

Category — Conversion function.

Syntax

```
void DBL_Oem2AnsiBuff(const DBL_U8 *Src,DBL_U8 *Dst,DBL_S32 DstSize)
```

```
Src: the source
Dst: the destination
DstSize: the number of characters to be converted
```

Remarks

DBL_Oem2AnsiBuff converts the character buffer from OEM to ANSI. You must specify the source buffer the destination buffer and the number of characters. This function should be used in conjuction with DBL_Ansi2OemBuff because it can successfully convert the characters from OEM to ANSI and back again. The comparable Windows function does not always perform this conversion successfully.

Example

```
DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);

/* Oem buffer is allocated room for 10 characters */
DBL_U8 Oembuff[10]

/* Ansi buffer is allocated room for 5 characters */(
DBL_U8 Ansibuff[5]

/* Copy the string "Hi" to the Oem buffer */
strcpy(Oembuff, "Hi")

/* Convert the two character string from OEM to ANSI */
DBL_Oem2AnsiBuff(Oembuff, Ansibuff, 2);

DBL_CloseDatabase();
DBL_Exit();
```

DBL_OpenCompany()

Function

Opens a company in an open database.

Category

Database function.

Syntax

```
void DBL_OpenCompany(DBL_U8* CompanyName);

CompanyName: Company to open
```

Remarks

DBL_OpenCompany allows an application to select a company and thereby open tables and access data (records) from the database.

An application can have only one company open at a time, but it can have many tables open, provided they are in the same company.

In a multiuser environment, different applications can access different companies within a single database (via a server), but each application can only have one company open at a time.

If CompanyName does not exist, the function raises an exception.

For more information about companies, see the *Application Designer's Guide*.

Example

```
DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);

/* Open company                                          */
/* Causes an exception if "Test Company" does not exist */

DBL_OpenCompany("Test Company");

printf("Current company is %s\n", DBL_CompanyName());

/* Close company */
DBL_CloseCompany();

DBL_CloseDatabase();
DBL_Exit();
```

DBL_OpenDatabase()

Function

Opens a database file.

Category

Database function.

Syntax

```
void DBL_OpenDatabase(DBL_U8* DatabaseName, DBL_S32 CacheSize,
DBL_BOOL UseCommitCache);

DatabaseName: Database to open
CacheSize: Size of cache in KB
UseCommitCache: Whether to use CommitCache or not
```

Remarks

DBL_OpenDatabase opens a database with a cache of the size specified in CacheSize and loads the database manager. All succeeding calls to access the database are passed to the database manager, which executes the operations.

A cache is an area of RAM that holds the results of recent disk accesses. CacheSize specifies the amount of memory assigned to the disk cache that is used by the database manager when it accesses the database file. The size depends upon which operating system is being used. As a general rule, the larger the cache, the better the performance.

For more information about cache size and performance, see the *Installation and System Management* manual.

There are no restrictions on opening a database, but access to the tables can be governed by a password. See DBL_Login.

Close the database by calling DBL_CloseDatabase. An application can have only one database open at a time. Use DBL_CloseDatabase before opening another database. Applications can switch between a server connection and a locally-opened database

(use DBL_ConnectServerandOpenDatabase, DBL_ConnectServer and DBL_OpenDatabase); remember to close the current connection before making the switch.

If there is an error, the function raises an exception and calls the exception handler.

For more information about exceptions and errors, see Handling Errors and Exceptions on page 18.

Example

```
DBL_Init();

/* Open database using 2000 Kb cache                        */
/* Causes an exception if database test.fdb does not exist  */
/* Causes an exception if 2000 Kb cache cannot be allocated */
DBL_OpenDatabase("test.fdb", 2000, 0);

/* ... */

/* Close database */
DBL_CloseDatabase();

DBL_Exit();
```

DBL_OpenTable()

Function

Opens a table and creates a handle to it.

Category

Table function.

Syntax

```
DBL_BOOL DBL_OpenTable(DBL_HTABLE* hTablePtr, DBL_S32 TableNo);

hTablePtr: New handle to the table
TableNo: Number of the table to be opened
```

Remarks

DBL_OpenTable opens the table identified by TableNo and assigns a handle to the table. This handle can be used for future calls. The handle remains valid until DBL_CloseTable is invoked. You can create several handles to the same table. You can also use other library functions to set filters and to set a current key individually for each handle.

DBL_OpenTable will open the table even if the user (verified by DBL_Login) does not have permission to access the table. But the application will receive an error when the user tries to read or modify the data in the table. Only the table description (field and key layout) can be accessed, not the table data.

If the table is successfully opened, 1 is returned. If the table does not exist, two things can happen:

1 If this result is allowed by DBL_Allow(DBL_Err_TableNot Found), 0 is returned.

2 If the result is not allowed, the function will raise an exception and call the exception handler with an error.

For more information about tables, see the *Application Designer's Guide*.

Example

```
DBL_HTABLE hTable;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");

/* Open table                                      */
/* Causes an exception if table 15 does not exist */
DBL_OpenTable(&hTable, 15);

/* Close table */
DBL_CloseTable(hTable);

/* Open table */
DBL_Allow(DBL_Err_TableNotFound);
if (DBL_OpenTable(&hTable, 16))
{
printf("Table opened\n");

/* ... */

/* Close table */
DBL_CloseTable(hTable);
}
else
printf("Table does not exist\n");

DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_OpenTemporaryTable()

Function

Creates a temporary table.

Category

Table function.

Syntax

```
DBL_BOOL DBL_OpenTable(DBL_HTABLE* hTablePtr, DBL_S32 TableNo);

hTablePtr: New handle to the table
TableNo:The number of the table to be used as a template for the temporary
table.
```

Remarks

DBL_OpenTemporaryTable creates a temporary table based on the table description of the "real" table with the number specified in TableNo. The temporary table can be accessed like any other table by using hTablePtr.

You cannot perform transactions on a temporary table because it is not a part of the database. The temporary table does not exist outside the application that creates it. Consequently, it is also "private" for the application that creates it, and other users in a multi-user system cannot access it. Other than that, you can perform the same operations as on a "real" table.

The benefit of using a temporary table is that it is held in memory and this makes performing operations on it very fast. In a client/server environment, this also reduces the load on the network. You can copy records from the corresponding "real" table by using DBL_FindRec and DBL_NextRec. You cannot use DBL_CopyRec because the source and the destination records must be in the same table when you are using this function. When you have performed a series of operations on the records in a temporary table, you can insert these records into the "real" table by using DBL_InsertRec or DBL_ModifyRec.

DBL_RecCount()

Function

Counts the number of records in a table.

Category

Record function.

Syntax

```
DBL_S32 DBL_RecCount(DBL_HTABLE hTable);

hTable:Handle to the table
```

Remarks

DBL_RecCount returns the number of records that meet the conditions specified in any filters assigned to the table handle. If no filters are set, DBL_RecCount returns the total number of records in the table.

This operation is very quick if the table is not filtered. Filters make movement through the table slower.

Example

```
DBL_HTABLE hTable;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);

printf("Table 15 contains %d record(s)\n", DBL_RecCount(hTable));

DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_ReleaseAllObjects()

Function — Releases all of the resources in C/FRONT.

Category — Database function.

Syntax

```
void DBL_ReleaseAllObjects(void);
```

Remarks — DBL_ReleaseAllObjects releases all of the allocated resources in C/FRONT. This means that all the tables are closed and all the allocated records are released.

This function is meant to be used when you are handling errors, where it is desirable to have all of the resources released at once.

The database is not closed, and any open companies are not closed either.

If all of the resources are not released, a call to DBL_OpenDatabase or DBL_CloseDatabase will raise an exception.

DBL_SelectLatestVersion()

Function — Selects the latest data version.

Category — Database operation

Syntax

```
void DBL_SelectLatestVersion(void);
```

Remarks — DBL_SelectLatestVersion accesses the newest version of the data in a database. All subsequent database operations will be performed on this version of the data. In a single-user environment, this function has no effect, because the application always accesses the newest version – there are no other active applications creating new versions.

For more information about database versions, see the *Application Designer's Guide*.

Example

```
DBL_HTABLE hTable;
DBL_HREC   hRec;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

/* Retrieve hRec from my data version */
DBL_FindRec(hTable, hRec, (DBL_U8*)"-");
printf("hRec was found\n");
```

```
/* Another application connected to the database/server */
/* and deleted hRec                                     */
/*...
DBL_BWT();
DBL_AssignField(hTable,hRec,1,DBL_FieldType(hTable,1),"100",
strlen("100"));
DBL_DeleteRec(hTable, hRec);
DBL_EWT();
...
************************************************************/

/* Retrieve hRec again from my data version */
DBL_AssignField(hTable,hRec,1,DBL_FieldType(hTable,1),"100",
strlen("100"));
DBL_FindRec(hTable, hRec, (DBL_U8*)"=");
printf("hRec was found\n");

/* Select the latest public version to be my data version */
DBL_SelectLatestVersion();

/* Retrieve hRec from my data version */
DBL_AssignField(hTable,hRec,1,DBL_FieldType(hTable,1),"100",
strlen("100"));
DBL_Allow(DBL_Err_RecordNotFound);
if (DBL_FindRec(hTable, hRec, (DBL_U8*)"="))
printf("Record still exists\n");
else
printf("Record has been deleted by another application\n");

DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_S32_2_BCD()

Function — Converts an S32 variable to a BCD (decimal number).

Category — BCD function.

Syntax

```
void DBL_S32_2_BCD(DBL_BCD *Dest, DBL_S32 Source);

Dest: Variable in which the converted S32 variable is placed
Source: S32 variable to be converted
```

Remarks — DBL_S32_2_BCD converts Source to a BCD and places the result in Dest.

Example

```
DBL_BCD b1;
DBL_S32 s1, s2;
```

```
DBL_Init();

s1 = 31415;
DBL_S32_2_BCD(&b1, s1);
s2 = DBL_BCD_2_S32(&b1);

if (s1 != s2)
    return(-1);

DBL_Exit();
```

DBL_SetCurrentKey()

Function — Sets the current key for a table handle.

Category — Key function.

Syntax

```
DBL_BOOL DBL_SetCurrentKey(DBL_HTABLE hTable, DBL_S32* Key);

hTable: Handle to the table
Key: Desired key or a NULL pointer
```

Remarks

DBL_SetCurrentKey assigns a specified key to a table handle. The key becomes the current key and is used by DBL_FindRec, DBL_NextRec and other functions until another key is selected. Use DBL_NextKey to scan the list of keys for the table to find out which keys are available. Only active keys will be retrieved.

The primary key of the table is the current key, until DBL_SetCurrentKey is called. When a secondary key is the current key, you can make the primary key the current key again by calling DBL_SetCurrentKey with Key set to NULL.

If a new current key is successfully assigned to the table handle, 1 is returned. If the requested key does not exist, two things can happen:

1. If this result is allowed by the DBL_Allow(DBL_Err_Key NotFound), 0 is returned.
2. If the result is not allowed, the function will raise an exception and call the exception handler with an error.

For more information about exceptions and errors, see Handling Errors and Exceptions on page 18.

For more information about keys, see the *Application Designer's Guide*.

Example

```
DBL_HTABLE hTable;
DBL_HREC   hRec;
DBL_S32    Key[DBL_MaxFieldsPerKey+1];
DBL_S32    *Field;
```

```
DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

/* Select key defined for field 2 as current key for the hTable */
/* handle. Causes an exception if table 15 does not have a      */
/* key on field 2                                               */
Key[0] = 2;
Key[1] = 0;
DBL_SetCurrentKey(hTable, Key);

printf("The current key on hTable contains these fields:\n");
for (Field = DBL_GetCurrentKey(hTable); *Field; Field++)
printf("%d\n", *Field);

/* Scan all records sorted by field 2 in ascending order */
DBL_Allow(DBL_Err_RecordNotFound);
if (DBL_FindRec(hTable, hRec, (DBL_U8*)"-"))
do
{

} while (DBL_NextRec(hTable, hRec, 1));

DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_SetExceptionHandler()

Function — Installs a user-written exception handler.

Category — Exception-handling function.

Syntax

```
void* DBL_SetExceptionHandler(DBL_pFuncExceptionHandler ExceptionHandler);

ExceptionHandler: The function to be installed as the exception handler for
the library. If set to NULL, the default exception handler is restored.
```

Remarks — This function installs ExceptionHandler as the exception handler that is used by the library. DBL_SetExceptionHandler returns the address of the previous exception handler. If you want to reinstall the default exception handler later, you must store this address, and use it in a call to DBL_SetExceptionHandler. Installing another handler, for example, the default one, is also the only way to disable a customized exception handler.

Example

```
void main(DBL_S16 argc, DBL_U8 *argv[], DBL_U8 *envp[])
```

```
{
/* ... */
DBL_SetExceptionHandler(My_ExceptionHandler);
/* ... */
}

void DBL_CDECL My_ExceptionHandler(DBL_S32  ErrorCode, DBL_BOOL IsFatal)
{
char *Fatal = (IsFatal) ? " Fatal" : "";
char *dbError = (19 == (ErrorCode/0x10000L)) ? "Database " :
""; /* Module No in high word */
ErrorCode &= 0xffffL; /* Error Code in low word */
printf("Exception Handler called with%s %sError: %d.\n",
Fatal, dbError, ErrorCode);
if(IsFatal)
exit(ErrorCode);
}
```

DBL_SetFilter()

Function — Assigns a filter to a specified field.

Category — Filter function.

Syntax

```
void DBL_SetFilter(DBL_HTABLE hTable, DBL_S32 FieldNo, DBL_U8* FilterStr,
void* ValuePtr1, ...);

hTable: Handle to the table
FieldNo: Number of the field for which a filter will be set
FilterStr: Filter expression for the field, consisting of alphanumeric
characters and one or more of the following operators: <   >   ?   &   |   =

ValuePtr1..9: Replacement values for FilterStr. For example, if FilterStr
contains the parameter %4, it is replaced by the fourth
argument in this list. The list of replacement values must
end with the parameter NULL.
```

Remarks

DBL_SetFilter assigns a filter to the field specified by FieldNo. Any filter that is already assigned to FieldNo for this table handle is removed before the new filter is attached. If FilterStr is empty or contains a NULL pointer, no filter will be assigned to FieldNo, and any filter that is currently assigned will be removed.

For more information about filter syntax and relational operators, see the *Application Designer's Guide*.

Example

```
DBL_HTABLE hTable;
DBL_HREC   hRec;
DBL_S16    MaxValue;
DBL_U8     FilterStr[100];

DBL_Init();
```

```
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

/* Set filter on field 3 */
MaxValue = 500;
DBL_SetFilter(hTable, 3, ">=200&<=%1", &MaxValue, NULL);

/* Retrieve filter on field 3 */
DBL_GetFilter(hTable, 3, FilterStr, sizeof(FilterStr));
/* Variable FilterStr now contains the string ">=200&<=500" */

printf("Current filter on field 3 is %s\n", FilterStr);

/* Scan records with a value in field 3 in the range 200-500 */
DBL_Allow(DBL_Err_RecordNotFound);
if (DBL_FindRec(hTable, hRec, (DBL_U8*)"-"))
do
{

} while (DBL_NextRec(hTable, hRec, 1) != 0);

DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_SetMessageShowHandler

Function — Installs a customized message handler that will be used instead of the default message handler.

Category — Error function

Syntax

```
void* DBL_SetMessageShowHandler(DBL_pFuncMessageShowHandler ShowFunc);

ShowFunc: pointer to the function to use for showing messages.
```

Remarks — DBL_SetMessageShowHandler installs a customized function for showing messages.

Example

```
void main(DBL_S16 argc, DBL_U8 *argv[], DBL_U8 *envp[])
{
/* ... */
DBL_SetMessageShowHandler(My_MessageShowHandler);
/* ... */
}

void DBL_CDECL My_MessageShowHandler(DBL_U8 *Msg, DBL_U32 MsgType, DBL_S32
ErrorCode)
```

```
{
DBL_S32 LnWidth = 70;
do {
DBL_U8 *Ret = (DBL_U8*)memchr(Msg,'\r',strlen((char*)Msg));

if (Ret > Msg+LnWidth) {
DBL_U8 *Space = (DBL_U8*)memchr_bckwrd(Msg,LnWidth,' ');
if (Space)
Ret = Space;
}

if (Ret)
*Ret++ = 0;

printf("%s\n",Msg);
Msg = Ret;
} while(Msg && *Msg != 0);

fflush(stdout);
}

static void* memchr_bckwrd(const void *Dst, DBL_S32 Size, DBL_U8 Chr)
{
DBL_U8 *pS = (DBL_U8*)Dst+Size;
while(--pS >= Dst)
if (*pS == Chr)
return(pS);
return(NULL);
}
```

DBL_SetNavisionPath()

Function — Sets the path to the directory where Navision is installed.

Category — Initialization function.

Syntax

```
void DBL_SetNavisionPath(DBL_U8 *Path);

Path: Path to the directory containing the Navision files.
```

Remarks — Normally the C/FRONT library (cfront.dll) reads the registry in order to locate the Navision DBMS system. However, if multiple Navision Systems are installed or if Navision is not present on the system, the function DBL_SetNavisionPath should be called with the path to the directory of the Navision installation or to a directory containing the following files from a Navision installation:

dbm.dll
nc_netb.dll
nc_tcp.dll

slave.exe
fin.flf

DBL_SetNavisionPath must be called before any other function in the library, except DBL_Init, DBL_SetExceptionHandler and DBL_SetMessageShowHandler

DBL_SetRange()

Function

Sets a range filter for a field.

Category

Filter function.

Syntax

```
void DBL_SetRange(DBL_HTABLE hTable, DBL_S32 FieldNo, const void* MinValue,
const void* MaxValue);

hTable: Handle to the table
FieldNo: Number of the field for which the filter is to be set
MinValue: Starting value. If set to NULL, the filter is removed,
regardless of the contents of MaxValue.
MaxValue: Ending value. If NULL, the range is set to MinValue..MinValue.
```

Remarks

DBL_SetRange provides a quick way to set a simple filter on a field. Any filter already assigned to the field is removed.

If MaxValue is set to NULL, the range is set to MinValue alone.

Both MinValue and MaxValue must be of the same type as FieldNo.

Example

```
DBL_HTABLE hTable;
DBL_S16    MinValue;
DBL_S16    MaxValue;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);

/* Set range filter on field 3                    */
/* Equals DBL_SetFilter(hTable, 3, ">=200&<=500") */
MinValue = 200;
MaxValue = 500;
DBL_SetRange(hTable, 3, &MinValue, &MaxValue);

/* Retrieve range limits from current filter on field 3        */
DBL_GetRange(hTable, 3, &MinValue, &MaxValue);
/* Retrieve range minimum limit from current filter on field 3 */
DBL_GetRange(hTable, 3, &MinValue, NULL);
/* Retrieve range maximum limit from current filter on field 3 */
DBL_GetRange(hTable, 3, NULL, &MaxValue);

/**********/
```

```
/* Set one value filter on field 3         */
/* Equals DBL_SetFilter(hTable, 3, "=200") */
MinValue = 200;
DBL_SetRange(hTable, 3, &MinValue, NULL);

/* Retrieve range limits from current filter on field 3 */
DBL_GetRange(hTable, 3, &MinValue, &MaxValue);
printf("MinValue and MaxValue now both contain the value 200\n");

/**********/

/* Remove filter on field 3            */
/* Equals DBL_SetFilter(hTable, 3, "") */
DBL_SetRange(hTable, 3, NULL, NULL);

/* Calling DBL_GetRange(hTable, 3, ..) now causes an exception */

DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_Str_2_Alpha( )

Function — Converts a string to an ALPHA variable.

Category — Conversion function.

Syntax

```
void DBL_Str_2_Alpha(DBL_U8* Alpha, DBL_S16 AlphaSize, DBL_U8* Str);

Alpha: Variable to receive the converted string
AlphaSize: Size of Alpha, in bytes
Str: String to be converted
```

Remarks — DBL_Str_2_Alpha converts a string to a variable of the ALPHA type and stores it in Alpha. If Alpha is not long enough to hold the converted string, the function raises an exception.

For more information about ALPHA variables, see Appendix B.

Example

```
DBL_U8 Alpha[12];
DBL_U8 Str[11];

DBL_Init();

DBL_Str_2_Alpha(Alpha, sizeof(Alpha), "Number10");
/* Variable Alpha now contains the alpha value "Number10" */

DBL_Alpha_2_Str(Str, sizeof(Str), Alpha);
/* Variable Str now contains the string value "Number10"  */
```

```
printf("Variable Str now contains the string value %s\n", Str);

DBL_Exit();
```

DBL_Str_2_BCD()

Function

Converts a string to a BCD (decimal number).

Category

Conversion function.

Syntax

```
void DBL_Str_2_BCD(DBL_BCD* Bcd, DBL_U8* Str);

Bcd: BCD variable in which the converted string is placed
Str: String to be converted
```

Remarks

DBL_Str_2_BCD converts Str to a BCD variable and stores it in Bcd. Str must be in a format that prefixes negative numbers with a minus sign (-) and uses the period character as a decimal point (U.S.format). Leading and trailing blanks are ignored during conversion.

Value	String
1234	"1234"
-1234	"-1234"
1234.00	"1234"
1234.050	"1234.05"
11234.56	"1234.56"
.005	"0.005"

If Str does not contain a valid decimal number and cannot be converted, the function will raise an exception.

Example

```
DBL_BCD Bcd;
DBL_U8  Str[11];

DBL_Init();

DBL_Str_2_BCD(&Bcd, "-1.2345");
/* Variable Bcd now contains the value -1.2345 */

DBL_BCD_2_Str(Str, sizeof(Str), &Bcd);
/* Variable Str now contains the string value "-1.2345" */
printf("Variable Str now contains the string value %s\n", Str);

DBL_BCD_2_Str(Str, 6+1, &Bcd);
/* Variable Str now contains the string value "******" */
printf("Variable Str now contains the string value %s\n", Str);
```

```
DBL_Exit();
```

DBL_Str_2_Date()

Function — Converts a string to a DATE value.

Category — Conversion function

Syntax

```
void DBL_Str_2_Date(DBL_DATE *Date, DBL_U8 *Str);

Date: variable in which to place the converted value.
Str: string containing to value to convert.
```

Remarks — The Str string must conform to the format that has been set in the Regional Settings in the Control Panel. If the format is not correct, the function will raise an exception.

Example

```
DBL_U8 Str[50];
DBL_DATE cDate;

/* ... */

/* Place date as string in Str. Note that the format depends */
/* upon the Regional Setting selected for the operating system */
sprintf(Str,"07-06-96");
/* Convert Str to DATE value */
DBL_Str_2_Date(&cDate,Str);

/* ... */
```

DBL_Str_2_Time()

Function — Converts a string to a TIME value.

Category — Conversion function

Syntax

```
void DBL_Str_2_Time(DBL_TIME *Time, DBL_U8 *Str);

Time: variable in which to place the converted value
Str: string containing the value to convert
```

Remarks — The Str string must conform to the format that has been set in the Regional Settings in the Control Panel. If the format is not correct, the function will raise an exception.

Example

```
DBL_U8 Str[50];
DBL_TIME cTime;
```

```
/* ... */

/* Place time as string in Str. Note that the format depends */
/* upon the Regional Setting selected for the operating system */
sprintf(Str,"17.05.57");
/* Convert Str to TIME value */
DBL_Str_2_Time(&cTime,Str);

/* ... */
```

DBL_TableName()

Function — Retrieves the name of an open table.

Category — Table function.

Syntax

```
DBL_U8* DBL_TableName(DBL_HTABLE hTable);

hTable: Handle to a table
```

Remarks — DBL_TableName returns the name of the table to which handle hTable is bound. The handle was created and bound to the table when DBL_OpenTable was called.

This function cannot be used to change the table name. Table names can only be changed in C/SIDE.

For more information about tables, see the *Application Designer's Guide*.

Example

```
DBL_HTABLE hTable;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);

printf("Table number 15 is named %s\n", DBL_TableName(hTable));

DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_TableNo()

Function — Retrieves the number of the table with a specified table name.

Category — Table function.

Syntax

```
DBL_S32 DBL_TableNo(DBL_U8* TableName);

TableName: Table name
```

Remarks — This function is only needed by applications that do not support a table number. It is used to convert a name into a number for use by the DBL_Open Table function.

If TableName does not exist, 0 is returned.

For more information about tables, see the *Application Designer's Guide*.

Example

```
DBL_S32    TableNo;
DBL_HTABLE hTable;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");

/* Look up the number of TestTable */
TableNo = DBL_TableNo("TestTable");
if (TableNo != 0)
{
DBL_OpenTable(&hTable, TableNo);

/* ... */

DBL_CloseTable(hTable);
}
else
printf("Table does not exist\n");

DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_Time_2_HMST()

Function — Converts a TIME variable to hours, minutes, seconds, and thousandths of a second.

Category — Conversion function.

Syntax

```
void DBL_Time_2_HMST(DBL_S32* h, DBL_S32* m, DBL_S32* s, DBL_S32* t, DBL_TIME
Time);
```

```
h: Variable to receive the value for hours
m: Variable to receive the value for minutes
s: Variable to receive the value for seconds
t: Variable to receive the value for thousandths of a second
Time: TIME variable to be converted
```

Remarks

DBL_Time_2_HMST dismantles a TIME variable and creates values for hours, minutes, seconds and thousandths of a second. Any of the four output variables (h, m, s and t) can be set to NULL, if they are not needed.

If this function is called with Time=zero (undefined), an exception is raised. To prevent an exception being raised, test the value of the Time variable before you call this function.

Example

```
DBL_TIME Time;
DBL_S32  h,m,s,t;

DBL_Init();

DBL_HMST_2_Time(&Time, 14, 23, 30, 1);
/* Variable Time now contains the time 14:23:30.1 */

DBL_Time_2_HMST(&h, &m, &s, &t, Time);
/* Variables h,m,s and t now contain 14, 23, 30 and 1 */
printf("h,m,s and t now contain values %d, %d, %d and %d\n",
h,m,s,t);

DBL_Exit();
```

DBL_Time_2_Str()

Function

Converts a TIME value to a string.

Category

Conversion function.

Syntax

```
void DBL_Time_2_Str(DBL_U8 *Str, DBL_S16 StrSize, DBL_TIME Time);

Str: string in which to place the converted value
StrSize: size (in bytes) of Str
Time: variable containing the TIME value to convert
```

Remarks

DBL_Time_2_Str converts the TIME value in Time to a string. StrSize is the size of the destination string: the number of bytes to place in Str. It is your responsibility to ensure that the converted value is not truncated.

Example

```
DBL_HTABLE hTable;
DBL_HREC hRec;
DBL_U8 resStr[50];
```

```
DBL_TIME *pTime;

DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);
DBL_OpenCompany("Test Company");
DBL_OpenTable(&hTable, 15);
hRec = DBL_AllocRec(hTable);

/* Retrieve a record: */
DBL_FindRec(hTable,hSrcRec,(DBL_U8*)"-")
/* Get pTime from field 30: */
pTime = (DBL_TIME*)DBL_GetFieldDataAddr(hTable,hRec,30);
/* Convert pTime to a string: */
DBL_Time_2_Str(resStr,sizeof(resStr),*pTime);
/* Print out the string: */
printf("Time as string: %s\n", resStr);

DBL_FreeRec(hRec);
DBL_CloseTable(hTable);
DBL_CloseCompany();
DBL_CloseDatabase();
DBL_Exit();
```

DBL_UseCodeUnitsPermissions

Function — Allows you to use the permissions of a specific codeunit.

Category — Security Function.

Syntax

```
void DBL_UseCodeUnitsPermissions(DBL_S32 CodeUnitID);

CodeUnitID: the ID of the codeunit whose permissions you want to use.
```

Remarks — In order to run this function your permissions in this database must include execute permission for the codeunit whose permissions you want to use. The code unit you point to must have the property CFRONTMayUsePermissions set to Yes.

Example

```
DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);

/* use the permissions for codeunit 55000 */

DBL_UseCodeUnitsPermissions(55000);

/* ... */

DBL_CloseDatabase();
DBL_Exit();
```

DBL_UserCount()

Function — Counts the user IDs in the database.

Category — Database function.

Syntax

```
DBL_S32 DBL_UserCount(void);
```

Remarks — DBL_UserCount returns the number of users that are permitted to access the database tables. Users are created and assigned user IDs in C/SIDE.

This function is used to determine whether the function DBL_Login is needed. If at least one user is registered, all users are required to log in. The value returned by this function does not reflect how many users are currently connected.

Example

```
DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);

/* Test whether login is required */
if (DBL_UserCount() > 0)
{
/* Log into database */
DBL_Login("MyUserID", "MyPassWord");

printf("Current userid is %s\n", DBL_UserID());
}

/* ... */

DBL_CloseDatabase();
DBL_Exit();
```

DBL_UserID()

Function — Retrieves the login name of the current user.

Category — Database function.

Syntax

```
DBL_U8* DBL_UserID(void);
```

Remarks — DBL_UserID returns the User ID that was used to connect to the current database from CFRONT and accepted by DBL_Login. The returned value is a pointer to a string containing the current User ID. If no DBL_Login has been issued, the function returns a pointer to an empty string.

User IDs can only be created, modified and deleted within C/SIDE.

Example

```
DBL_Init();
DBL_OpenDatabase("test.fdb", 2000, 0);

/* Test whether login is required */
if (DBL_UserCount() > 0)
{
/* Log into database */
DBL_Login("MyUserID", "MyPassWord");

printf("Current userid is %s\n", DBL_UserID());
}

/* ... */

DBL_CloseDatabase();
DBL_Exit();
```

DBL_YMD_2_Date()

Function

Converts year, month, day to a DATE variable.

Category

Conversion function.

Syntax

```
void DBL_YMD_2_DATE(DBL_DATE* Date, DBL_S32 y, DBL_S32 m, DBL_S32 d, DBL_BOOL
Closing);

Date: DATE variable into which the converted date will be
placed
y: Value for a year in the range 0001..9999
m: Value for a month in the range 1..12
d: Value for a day in the range 1..31
Closing: 1 if Date is designated a closing date, otherwise 0
```

Remarks

DBL_YMD_2_Date constructs a DATE variable.

Example

```
DBL_DATE Date;
DBL_S32  y,m,d;
DBL_BOOL c;

DBL_Init();

DBL_YMD_2_Date(&Date, 2000, 5, 17, 0);
/* Variable Date now contains the date May 17, 2000 */

DBL_Date_2_YMD(&y, &m, &d, &c, Date);
/* Variables y,m,d and c now contain 2000, 5, 17 and 0 */
printf("y,m,d and c now contain %d, %d, %d and %d\n",
y,m,d,c);

DBL_Exit();
```

Appendix A
C/FRONT Library Specifications

This appendix contains the specifications for the C/FRONT library functions.

- C/FRONT Library Specifications

A.1 C/FRONT LIBRARY SPECIFICATIONS

The following sections describe the specifications of the C/FRONT library functions.

Type and Constant Definitions

The library offers an expanded variety of alphanumeric, numeric, boolean and string types, known collectively as the DBL_ types.

typedef unsigned char	DBL_U8;
typedef unsigned short int	DBL_U16;
typedef signed long int	DBL_S16;
typedef unsigned long int	DBL_U32;
typedef signed long int	DBL_S32;
typedef DBL_U32	DBL_BOOL;
typedef DBL_U32	DBL_DATE;
typedef DBL_U32	DBL_TIME;
typedef DBL_O32	DBL_O32
typedef DBL_TABLE*	DBL_HTABLE;
typedef struct { DBL_U8 Exp; DBL_U8 Mant[9]; DBL_U8 Slack[2] }	DBL_BCD;
typedef double	DBL_DOUBLE
typedef biginteger { DBL_U32 LowPart; DBL_S32 HighPart }	DBL_S64
typedef DBL_S64	DBL_Duration
typedef DBL_S64	DBL_Datetime
typedef struct { DBL_U32 Data1; DBL_U16 Data2; DBL_U16 Data3; DBL_U8 Data4[8] }	DBL_GUID

Field Types

Data fields in a record can use any of the following nine field types. The right hand column shows the sizes of the different field types:

#define DBL_Type_O32	/* 4 bytes */
#define DBL_Type_BOOL	/* 4 bytes */
#define DBL_Type_BCD	/* 12 bytes */
#define DBL_Type_STR	/* Max Field Len + 1 byte */
#define DBL_Type_DATE	/* 4 bytes */
#define DBL_Type_TIME	/* 4 bytes */
#define DBL_Type_ALPHA	/* Max Field Len + 2 bytes */
#define DBL_Type_S32	/* 4 bytes */
#define DBL_Type_BLOB	/* Max 2 GB*/

#define DBL_Type_S64	/* 8 bytes */
#define DBL_Type_Duration	/* 8 bytes */
#define DBL_Type_Datetime	/* 2 x 4 bytes */
#define DBL_TYPE_GUID	/* 16 bytes (4+2+2+8) */

New field types cannot be defined. The type of a field can be retrieved with the function DBL_FieldType.

Declaration of Variables

Record field variables are declared as shown in the following table. The right hand column shows how the variable is supposed to be represented in the database.

```
#define My_Max_Len 10
/* Note that it is the length of the specific field */
/* in the database. *
```

DBL_O32	Option;	/* DBL_Type_O32 */
DBL_BOOL	Boolean;	/* DBL_Type_BOOL */
DBL_BCD	Decimal;	/* DBL_Type_BCD */
DBL_U8	Text[My_Max_Len+1];	/* DBL_Type_STR */
DBL_Date	Date;	/* DBL_Type_DATE */
DBL_Time	Time;	/* DBL_Type_TIME */
DBL_U8	Code[My_Max_Len+2];	/* DBL_Type_ALPHA */
DBL_S32	Integer;	/* DBL_Type_S32 */
DBL_BLOB	BLOB;	/* DBL_Type_BLOB */
DBL_S64	BigInteger	/* DBL_Type_S64 */
DBL_Duration	Duration	/* DBL_Type_DUR */
DBL_Datetime	DateTime	/* DBL_Type_DATETIME */
DBL_GUID	GUID	/* DBL_Type_GUID */

The names of the variables describe how the corresponding field types are represented in the C/SIDE Table Designer.

Field Classes

Data fields in a record belong to one of the following three classes:

#define DBL_Class_Normal	0
#define DBL_Class_FlowField	1
#define DBL_Class_FlowFilter	2

The class of a field can be retrieved with the function DBL_FieldClass.

Other C-Library Constant Definitions

See the `cf.h` file for definitions of all the constants.

Appendix B
The Alpha Type

This appendix contains a description of the Alpha data type – an extended C string type.

B.1 Alpha Type

An Alpha type is an extended C string (zero-terminated string). The contents of the first byte differ, depending upon the contents of the string:

If the string contains only the numeric characters '0'..'9', the first byte contains the length of the string.

If the string contains non-numeric characters, the first byte contains the hex value FF.

This extra byte is used to give the correct weight to the strings when comparing two alpha type fields.

Ordinary strings

```
char a[] = "990";
char b[] = "1000";
```

When compared, a is greater than b, and is sorted as follows:

```
"1000"
"990"
```

Alpha strings

```
char a[] = "\3""990";    /* \3 is the length of 990  */
char b[] = "\4""1000";   /* \4 is the length of 1000 */
```

When compared, a is less than b, and is sorted as follows:

```
"\3""990"
"\4""1000"
```