PlugSys OXBC Data Access Guide

featuring

- ODBC Connector
- DBF Extender

Data Access Guide Version 2.00 2001-03-22a

Trademark Notice

Max™, Joiner™, Joiner™, Joiner 96™, Joiner 97™ are trademarks of PlugSys International LLC.

CodeBase™ is a trademark of Sequiter Software.

dBASE™, dBASE III Plus™, dBASE IV™, Visual dBASE™ are trademarks of Borland/ Inprise and dBASE Inc. Clipper™, CA-Visual Objects™ and CA-Clipper™ are trademarks of Computer Associates International. Internet Information Server™, MSDOS™, Windows™, Windows 95™, Windows 98™, Windows NT™, Visual Basic™, FoxPro™, Visual FoxPro™, Front Page™, Front Page Express™, Front Page 2000™, Visual Interdev™, Visual Studio™ and SQL Server™ are trademarks of Microsoft Corporation. Blinker™ is a trademark of Blink Inc. Dreamweaver™ is a trademark of Macromedia, Inc. HomeSite™ is a trademark of Allaire Corporation. Netscape Enterprise Server™ and Netscape FastTrak Server™ and iPlanet™ are trademarks of the Netscape/AOL/Sun Alliance and iPlanet.

All trademarks for products discussed in this document and not named above are those of their publisher.

ODBC Connector Language Reference 75

Introduction

This chapter provides an overview of the PlugSys Data Access Guide.

What is OXBC?

Open XBase Connectivity (OXBC) is PlugSys's strategic interface for accessing data in a heterogeneous environment of relational and nonrelational database management systems (DBMS's).

The advantage of OXBC over other interfaces such as ODBC is that OXBC is specifically designed to match Xbase language semantics. Thus, developers can use Xbase commands to manipulate all kinds of data: DBF files, text files, SQL databases.

OXBC was designed as a layer of abstraction so that other drivers could "speak directly" with the underlying data source. This allows PlugSys products to interact with drivers like the ODBC Connector (to work in conjunction with ODBC drivers for SQL databases) and DBF Extender (to work with data files in FoxPro, dBASE and Clipper formats).

OXBC is platform-independent. So you can depend on OXBC drivers for Win32 and Linux.

Implementation Issues

OXBC Driver Function Calls

Functions defined within OXBC drivers are only available ***after you load the driver***. USE **`SET DBTYPE TO`** to explicitly load a driver. This ensures that all driver-resident functions will be available. Of course, these functions become unavailable when you unload the OXBC driver.

This is especially useful when working with the ODBC Connector, when driver functions like **`SQLConnect()`** have to be called before populating a virtual DBF with **`USE`** <dbf> **`DBTYPE ODBC`**.

DBF Extender

Getting Started

The purpose of this chapter is to provide the details you will need to plan application development when you access FoxPro, dBASE or Clipper data files.

About The DBF Extender

The Max family of Xbase tools was conceived to simplify the development process. PlugSys set a goal of providing access to popular Xbase data files without adding complexity.

The DBF Extender realizes this goal by employing classic Xbase commands to open and maintain Xbase files beyond those handled by the native Max engine.

Data Sharing

The DBF Extender lets organizations run legacy applications under FoxPro, dBASE or Clipper. Using the DBF Extender, your new Max applications and Max Server Pages share the data using compatible locking.

Higher Performance and Fewer Limits

For applications where you are not restricted by the constraints of legacy applications, the DBF Extender gives you access to larger file sizes, record sizes, etc. Using the FoxPro file formats supported by the DBF Extender, you will obtain the best balance of performance and capacity.

To give the DBF Extender maximum performance and reliability, PlugSys has teamed up with Sequiter Software to bring CodeBase technology into the DBF Extender. (See **About CodeBase** on page 12.)

About CodeBase

CodeBase is a high-speed Xbase-compatible database engine that's small, easy-to-use and portable. It is available with programming interfaces for various languages. CodeBase provides superb access to FoxPro/dBASE and Clipper database files.

CodeBase was developed by Sequiter Software. The technology has more than a decade of solid service worldwide. CodeBase has won the praise of programmers, technical journals and information technology managers.

PlugSys teamed up with Sequiter Software to bring CodeBase technology to Max.

NOTE: More information can be found on CodeBase products can be found at the Sequiter Software web site. (**http://www.sequiter.com**)

Traditionally, development with CodeBase required a great deal of low-level calls to specialized library functions. In **How CodeBase Works** (on page 14), we contrast the low-level programming approach required to use the traditional CodeBase API with the straightforward Xbase approach you can adopt with Max together with the DBF Extender.

NOTE: The DBF Extender is distributed and supported by PlugSys International. (**http://www.plugsys.com**)

Supported File Formats

CodeBase can read and write to three distinct types of Xbase data file formats:

- FoxPro: CDX indexes, FoxPro memo files
- dBASE: MDX indexes, dBASE memo files
- Clipper: NTX indexes, DBT memo files

Applications using the DBF Extender can share data with other applications in FoxPro, dBASE and Clipper. The locking mechanism is consistent with that of the database type selected. (e.g. when your application selects FoxPro file access, you can share data files with FoxPro applications.)

Because database handling differs among the supported file formats, you specify the appropriate **DBTYPE** targeted for each of the supported formats.

How CodeBase Works

Scenario 1: The CodeBase API: *(Programming in C)*

To use the CodeBase functionality with custom software development, developers must learn the CodeBase API. This is used to interact with CodeBase and perform operations like opening databases, indexes, reading records, obtaining field values, etc. ***None of this is necessary in Max, thanks to PlugSys Open XBase Connectivy (OXBC).***

This example in C shows how to open an Xbase DBF table using the CodeBase API:

```
#include "d4all.h"

CODE4 codeBase ;

DATA4 *OpenDatafile( void )
{
   DATA4 *Datafile = 0 ;

   Datafile = d4open( &codeBase, "TEST.DBF" ) ;

   /* datafile is opened; datafile assigned a value */
   return (Datafile) ;
}

void main( void )
{
     DATA4 *data ;
     code4init( &codeBase ) ;
     data = OpenDatafile( ) ;

     /* data now has a valid address */

     d4top( data ) ;

     code4close( &codeBase ) ;
}
```

NOTE: The CodeBase API is licensed by Sequiter Software. Please visit the Sequiter web site at **http://www.sequiter.com** for information on licensing and availability.

Scenario 2: Using CodeBase OXBC *(Simple, Transparent)*

PlugSys Open Xbase Connectivity (OXBC) frees Max developers from learning the CodeBase API. Max transparently handles CodeBase interaction in response to classic Xbase commands like **USE** and **CREATE**.

- You specify the OXBC driver to be used when opening a DBF table. (Use **DBTYPE** clauses to classic commands or **SET DBTYPE**.)
- Max maps and redirects all database-handling commands and functions to the specified driver.

This example illustrates how to use the DBF Extender for FoxPro in Max:

```
USE customers ALIAS cust DBTYPE foxpro
GO TOP
DO WHILE ! EOF()
   ? cust->name
   SKIP
ENDDO
CLOSE
```

The only change from your past Xbase experience is to specify the **DBTYPE** as **FOXPRO**, **DBASE** or **CLIPPER**.

NOTE: Refer to Max OXBC documentation and the Max Language Reference for more information about OXBC-related commands and functions available in Max.

The DBF Extender Files

The DBF Extender kit includes separate files for each Xbase implementation. The index file extension embedded within the driver filename indicates the Xbase implementation:

- CBOXBC-CDX.DLL (win32) / cboxbc-cdx.so (Linux) – FoxPro files with CDX index
- CBOXBC-MDX.DLL (win32) / cboxbc-mdx.so (Linux) – dBASE files with MDX index
- CBOXBC-NTX.DLL (win32) / cboxbc-ntx.so (Linux) – Clipper files with NTX index

The **DBTYPE** clause available in Max language is used to specify which file format should be used. Tipically, the developer uses the command **DEFINE DBTYPE** to define what DLL file handles a specific DBTYPE.

Max already has three predefined DBTYPEs that map to the files above. So developers don't have to perform the **DEFINE DBTYPE** to use DBF Extenders.

Implementation Constraints

Indexes

Complexity of Expressions

Index key expressions are stored within index files. Therefore, this is a runtime operation. So any expressions must be evaluated by the CodeBase driver *and not by Max*. CodeBase does not include a full Xbase expression evaluator.

CodeBase has its own expression evaluator to support the common Xbase operators and a selected list of Xbase functions.

NOTE: Before proceeding with a CodeBase OXBC application, be sure to certify your index key expressions. This manual provides a full list of operators and functions which CodeBase will recognize and process. (See **Creating Index Key Expressions That CodeBase Can Evaluate** on page 28.)

NOTE: Although the Clipper .NTX is supported, we strongly recommend adopting a more modern data format if this is possible within your application design. The multiple index formats are more robust and lead to self-maintained indexes.

User-Defined Functions In Index Key Expressions

Because index expressions must be evaluated by the CodeBase engine, *user-defined functions (UDF's) cannot be processed by the driver*.

If you are accustomed to writing user defined functions as any part of a key expression, you will have to make some design decisions: Can you create identical or similar functionality in another way?

- Can you move filtering or mapping into the program code as the user navigates or in a processing loop?
- Will **`SET FILTER TO`** perform acceptably at runtime?
- Will a more conventional Xbase expression perform the same data transformation as your UDF?

Single Index File Implementations: Supported and Unsupported

The following format is supported by the DBF Extender:

- Clipper .NTX

NOTE: The DBF Extender creates Clipper .NTX filenames in upper case. This behavior cannot be changed by you. Be certain when using a case-sensitive operating system that you open the file using an upper case filename.

The following formats are not supported:

- dBASE .NDX
- FoxPro .IDX

Other Implementation Issues

The following subsections discuss specific issues about index files. For a fuller discussion, see **About Indexes** on page 21.

Multiple Index File Handling

These Xbase implementations are supported for multiple index files:

- FoxPro CDX
- dBASE MDX)

Applications using the **TAG** clause (in **USE** and **SET ORDER**) are handled by the DBF Extender. The **ORDER** clause will also work appropriately when referring to an index "tag".

NOTE: Because a multiple index file can contain a large number of indexes, it is unlikely that any given table would require more index expressions than can be contained in a production index.

About Production Index Files

A production index file is a multiple index file with the same structure and rules as any other multiple index file. But there is one important distinction. You create it with the same root filename as the DBF file with which the index is associated. (Example: In the case of **`customers.dbf`** and **`customers.cdx`**, the latter file is a production index.)

The DBF Extender automatically opens and updates all indexes within the production index file associated with the DBF:

- Your application issues a **USE** command on a **DBTYPE** that supports multiple index files.
- CodeBase looks in the same directory for a multiple index file with the same root filename. This would be the *production index file*.
- If such a file is located, CodeBase opens that production index file.
- All subsequent updates to the DBF file will result in automatic updates to relevant indexes within this production multiple index file.

NOTE: Max supports ***one multiple index file per DBF table***. This is a production index file. If you make this your standard way of storing your indexes, you eliminate code and maintenance problems. Production index files mean you no longer need to explicitly "open" indexes when users update the DBF.

NOTE: ***Create a well-behaved heterogeneous environment using the DBF Extender.*** If your Max application is sharing data with other Xbase tools, you have a very compelling reason to rely on production indexes. Each application may need only a subset of the indexes. And it does not have to "know" about other indexes used by other applications. Having a production index ensures that each application can update all relevant indexes without knowing about indexes used by other applications. (And this works equally well whether you are working only with Max or a combination of Max applications and other Xbase tools.)

For more information, **Production Indexes & Conventional Indexes** on page 24.

How To Generate A FoxPro Production .CDX For The First Time

- Open the DBF file you want to index in ***exclusive*** mode.

If A Production CDX Is Deleted Or Omitted From A Copy

Once you create a CDX file, the DBF header contains a link to the related production CDX file. If you manually remove (delete) the CDX file, the DBF Extender will fail to open the DBF file. If this happens to you, you have to use a tool that can alter the DBF header. This is necessary to remove the reference to the now deleted CDX file.

HEADER()

`header()` returns 0 when used with OXBC drivers that do not support it. The DBF Extender does not support `header()`.

About Indexes

A Quick Introduction (For Xbase Beginners)

The purpose of an index is to order data records in purposeful ways. One approach is to physically sort records in your data files. But this is inefficient. It requirescontinually shuffling records as users interact with the data by updating or inserting. And sorting only permits you to maintain a single sequence of records.

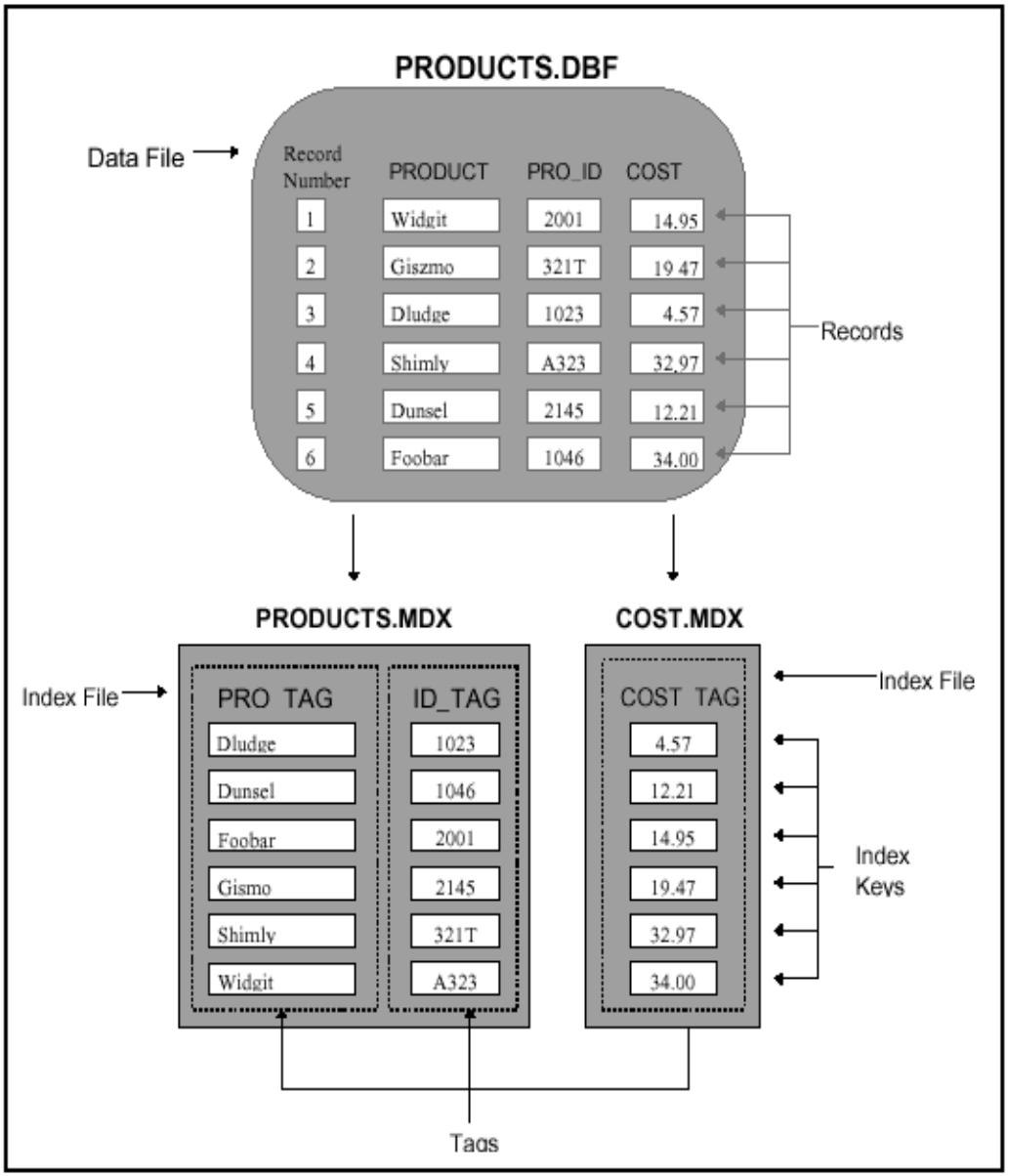
Indexes permit you to leave the data records in their original order within the data table. The sorted orderings are instead stored in a separate file called an index file. When you create an index file, you are effectively sorting the data file. Index files are efficiently maintained and you can have an unlimited number of sorted orderings continually available.

Comparing The Formats

Each index file can contain one or more sorted orderings. These sorted orderings are identified by a *ta*g. That is, each index file tag corresponds to a single sorted ordering. CodeBase supports three types of index files. Their attributes and differences are described below:

File Format	Compatibility	Number of Tags (Index Keys)	Production Indexes	Index Filtering	Descending Order?
.MDX	**dBASE IV**	**1-47**	**Yes**	**Yes**	**Yes**
.CDX	**FoxPro 2.x FoxPro 3.x**	**1-47**	**Yes**	**Yes**	**Yes**
.NTX	**Clipper**	**1**			with descend() function

The difference between indexes and tags are illustrated below:



Index Expressions

An index expression is an Xbase expression that is used to determine the index key for each record. An index expression must evaluate to a value of one of the following types: Numeric, Character, or Date types. When you are using .CDX (FoxPro) indexes, you can also index on Logical expressions. (See **Creating Index Key Expressions That CodeBase Can Evaluate** on page 28.)

The most commonly used index expression is a field name. For example, the index expression for the PRO_TAG tag in Figure 5.2 is "PRODUCT". When this expression is evaluated for record number 6, the value of the field PRODUCT is returned; for record number 6, this value is "Foobar ". To put it simply, tag PRO_TAG is ordered by the contents in field PRODUCT.

Other common index expressions involve generating tags based on two or more fields. This is known as a compound index key. For example, if we assume that field PRO_ID is also a character field, we can base a tag ordering with the following index expression: **`PRODUCT + PRO_ID`**. This produces an index key consisting of the PRODUCT field concatenated with the PRO_ID field. For record number 6, the resulting index key is "Foobar 1046".

A large group of Xbase functions are also permitted.(See **Creating Index Key Expressions That CodeBase Can Evaluate** on page 28.) If we wanted to base a tag on the PRODUCT field, but remain case insensitive, we can use UPPER() to convert the index key to upper case. In this case the index expression would be **`UPPER(PRODUCT)`**.

Multiple Indexes Are Better

In Max, Clipper, dBASE II and dBASE III Plus index files must be manually or programmatically opened each and every time the data file is updated or put into control. This leaves the possibility that you might forget to open an index file when needed. The result is that the index loses synchronization with the .DBF.

dBASE IV and FoxPro use multiple index files (with more than one index expression (tag) in a file). Additionally these implementations suport production index files. (Where the database engine automatically opens the multiple index file when you open the data file.)

Production Indexes & Conventional Indexes

Creating an index file is similar to creating a data file; in fact you can even create both at the same time. The DBF Extender for Max supports these types of index files:

- **Multiple production index file**- a multiple index file that is opened automatically when its associated data file is opened. ***A data file can have only one production index.*** This is a multiple index file it must have the same root name as the DBF file:

```
USE customer      && this automatically opens:
                  && if dBASE- customer.mdx
                  &&  if FoxPro - customer.cdx
```

- **Single index file**- A data file can have an unlimited number of these non-production indexes. To use these indexes, you must explicitly open them along with the DBF file (with the **USE** command) or later using the **SET INDEX TO** command.

Production Indexes Are Automatically Updated

CodeBase automatically updates all open indexes for the data file as long as you rely on production index files. When a record is added, modified, or deleted, the appropriate tag entries for all of the open index tags are modified automatically. This is true whether CodeBase has automatically opened a production index, whether your program has explicitly opened a non-production multiple index file or you are using a conventional single index file.

Once the index file is open, CodeBase handles all of the key manipulation in the background, letting you concentrate on application programming.

Creating Indexes

```
USE mytable DBTYPE
INDEX on name TAG name
INDEX on country TAG country
INDEX on recnum TAG myrecnum
 ...
SET ORDER TO name
 ...
```

Modifying Production File

If you want to remove or change individual index tags once you have created a multiple index file:

- Close down any applications that could potentially use the DBF and multiple index file.
- Physically delete the multiple index files.
- Then use the **CBAutoOpen()** function to turn off automatic opening of the production index.
- Then write code to generate the new tags for the newly regenerated multiple index file.

Selecting A Tag In A Multiple Index File

Initially a data file does not have a selected tag and is therefore in natural order. Natural order is the order in which the records were added to the data file. When you want to use a particular sort ordering, select a tag using SET ORDER TO:

```
SET ORDER TO [TAG] product_code
```

NOTE: **TAG** is an optional clause.

NOTE: The **TAG** clause is not valid for use with **SET INDEX TO**.

Record Navigation After Selecting a Tag

Once a tag is selected, the behavior of navigation commands change. For each of these commands, the behavior is as if the table were in the physical order of the index now in effect:

- **GO TOP**- first record in indexed order
- **GO BOTTOM**- last record in indexed order
- **SKIP**- next record in indexed order

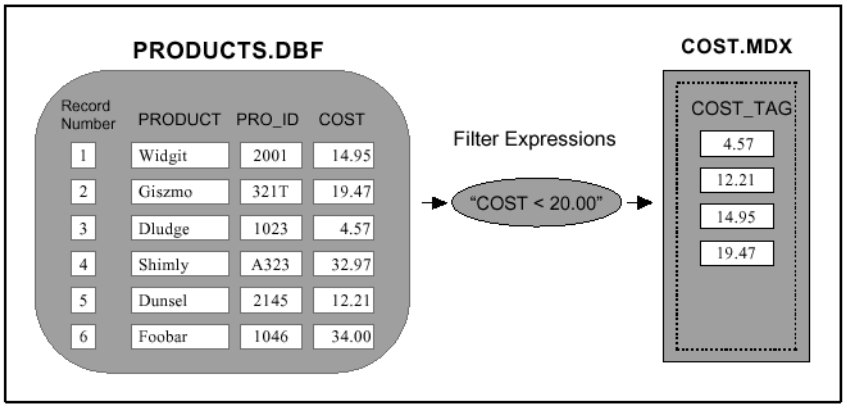
Indexes With Filtering Characteristics

You can create filter indexes to constrain records to a subset of the entire table. The subset is created using conditional expressions in the **INDEX** command's **FOR** clause. Only records that evaluate to **.T.** will appear in that tag. A tag filter is created when the tag is created.

NOTE: CodeBase cannot evaluate user defined functions at runtime. Be sure to use expressions and functions that CodeBase can evaluate. (See **Creating Index Key Expressions That CodeBase Can Evaluate** on page 28.)

Filter Expressions

A filter expression is an Xbase expression that returns a Logical result. This is used as a tag filter. The expression is evaluated for each record as its tag entry is updated. If the filter expression evaluates to true, an entry for that record is included in the tag. If it evaluates to false, that record's tag entry is omitted from the tag:.



Group Files: Making Clipper Indexes Emulate Multiple Index Files

In Clipper, **.NTX** index files must be manually opened each and every time the data file is updated or put into control. This leaves the change that you might forget to open an index file and put an index out of synchronization with the .DBF. dBASE IV and FoxPro use compound index files (more than one tag in a file) and production index files (automatically opens when data file opens) to avoid these problems.

CodeBase has introduced group files in order to compensate for this limitation of the **.NTX** file format. A group file allows you to use the same function calls when using **.NTX** index files as you would when using **.CDX** and **.MDX** index files.

This is accomplished by emulating production indexes and multiple tags per index file.

Creating Group Files

Use a text editor to create a group file definition for existing index files.

- Name the file with the same root name (before the dot) as the related .DBF file. Give it an extension of **.CGP**.
- Then enter the names of the index files (without an extension) you want Max to open when the DBF is opened.
- Enter one index filename per line with a carriage return after each index filename.

GROUP FILE: **PRODUCTS.CGP**

```
PRO_TAG
P_NUM_TAG
```

NOTE: Group files are a CodeBase-specific feature. They must be manually created by you. Index files generated with Clipper will not automatically have group files. Either access the .NTX group files directly or create your own group files as described above.

Creating Index Key Expressions That CodeBase Can Evaluate

The CodeBase expression evaluator implements a core set of common Xbase operators and library functions. Please consult this list to confirm whether your index key expressions can be properly interpreted by the DBF Extender.

Operators

Numeric

+	Addition
-	Subtraction
*****	Multiplication
/	Division

Character

+	Concatenation
-	Concatenation (removing intervening spaces)

Relational

=	Equal to
<>	Not equal to
#	Not equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
$	Containing

Logical

.NOT.	Not
.AND.	And
.OR.	Or

Functions

The following functions are supported by CodeBase. Notes are provided only when an implementation presents differences from Max or other common Xbase tools.

alltrim()	
ascend()	**ASCEND() accepts all types of parameters, except complex numeric expressions. ASCEND() converts all types into a Character type in ascending order. In the case of numeric types, the conversion is done so that the sorting will work correctly even if negative values are present.** ***NOTE:*** This function is not supported by dBASE, FoxPro or Clipper. Either create a UDF in those products or reserve the use of this function for Max-only applications.
chr()	

ctod() **date()**	
day()	
descend()	**DESCEND() accepts any type of parameter, except complex numeric expressions. DESCEND() converts all types into a character type in descending order.** **For example, the following expression would produce a reverse order sort on the field ORD_DATE followed by normal sub-sort on COMPANY.** **e.g.** `DESCEND(ORD_DATE) + COMPANY` **Also see ascend().** ***NOTE:*** This function is not supported by dBASE or FoxPro. Either create a UDF in those products or reserve the use of this function for Max-only applications. ***NOTE:*** This function is compatible with Clipper, only if the parameter is a Character type.
deleted()	

dtoc()	
dtos()	
iif()	
left() **ltrim()**	
month()	
reccount()	
recno()	

right()	
stod()	
str()	
substr() **time()**	
trim()	
upper()	
val() **year()**	

Max Extension Functions

OXBC drivers may carry Max Extension Functions to provide specific services not completely covered by the driver's implementation. PlugSys provides the functions described below to add functionality to the CodeBase driver.

NOTE: If you intend to open datafiles of more than one **DBTYPE** in the same application, please be aware of the following:

- There are separate drivers for each DBTYPE.
- The DBF Extender does not close down the previous driver for a given DBTYPE when another DBTYPE has been opened.
- This means that the Max Extension Functions from the previously opened DBTYPE remain in memory when additional ones are opened.
- When functions have the same name the one loaded first is the one that gets excuted if called.
-Under most circumstances, this does not pose a problem. But if you call **`CBErrorCode()`**, you must be aware that you are calling the implementation from the first DBTYPE driver’s extension functions.

CBAutoOpen()

```
CBAutoOpen( newState )
```

CBAutoOpen( ) is used to control how the DBF Extender opens production indexes (CDX, MDX). By default, production indexes are automatically opened when you open the associated DBF file (equivalent to **`CBAutoOpen = .T.`**). The status of CBAutoOpen( ) can be changed at any time.

Arguments:

newState: a logical value (.T. or .F.) indicating the new state.

Return Value

The function returns the current state. If you call the function to set the state, this function returns the ***previous state***.

Remarks

This function resides in the DBF Extender. To be available to the application, the driver must be loaded. The command SET DBTYPE TO FOXPRO/DBASE/CLIPPER should be used to automatically load the DBF Extender and make the CBxxxx( ) functions available at runtime.

NOTE: If a production index (CDX/MDX) was erased, corrupted or is not available for any other reason, the **USE** command will typically fail. Disabling CBAutoOpen() bypasses this problem by allowing an application to open only the DBF file.

NOTE: Turn off CBAutoOpen() whenenver you need to recreate a production index file.

Default: The default value for CBAutoOpen is TRUE.

Example:

```
// Load DBF Extender to make CBxxx() functions available
SET DBTYPE TO FOXPRO
erase "customers.cdx"    // Delete the production file that will
be reacreated
CBAutoOpen(.F.)    // Disabling CBAutoOpen, or the USE command
will fail as
                  // the production index customers.CDX has been
erased
use customers      // Opening only the DBF file
index on NAME tag NAME    // Now creating the production index
customers.cdx
```

CBIndexDescend()

```
CBINDEXDESCEND( [descend_on_off] )
```

This function controls how index keys are sorted (ascending or descending) when indexes are created by the DBF Extender. Default setting is **OFF** (indexes are created with keys in ascending order).

Return value

The function returns the current status (.F. = keys are sorted in ascending order / .T. = keys are sorted in descending order). If you call the function with a setting parameter, the function changes the status and returns the previous setting.

Remarks

This setting only takes effect when indexes (or index tags) are created. To change the sort order, you must call this function and regenerate the index. The sort order flag is stored in the index file whenever the entire index is generated with **`INDEX ON`**.

Example

```
set DBTYPE to FOXPRO
use customers
// Creating an index tag with keys in descending order
SaveStatus := CBIndexDescend( .T. )
index on DT_REGISTR tag dt_registr
// Restoring the default setting
NewStatus := CBIndexDescend( SaveStatus )
```

CBLargeOn()

- Enables large file support.

Once enabled, large file support remains enabled for the duration of the application's operation. (It can not be turned off without restarting the application.) This function must be used before any CodeBase-managed data files are opened.

NOTE: Large file support is ***not a standard mode of operation***. Keep this in mind if you need to share data with other Xbase products. CodeBase will use an ***incompatible locking algorithm***. This means that all applications that are accessing the same files are using the same locking algorithm. If multiple applications share the same files but use different locking algorithms, the database files can become corrupt. Use sensible precautions if you accept the tradeoff of large file support.

CBErrorCode()

Use this function for debugging or monitoring purposes.

Returns the current error code. (See **Error Codes** on page 41.)

- `0` = no error condition.
- Values less than zero represent errors.
- Occasionally, a function may set this member to a positive value, indicating a non-error condition.

CBErrorText(nErrorCode)

Use this function for debugging or monitoring purposes.

- Returns the message corresponding to the error code. (See **Error Codes** on page 41.)

CodeBase Data Size Specifications

NOTE: *Some CodeBase limits exceed those of FoxPro, dBASE or Clipper. It is possible to be within CodeBase limits but lose compatibility with other products. (See **CBLargeOn()** on page 33.)

DBF size	1 Gigabyte (*8,589,934,590 Gigabytes with large file support)
Index file size	4 GB*
Memo file size	131,072 Gbyte
Memo entry size	4,294,967,296 - 100 = 4,294,967,196 bytes (approx. 4 GB)
Character field width	65517 bytes* ***NOTE:*** Be sure you understand the maximum field width if database compatiblity matters to you. Use 254 byte maximum width for FoxPro or dBASE compatibility or 32767 byte maximum for Clipper compatibility.(See **CBLargeOn()** on page 33.)

Float field	19
Numeric field	• 20 FoxPro • 20 dBASE IV • 19 Clipper
Maximum Number of fields	2046* ***NOTE:*** Be sure you understand the maximum number of fields if database compatiblity matters to you. Use 128 fields maximum for dBASE compatibility or 1024 fields maximum for Clipper compatibility. (See **CBLargeOn()** on page 33.)
Record size	2 Gbytes* (if you use large file size features) ***NOTE:*** Be sure you understand the maximum record size if database compatiblity matters to you. Use a maximum of 128 bytes for dBASE compatibility or 1022 bytes for Clipper compatibility. (See **CBLargeOn()** on page 33.)
Tags per index	• Unlimited FoxPro • 47 dBASE IV • 1 Clipper (actually a classic NTX file)
Index key expression size	• 240 bytes: FoxPro • 102 bytes: dBASE IV • 338 bytes: Clipper

Field Types.

Field Type	Structure Code	Comments
Character	C	
Numeric	N	Numeric fields store numerical information. It is stored internally in the data file as a string of digits. ***NOTE:*** This field is useful for compatibility with dBASE and FoxPro, which treat Floating point and Numeric fields differently. ***NOTE:*** Use this field to store values that will NOT be used in floating point calculations.
Logical	L	
Date	D	
Memo	M	
Binary	B	Binary fields are handled in the same way as Memo fields. It stores binary information. The field length is fixed at 10 bytes. ***NOTE:*** Only available with dBASE IV file compatibility.ient
Character (binary)	Z	This is the same as a Character field, but can store binary information more efficiently. ***NOTE:*** Only available with Visual FoxPro file compatibility.
Currency	T	The monetary data is stored in binary form. The field length is fixed at 8 bytes. ***NOTE:*** Only available with Visual FoxPro file compatibility.

Binary	B	Binary fields are handled in the same way as Memo fields. It stores binary information. The field length is fixed at 10 bytes. ***NOTE:*** Only available with dBASE IV file compatibility.ient
Character (binary)	Z	This is the same as a Character field, but can store binary information more efficiently. ***NOTE:*** Only available with Visual FoxPro file compatibility.
Currency	T	The monetary data is stored in binary form. The field length is fixed at 8 bytes. ***NOTE:*** Only available with Visual FoxPro file compatibility.
DateTime	T	The date and time is stored as binary data. The field length is fixed at 8 bytes. ***NOTE:*** Only available with Visual FoxPro file compatibility.
Double	D	64-bit double values are stored in binary form. The field length is fixed at 8 bytes. ***NOTE:*** Only available with Visual FoxPro file compatibility.

.

Float	F	CodeBase treats this field as a Numeric field. ***NOTE:*** This field is useful for compatibility with dBASE and FoxPro, which treat Floating point and Numeric fields differently. ***NOTE:*** Use this field to store values that will be used in floating point calculations.
General	G	General fields are handled in the same way as Memo fields. It stores OLE objects. The field length in the data file is fixed at 4 bytes in Visual FoxPro tables, otherwise it is fixed at 10 bytes. ***NOTE:*** Only available with FoxPro file compatibility.
Integer	I	Integer fields store 32-bit integers in binary form. The field length is fixed at 4 bytes. ***NOTE:*** Only available with Visual FoxPro file compatibility. i

Memo (binary)	X	This is the same as a Memo field, but can store binary information more efficiently. The field length is fixed at 4 bytes. ***NOTE:*** Only available with Visual FoxPro file compatibility.
Unicode	W	Unicode fields store Unicode strings. The field is padded with nulls. When creating a table with this field type, remember to specify the length as twice the required number of Unicode characters, because each Unicode character is stored in 2 bytes. ***NOTE:*** This field type is only compatible with CodeBase. It will not be recognized by other Xbase tools. When creating a Unicode field in a FoxPro table, CODE4 compatibility must be set to 30.

Error Codes

Disk Errors

Code	Description
-10	Closing File- An error occurred while attempting to close a file.
-20	Creating File- This error could be caused by specifying an illegal file name, attempting to create a file which is open, having a full directory, or by having a disk problem. This error also results when the operating system doesn't have enough file handles. ***NOTE:*** See **-60** for more information.
-30	Determining File Length- An error occurred while attempting to determine the length of a file. This error occurs when CodeBase runs out of valid file handles. ***NOTE:*** See **-63** for more information.
-40	Setting File Length- An error occurred while setting the length of a file. This error occurs when an application does not have write access to the file or is out of disk space.
-50	Locking File- An error occurred while trying to lock a file. Generally this error occurs when lock enforcement is on and an attempt is made to modify an unlocked record.
-60	Opening File- A general file failure occurred opening a file. This error may also include any of the -6x errors if the selected compiler or operating system does not allow for distinguishing between various file errors.
-61	Permission Error Opening File- Permission to open the file as specified was denied. For example, another user may have the file opened exclusively.
-64	File Find Error Opening File- File was not found as specified.

-70	Reading File- An error occurred while reading a file. This could be caused by calling d4go with a nonexistent record number.
-80	Removing File- An error occurred while attempting to remove a file. This error will occur when the file is opened by another user or the current process, and an attempt is made to remove that file.
-90	Renaming File- An error occurred while renaming a file. This error can be caused when the file name already exists.
-110	Unlocking File- An error occurred while unlocking part of a file.
-120	Writing to File- This error can occur when the disk is full.

DBF Errors

Code	Description
-200	Not a data file- File is not in expected datafile format. This error occurs when attempting to open a file as a .DBF data file when the file is not actually a true data file. If the file is a data file, its header and possibly its data is corrupted. It will also occur when creating a data file with an invalid field type.
-210	Unrecognized Field Name- An internal function was called with a field name not present in the data file.
-220	Unrecognized Field Type- A data field had an unrecognized field type. The field type of each field is specified in the data file header.
-230	Record Length too Large- The total record length is too large.
-240	Record Append Attempt Past End of File
-250	Seeking- This error can occur if int d4seekDouble tries to do a seek on a non-numeric tag.

Index Errorss

Code	Description
-300	Tag Entry Missing- A tag entry was not located. This error occurs when a key, corresponding to a data file record, should be in a tag but is not.
-310	Not a Correct Index File- This error indicates that a file specified as an index file is not a true index file. Some internal index file inconsistency was detected.
-330	Tag Name not Found- The tag name specified is not an actual tag name. Make sure the name is correct and that the corresponding index file is open.
-340	Unique Key Error- An attempt was made to add a record or create an index file which would have resulted in a duplicate tag key for a unique key tag.
-350	Tag information is invalid- Usually occurs when creating an index tag with invalid information in the input TAG4INFO structure.

Expression Evaluation Errors

Code	Description
-400	Comma or Bracket Expected- A comma or a right bracket was expected but there was none. For example, the expression "SUBSTR(A" would cause this error because a comma would be expected after the 'A'.
-410	Expression not Complete- The expression was not complete. For example, the expression "FIELD_A +" would not be complete because there should be something else after the '+ '.
-420	Data File Name not Located- A data file name was specified but the data file was not currently open. For example, if the expression was "DATA->FIELD_NAME", but no currently opened data file has "DATA" as its alias.
-422	IIF() Needs Parameters of Same Length- The second and third parameters of dBASE function IIF() must resolve to exactly the same length. For example, IIF(.T., "12", "123") would return this error because character expression "12" is of length two and "123" is of length three.
-425	SUBSTR() and STR() need Constant Parameters- The second and third parameters of functions SUBSTR() and STR() require constant parameters. For example, **`SUBSTR( "123", 1, 2 )`** is fine; however, **`SUBSTR( "123", 1, FLD_NAME )`** is not because FLD_NAME is not a constant.
-430	Number of Parameters is Wrong- The number of parameters specified in a dBASE expression is wrong.
-440	Overflow while Evaluating Expression- The dBASE expression was too long or complex for CodeBase to handle. Such an expression would be extremely long and complex. The parsing algorithm limits the number of comparisons made in a query. Thus, very long expressions can not be parsed.
-450	Right Bracket Missing- The dBASE expression is missing a right bracket. Make sure the expression contains the same number of right as left brackets.
-460	Sub-expression Type is Wrong- The type of a sub-expression did not match the type of an expression operator. For example, in the expression "33 .AND. .F.", the "33" is of type numeric and the operator ".AND." needs logical operands.

-470	Unrecognized Function- A specified function was not recognized. For example, the expression **SIMPLE(3)** is not valid.
-480	Unrecognized Operator A specified operator was not recognized. For example, in the dBASE expression "3 } 7", the character '}' is in a place where a dBASE operator would be expected.
-490	Unrecognized Value- A character sequence was not recognized as a dBASE constant, field name, or function.
-500	Unterminated String- According to dBASE expression syntax, a string constant starts with a quote character and ends with the same quote character. However, there was no ending quote character to match a starting quote character.
-510	Expression Invalid for Tag- The expression is invalid for use within a tag. For example, although expressions may refer to data aliases, tag expressions may not.

Optimization Errors

Code	Description
-610	Optimization Error- A general CodeBase optimization error was discovered.
-620	Optimization Removal Error- An error occurred while suspending optimization.
-630	Optimization File Flushing Failure- An error occurred during the flushing of optimized file information.

SET RELATION Errors

Code	Description
-710	Relation Error- A general CodeBase relation error was discovered.
-720	Matching Slave Record Not Located- CodeBase could not locate the master record's corresponding slave record.
-730	Relation Referred to Does Not Exist or is Not Initialized- Referenced a non-existent or improperly initialized relation. Possible cases are: non-initialized memory or an invalid pointer has been passed to a relate module function, or function calls have occurred in an invalid sequence (for example, relate4skip may not be called unless relate4top has previously been called).

Severe Errors

Code	Description
-910	Unexpected Information- CodeBase discovered an unexpected value in one of its internal variables.
-920	Out of Memory- CodeBase tried to allocate some memory from the heap, in order to complete a function call, but no memory was available. This usually occurs during a database update process, which happens when a record is appended, written or flushed to disk. During the update, if a new tag block is required, CodeBase will attempt to allocate more memory. If the memory is not available, CodeBase will return the "Out of Memory" error. If this error occurs during the updating process, the index file will most likely become corrupt.
-930	Unexpected Parameter- A CodeBase function was passed an unexpected parameter value. This can happen when the application programmer forgets to initialize some pointers and thus null pointers are passed to a function.
-935	Null Input Parameter unexpected Unexpected parameter - null input.
-970	Data Structure Corrupt or not Initialized CodeBase internal structures have been detected as invalid.

Unsupported Feature Errors

Code	Description
-1090	Function unsupported- Operation generally not supported in this configuration.
-1095	Application/Library version mismatch- Version mismatch (e.g. client version mismatches server version).

Memo Errors

Code	Description
-1110	Memo File Corrupt- A memo file or entry is corrupt.
-1120	Error Creating Memo File- For example, the CODE4.memSizeMemo is set to an invalid value.

Communication Errors

Code	Description
-1300	Communication Information Corrupt- Connection information corrupt. In general would indicate a network hardware/software failure of some sort. For example, out of date device drivers may be being used on either a client or a server machine.
-1310	Connection Failure- A connection failure. For example, a connection failed to be established or got terminated abruptly by the network.
-1320	Socket Failure- A socket failure. All CodeBase software use sockets as their basis for communications. This error indicates a failure in the socket layer of the communications. For example, the selected communication protocol may be unsupported on the given machine. Alternatively, an unsupported version of the networking software may be being used (e.g. Windows Sockets 1.0 or Novell 2.x).
-1330	Network Failure- A network error occurred. Some CodeBase communications protocols are dependent on network stability. For example, if the local file-server is shut-down, CodeBase may be unable to continue operations, and may therefore fail with an error. Alternatively, a physical network error may be detected (for example, if a network cable is physically cut or unplugged, thus removing the physical connection of the computer from the network.)
-1340	Failure Loading Communication DLL- An attempt to load the specified communication DLL has failed. Ensure that the requested DLL is accessible to the application. This error may also occur if attempting to start a client or server under Windows if Windows is unstable.
-1350	Network Timed Out- This error occurs whenever CodeBase has timed out after timeout seconds have elapsed.
-1360	Communication Message Corrupt- A communication message error has been detected. For example, a client may have not been able to properly send a complete message to the server.
-1370	Communication Packet Length Mismatch- A packet length error has been detected. Possibly the CodeBase client software mismatches the server implementation.
-1380	Communication Packet Corrupt- A packet corruption has been detected. Check e4corrupt for potential causes of this failure.
-1390	System-level Communications Error- This error would occur if the network hardware failed or if the server disconnected from the client.

Miscellaneous Errors

Code	Description
-1400	CodeBase Capabilities Exceeded (system maxed out)- The physical capabilities of CodeBase have been maxed out. For example, the maximum allowable connections for a computer may have been exceeded by the server. Often these errors can be solved by modifying system or network configuration files which have placed arbitrary limits on the system. This error will also be generated when the maximum number of users for the server is exceeded.
-1410	CodeBase in an Unacknowledged Error State- CodeBase failed due to being in an error state already. Generally comes out as an error return code if a high-level function is called after having disregarded a CodeBase error condition.
-1420	Name not Found error- The specified name was invalid or not found. For example, d4index was called with a non-existent index alias.
-1430	Authorization Error (access denied)- The requested operation could not be performed because the requester has insufficient authority to perform the operation. For example, a user without creation privileges has made a call to create.

Server Failure Errors

Code	Description
-2100	Server Failure- A client-server failure has occurred. In this case, the client connection was probably also lost.
-2110	Server Configuration Failure- An error has been detected in the server configuration file. The configuration file is only accessed when the server is first started, so once the server is operational, this error cannot occur.
-2120	Catalog Failure- A catalog failure has occurred. For example, the catalog file may exist but may be corrupt.

The PlugSys ODBC Connector

Getting Started

The purpose of this chapter is to give you direct implementation information and the support commands and functions necessary to make ODBC connections within PlugSys development products.

ODBC is primarily intended to provide access to SQL-oriented database systems. If you are new to SQL and relational DBMS engines (RDBMS's), we strongly encourage that you learn more about that world before attempting your first development effort. We provide some suggested web links in **More Resources** on page 74.

What is ODBC?

Open Database Connectivity (OXBC) is an access method devised by Microsoft and built upon the Call Level Interface work of the SQL Access Group. Its goal is to create a universal way of accessing data sources. This makes application development more convenient and, when implemented in a generic way, simplifies the process of migration from one database management system to another.

NOTE: More background information on ODBC can be found within this manual. See “More Resources” on page 74.

Universal Access: Implementation Guidelines

The success of any migration using ODBC depends heavily on how generic you have made your implementation:

- Don’t rely on exotic data types when you create a new SQL database- make sure whenever possible to rely on standard ODBC data types. (See **ODBC Syntax For Database Interoperability** on page 70.) If possible, alter your database schema accordingly.
- ODBC can accept queries with proprietary SQL commands and functions if you use pass-through strings with **SQLExec()**. But if you do this, you will have to rewrite your SQL strings when you migrate to another DBMS.

About The ODBC Connector

The Max family of Xbase tools was conceived to simplify the development process. PlugSys set a goal of providing access to popular Xbase data files without adding complexity. The CodeBase OXBC driver realizes this goal by employing classic Xbase commands to open and maintain Xbase files beyond those handled by the native Max engine.

The ODBC Connector is intended as an intelligent gateway between the PlugSys Xbase engine and your database management system. You are responsible for obtaining and licensing the correct ODBC driver(s) for your RDBMS. You license the PlugSys ODBC Connector to complete the picture.

Supported Databases

The OBDC Connector was designed to interact with well-behaved ODBC drivers. PlugSys urges you to obtain the most current version of the correct ODBC driver for your database.

How The ODBC Connector Works

Before you can work successfully with an RDBMS, it is important for you to understand how the process works.

ODBC is an interface designed to provide database independence. ODBC uses the term ***Data Source*** to refer to any database engine that can communicate with the ODBC Manager, provide data and process requests.

The module that acts as a bridge between the ODBC Manager and the actual database engine is called an ***ODBC driver***.

- Before you can access any RDBMS (like MySQL or PostgreSQL), you must install an ODBC driver for that data source.
- Once the ODBC driver module is copied to the machine, it is necessary to define a DSN (Data Source Name) that your application will use to connect to that particular data source.
- Once you have the appropriate ODBC driver installed and a DSN defined, it is easy to connect and interact with that particular data source:

Accessing The Data: A Step-By-Step Explanation

1. **SELECT** a work area as you would do in classic Xbase. . (This is an optional step. As in classic Xbase you may omit it and the current work area is used.)
2. Invoke the PlugSys ODBC connector with **SET DBTYPE TO ODBC**. (This step is necessary before accessing any tables or invoking **USE**. This step loads the ODBC Connector and make its functions available. Making an ODBC Connection requires you to make calls to its functions ***before*** you can actually "USE" the table.)
3. Establish a connection with the ODBC data source with **SQLConnect()**. (See page 86.) (This ODBC connection will be available in the current work area.)
4. Test for the success of the ODBC connection by testing the return value from **SQLConnect()**. (If there is a connection error, use **SQLErrorMessage()** to retrieve information about the nature of the problem. The functions **SQLState()** and **SQLError()** can also be used to retrieve information about the error.)
5. **USE** any table or view available to the connected ODBC data source. This step loads the returned data set into a "virtual DBF" (VBDF). Or you may specify any valid SQL query that will produce a result set. This fills a vDBF with **USE...QUERY**.

   (See **How To Create A vDBF** (on page 57) and **The USE Command Syntax For ODBC** (on page 58).)
6. You must test for success of the USE command. The USE command never generates an error message upon failure. USED() determines if the vDBF has been successfully opened. EOF() can be used to determine if the vDBF is empty (empty recordset). There are two scenarios:

- **USED() = .F.**:
  This indicates that **USE** failed. Use **SQLErrorMessage()** to obtain details about the error. Common causes are invalid table/view names, or if you are using USE...QUERY, syntax errors (invalid column names, etc) on the SQL query statement submitted.
- **USED() = .T.** and **EOF() = .T.**:
  This indicates that the USE command was successful (the SQL query statement was processed successfully) but it returned no rows (an empty recordset).

  It is very common to try to open a vDBF with a narrow-scope query (like "NAME where CustID = nnn") to check if a specific record already exists on a table. After a successful **USE...QUERY** (when **USED()** returns .T.), you can check **EOF()** to determine whether the record exists (was returned by the data source, thus **EOF()=.F.**) or if that record does not exist on the table (in that case no records are returned by the data source, the vDBF is empty, and consequently **EOF() = .T.**)

7. You may navigate and interact with the vDBF data. SQL data types are automatically mapped to standard Xbase types.

NOTE: The current version of the ODBC Connector does not allow you to add or modify data using the Xbase **REPLACE** command. Instead, you should use **SQLExec()** to send INSERT, UPDATE or DELETE queries directly to the database engine. **SQLFormat()** is extremely useful to convert Xbase data types to ODBC data types, when creating SQL INSERT and UPDATE queries.

8. Your application is not automatically notified about data changes to the remote database. These changes could be a result of a transaction initiated by your own application, other applications, or activity at the database server. So you must explicitly fetch fresh data. After performing **SQLExec()**, update your application's own data buffer with **SQLRefresh()**. (See page 106.)
9. Keep the ODBC connection open for any further activity within the application.
10. Any open ODBC connections will be closed when the application or web connection is terminated. (If you explicitly want to close a SQL connection, use **SQLDisconnect()**.)

For further documentation on the language supporting the ODBC Connector (including example code), see **ODBC Connector Language Reference** on page 75.

PlugSys has aimed to make ODBC data access as natural as possible to Xbase developers. But there are some differences you should be aware of. For further information, see **Implementation Issues** on page 62.

Exploring the Virtual DBF (vDBF)

What Is A "Virtual DBF" (vDBF)?

- A Virtual DBF (vDBF) is a memory image of data representing the results of a SQL SELECT query.
- This vDBF behaves as if it were a physical Xbase DBF data table. You can refer to **`alias->fieldname`** the same way you would do with a local physical DBF file.
- You can apply the classic physical navigation commands to a vDBF. (**`SKIP`**, **`GOTO`**, **`GO BOTTOM`**, **`GO TOP`**). (There are some limitations. For further information, see **Implementation Issues** on page 62.)
- When you invoke **`USE`** in an ODBC work area, the RDBMS is queried and the result set populates the work area. We build the vDBF header and field descriptors. (See **The USE Command Syntax For ODBC** on page 58.)
- When you perform an Xbase record positioning instruction (such as **`GOTO`** or **`SKIP`**), the ODBC Connector will try to point to the requested record (or, in SQL terminology, the next row).

If this record is already in memory (i.e. the row has already been fetched and data mapped to Xbase format), that data is made instantly available. Otherwise, the desired record is fetched from the result set. The fetching behavior is set using **SQLFetchOnDemand()**. (See page 97.)

Navigation In A vDBF

- Xbase indexes are not understood by a vDBF. You cannot index or use **`SEEK`**.
- Use a loop with **`SKIP`** and **`IF ... ELSE`** blocks.
- You can alsp use **`LOCATE`** and **`CONTINUE`** if your data set is likely to be small.

How To Create A vDBF

If you know Xbase, you know almost everything you need to create a vDBF. You retrieve the data and populate the vDBF with the classic **USE** command. (See **Accessing The Data: A Step-By-Step Explanation** (on page 54) for a stepwise overview of the full ODBC access model. Also see **The USE Command Syntax For ODBC** on page 58.)

Here are some of the rules:

- If you don't specify a **QUERY** or **ALIAS** clause:
  ```
  USE productlist DBTYPE ODBC
  ```
  the ODBC Connector will access the ODBC connection in the current workarea seeking a table or view with the name **productlist**. If such an entity exists, the ODBC Connector will submit the default query:
  ```
  SELECT * FROM productlist
  ```
  As in classic Xbase, the ALIAS is automatically inherited from the tablename: **productlist**.
- To populate the vDBF with a subset of the source table, view or join, use a query:
  ```
  USE myfriends ;
  QUERY "SELECT lname, fname FROM friends WHERE type = "FRIEND"
  ```

NOTE: Functions defined within OXBC drivers are only available ***after you load the driver***. USE **SET DBTYPE TO** to explicitly load a driver before attempting to populate a virtual DBF. This ensures that all driver-resident functions will be available. Of course, these functions become unavailable when you unload the OXBC driver.

Call **SQLConnect()** before populating a virtual DBF with **USE** <dbf> **DBTYPE ODBC**.

The USE Command Syntax For ODBC

USE

- opens a classic DBF data table, its associated memo file and its associated indexes. (This accesses native Max DBF files or, with the CodeBase OXBC driver, can open FoxPro, dBASE and Clipper files.)
- passes a SQL SELECT statement to an ODBC data source and opens a virtual DBF in the specified work area. ***(Requires PlugSys ODBC Connector.)***

Platforms:	

☑**DOS32/W**
☑**Linux**
	`USE [ data_table ]`
	`[ IN work_area_number / NEW ]`
	`[ ALIAS database_alias ]`
	`[ INDEX idxfile_1 [ KEY Index_Exp1 [ FOR ForCond_1 ] [ UNIQUE ] ]`
	`[, idxfile_2 [ KEY Index_Exp2 [ FOR ForCond_2 ]`
	`[ UNIQUE ] ]`
	`[, ... ] ] ]`
	`[DBTYPE <datasource_type>]`
	`[ORDER index_tag]`
	`[QUERY sql_query]`
	`[ NOUPDATE ]`
	`[ EXCLUSIVE ]`
	`[ SHARED ]`

☑**OXBC: ODBC**	`*USE <tablename|viewname> [QUERY <cSqlQuery>]`
	`[IN <nWorkArea>]`
	`[ALIAS <cAlias>] [DBTYPE <dbtype>]`

NOTE: *This is designed to document how **USE** works when accessing ODBC SQL data sources in conjunction with the PlugSys ODBC Connector. Please see the Max Language Reference for baseline documentation of this command and its Xbase-oriented behavior.

Remarks:

- The **QUERY** clause is only valid when applied to ODBC connections with SQL databases.
- Whenever you know you will require a join for common operations, consider creating a SQL view at the database. This is handy for frequent reuse. Do this also insulates your application code from any changes to the database schema. (As long as the viewname returns the same columnnames and data types, you can make significant changes to the underlying SQL database schema while preserving the code in your application.) And most database engines provide improved performance. Once you have created the join, your application can open the connection and call the viewname: **USE ViewName DBTYPE ODBC**.

Examples:

```
* This exploits default behaviors and values:
* ODBC Connector opens view or table called “productlist”
* and assigns the default alias  “productlist”
* query is omitted: connector submits “SELECT * from productlist”
USE ProductList DBTYPE ODBC // Alias is "PRODUCTLIST"
                            // Use default SELECT * query
                            // "SELECT * from ProductList"

* Submit an explicit query
USE ProductList QUERY [SELECT ProdID, PriceUSD FROM Products] in 2
                            // uses default alias “productlist”
                            // returned set has 2 columns
```

Tips About vDBF Operations

Create Views

- Whenever you know you will require a join for common operations, consider creating a ***SQL view*** at the database. This is handy for frequent reuse because you only have to remember the viewname. And most database engines provide improved performance.

  The strongest argument for creating a view is that this insulates your application code from subsequent changes to the database schema. (As long as the viewname returns the same columnnames and data types, you can make significant changes to the underlying SQL database schema while preserving the code in your application.)

  Once you have created the join, your application can open the connection and call the viewname: **`USE ViewName DBTYPE ODBC`**.
- Number of Records in the result set- This information is not instantly available if you have chosen to turn on **SQLFetchOnDemand()**. Getting the count requires the ODBC Connector to request ***all result rows***. If **`SQLFetchOnDemand()`** is off, the app must request the value of **`LASTREC()`** or **`RECCOUNT()`**, the entire result set will be downloaded and all records will be read.
- If **`SQLFetchOnDemand()`** is off, **`GO BOTTOM`** will force the ODBC Connector to retrieve all remaining rows and populate the vDBF.
- vDBF tables support NULL values. However, the native PlugSys DBF file format does not support NULL values. If you use **`COPY`** to create a physical version of the table, make sure the target database is capable of storing NULLs. (PlugSys offers the CodeBase OXBC driver to support NULL storage in some DBF file formats.)

Field Naming Rules In Virtual DBFs

The virtual DBF is designed to mimic the character of an open physical Xbase datafile. Therefore the vDBF follows the same naming rules. But ODBC data sources may contain field names that violate Xbase naming rules. This section describes how the ODBC Connector handles field names when building a vDBF.

NOTE: This section explains how the ODBC connector handles conflicting naming rules in the data source and Xbase. However, there is a simple precaution giving you complete control over field names in the vDBF. PlugSys strongly recommends that you make use of the AS clause in your SQL SELECT statements. This step will help you whenever:

- you are including an expression in the SELECT list (rather than a field)
- there may be a duplicated column name in the result set
- the data source has a field name with non-alphanumeric characters
- the data source has a long fieldname
- the data source has ambiguous or unfriendly names.

Field Name Length

Virtual DBF field name lengths are more generous than the maximums for standard Xbase data files. The VDBF field name length may be up to 100 characters. Field names exceeding that limit will be truncated to the 100-character maximum.

NOTE: If you are a longtime Xbase developer, you may be tempted by the luxury of super-long field names. But bear the impacts in mind; should you decide to copy a vDBF to a local physical DBF, the field names must conform to the rules of the target DBF file.

- Use **SQLFieldName()** to retrieve the column name as returned by the ODBC data source.

Unnamed Column In SELECT Query

Most likely cause of this condition: You have specified an expression in your SELECT statement (rather than a column name) and you omitted an AS clause (which would specify a column name alias).

- The ODBC Connector will automatically generate the name "FIELD999", where 999 is a zero-padded value representing the field's order as listed in the SELECT query (Example: FIELD003).

Non-Alphanumeric Characters In Data Source

All non-alphabetic and non-numeric characters are converted to underscores. (Example: a column named "State and/or Province" will be available in Xbase as the field `state_and_or_province`).

Implementation Issues

This section of the documentation focuses on specific implementation issues of concern to developers integrating Xbase development with SQL databases and using ODBC as the middleware to achieve this goal.

Determining What Features Are Supported

Because the PlugSys development tools cover multiple platforms and environments, you'll sometimes want to confirm whether a particular feature is supported for your target environment. Here is how to determine the answer:

- If this is not a feature exclusively intended for the ODBC Connector, check the Max Language Reference for that language item and look for the Platforms checkboxes.
- First check for the operating system indicators. If the box is unchecked for your targeted operating system, then this feature is not available (this includes cases where you may be running MSP under a web server under that particular operating system.)

If this is a feature exclusively intended for ODBC, confirm whether the language feature is being used appropriately. Issues to watch for include:

- Are you attempting to use a NULL value? Is this supported by the language feature?
- Is the data type supported by the language feature?
- Are you submitting the proper datatype or correctly formatted value to the ODBC Connector?

Unsupported RDBMS Data Types

- The ODBC Connector supports all ODBC data types. However, some ODBC drivers may not support all of these. This is rarely an issue if you are using the most popular database drivers, but it sometimes occurs when the driver developer expects customers to perform custom data transformations in C or C++.
- The PlugSys ODBC Connector can only handle standard ODBC data types.
- If you submit a non-ODBC data type through the ODBC connector, the PlugSys database engine will fail to open the Virtual DBF. **SQLErrorMessage()** returns the message: "Field Type Not Supported: <fieldname>".

Conflicting Data Types (Xbase, SQL, ODBC)

The world of cross-platform/cross-database development can seem overwhelming to newcomers. The next paragraph asks you to apply common sense to your development effort. If you do, your effort will be much smoother.

SQL databases address a broader range of specialized data types than those available within classic Xbase. Before submitting a value to an Xbase function or to the ODBC connector, confirm whether that value conforms to a valid data type in the target context (Xbase, SQL, ODBC, your RDBMS implementation). This is equally true whether you wish to pass the value back from Xbase to SQL or you wish to format the value for presentation to the user.

- Start with the Max Language Reference to familiarize yourself with the baseline language feaures.
- Determine whether there is an appropriate conversion function for the job. (If this involves interaction with a SQL data type, consult the **ODBC Connector Language Reference** (on page 75).)
- See “ODBC Syntax For Database Interoperability” on page 70.
- See “NULL Values” on page 64.
- See “DateTime Data Type” on page 64.

DateTime Data Type

The DateTime data type is a common SQL data type that stores dates or times, or both.

These functions help process datetime values:

- **`ctot()`**returns a datetime value from a validly formatted character string.
- **`datetime()`** retrieves the system date and time as a datetime value.
- **`dtot()`** converts an Xbase date value to a valid SQL datetime value.
- **`hour()`** extracts the hour portion from a datetime value.
- **`minute()`** extracts the minutes portion from a datetime value.
- **`sec()`** extracts the seconds portion from a datetime value.
- **SQLFormat()** (on page 100)- returns a properly formatted ODBC value for use in SQL statements.

NULL Values

SQL databases are able to distinguish between an "empty value" and the non-entry of a value into a field. When a field has had no value entered, it is considered NULL regardless of its data type.

- Max introduces a **`.NULL.`** compile-time constant for assignment to variables or for submission to the data source.
- **SQLNull()** (on page 104)
- **`isnull()`** tests whether a variable or field contains a null value.
- **`unnull()`** returns the appropriate empty value of a vDBF field according to its data type when it contains a NULL value .

Null Values Are Not Acceptable For Routine Xbase Operations

Null values come from external data sources and are not a native Xbase data type.

- Xbase environment commands and functions will generate errors If they receive NULL values. Operations with null values are also not allowed.
- Null values in expressions generate runtime error messages.

Acceptable Contexts For NULL values

Null values are supported in the following operations:

- Variable assignment: You may create memory variables with Null data. This can be done by assigning a variable from a field or from the `.NULL.` constant.
- Submission to the data source.
- Data Output Operations: Max will display `.NULL.` as the content of Null variables/fields in all data output operations.
- `Empty()`: returns .T. for variables/fields containing NULL
- `Type()`: returns the actual data type
- `IsNull()`: allows for inspection and detection of variables/fields containing NULL data
- `UnNull( var )`: returns a guaranteed non-null value for the field/variable passed

Using Null Values

If you connect to a data source with nullable fields, it is your responsibility to handle these fields properly while in the Xbase environment.

Null values in expressions usually propagate. So if you request a column like `Upper(NAME+LASTNAME)` and one of the fields is NULL, the result will be NULL. So check for NULL values and handle them in your applications.

- `isnull()` detects which fields contain Null values.
- `unnull()` turns a null value vDBF field into the Xbase empty value for its data type.
- **SQLNull()** specifies how the ODBC Connector will return NULL values.

Classic Xbase applications do not know or care about NULL values. But as you've see in **Null Values Are Not Acceptable For Routine Xbase Operations** (on page 64), these will be hazardous if you do not handle them. Use `unnull()` to replace a NULL value with an empty value of the same data type. (See the Max Language Reference.)

The ODBC Connector exposes a function to perform this operation automatically upon data retrieval. SQLNull(.F.) will cause all Null data fields to return default empty values instead of NULL.

`.NULL.` is a compile-time constant that can be used to assign NULL values to variables.

```
Var = .null.
? Var   // Outputs ".NULL."
```

A Virtual DBF Cannot Be Indexed

Because a virtual DBF is not a physical entity, you cannot create index files from a vDBF.

Workarounds

- If you want to work with a physical version of the result set, use **`COPY`** to create a local DBF file and then index the newly created file using conventional methods.
- USE **`LOCATE`** and **`CONTINUE`**.

Index-Related Commands Are Not Supported

Any command or function that relies on index files to perform a function is not supported in a work area containing a vDBF. These are notable examples:

- SET SOFTSEEK
- SEEK
- INDEX ON
- SET INDEX TO

Instead of Indexes: Wise Coding For vDBF's

When you are using SQL databases, you should use a SQL SELECT statement to retrieve a set of valid records. (Even to retrieve one record.)

Approach 1

- Use the SQL ORDER BY clause (and also GROUP BY if you want) if you want to work with a sorted result set.
- Then use Xbase commands to navigate against the data in the sorted vDBF.

Approach 2

- Use the SQL SELECT statement to generate the result set into a vDBF.
- Use the Xbase **COPY TO** command to create a physical file.
- Open the newly create DBF file in a workarea.
- Index the newly created physical DBF file.
- Use **SEEK**.

We strongly recommend using **Approach 1**.

- With **Approach 2**, the larger the potential result set, the slower the response time as file Input/Output consumes time.
- The SQL database is very good at retrieving data fast. Then you can quickly navigate the VDBF to find subsets.
- If the result is a small enough set, the entire vDBF will be in memory. So, you can use LOCATE and CONTINUE to move through the data if you want.

Here is one approach to filter down the rows in a vDBF for various conditions:

```
* select the virtual DBF workarea
DO WHILE !EOF()
    ...           // if necessary change <condExpr>
    IF <condExpr> // the filtering condition
       ...
    ENDIF
    SKIP
ENDDO
```

Invalid Operations With A Virtual DBF

- APPEND
- REPLACE
- PACK
- ZAP

Using BROWSE/dbedit() With A Virtual DBF

- Disable hotkeys that trigger invalid operations (such as appending, replacing, etc.)
- Make sure to keep the **BROWSE**/**dbedit()** interface in read only mode.

Testing For An "Empty" Virtual DBF

A VDBF is opened even if the result set yields no rows.To determine whether a query returned rows into a vDBF, use:

- **COUNT**- returns 0 when the vDBF contains an empty result set.
- **LASTREC()**or **RECCOUNT()**- returns 0 when the vDBF contains an empty result set.
- **EOF()** and **BOF()**- Both of these are .T. when the vDBF contains an empty result set.
- You may also use the SQL count() function within a preliminary query. This will return one row. (Then monitor the value of that column.) If the value is > 0, then issue the desired query to retrieve the result set.

NOTE: **COUNT** and/or **LASTREC()/RECCOUNT()** should only be used when you can be certain that the maximum number of result rows is relatively small. Large result sets could require evaluating the full data set and impact performance.

Do not rely on:

- **USED()**
- **SQLError()**

to determine if the query returned rows. These functions are to test for error conditions rather than empty result sets.

Sources Of USE Errors With DBTYPE ODBC

- Network connection lost
- Locking problems
- Error in SQL query syntax

Testing for USE Errors

- Be sure to include **`IF USED()`** immediately after **`USE`** to test for success or failure.
- In the case of error, use **`SQLError()`** to retrieve further information.

vDBF And Header-Related Functions

The Virtual DBF is not a physical file and does not perform in quite the same manner as a classic Xbase DBF data table. The results will yield unreliable data or even generate errors.

- lastrec()/reccount()- This returns a valid numeric value indicating how many result rows were returned by the ODBC Connector. However, calling this function requires ODBC to fetch the entire result set. This could impact performance with large data sets. This fetching action will happen regardless of whether you have turned on **SQLFetchOnDemand()**. (See page 97.)

Workarounds

If you are requesting a nonstandard data type:

- exclude that field name or expression from the list of columns to be requested by the SELECT statement
- change the expression or database schema so that all columns are defined with valid ODBC datatypes

HEADER()

`header()` returns 0 when used with OXBC drivers that do not support it. This function is not available to ODBC database connections.

Fetching Behavior

The ODBC Connector by default turns OFF **SQLFetchOnDemand()**. When this feature is turned ON, the ODBC Connector retrieves data rows as needed (rather than loading the entire dataset.)

If your database permits fetching subsets, you may want to experiment with performance by turning this feature on. (See **Microsoft SQL Server** on page 70.)

Microsoft SQL Server

Microsoft SQL Server considers a fetch operation to be an uncompleted transaction. This could prevent updates to the record set.

Workaround

Allow the ODBC driver to retrieve all result rows in one fetch. Do not turn on **SQLFetchOnDemand()**. (The default for `SQLFetchOnDemand()` is .F.)

ODBC Syntax For Database Interoperability

Although the core syntax of SQL is common to most databases, every database vendor has unique syntax for the more advanced operations. Be aware of this and you can make the best choices for your own circumstances:

- Are you strongly committed to a particular database platform? If the answer is yes, then feel free to use the SQL syntax you already know.
- Are you uncertain or want to leave your options open to migrate later? Then try to use the most common subset of SQL. When you cannot stay within that subset, learn to use ODBC syntax. If you do this, you will be able to migrate because the ODBC driver will correctly interpret that syntax and present it to the current database in the syntax it understands.

NOTE: A fuller discussion of this topic can be found at **http://msdn.microsoft.com/library/psdk/dasdk/odch5v1v.htm**.

ODBC Escape Sequences

Use these to promote database independence or to submit Xbase data types to the appropriate data type for your SQL database.

NOTE: Further information on ODBC may be found at . Escape sequences are explored at **http://msdn.microsoft.com/library/psdk/dasdk/odch34df.htm**

Date and Time Values

The syntax for date/time escape sequence works like this:

{input-type 'value'}

where input-type is.

input-type (code)	Data Format
Date (d)	yyyy-mm-dd
Time (t)	hh:mm:ss
Timestamp/DateTime (ts)	

Examples Of Formatted

This is an example of an ODBC date escape sequence:

```
UPDATE Orders SET OpenDate={d '1995-01-15'} WHERE OrderID=1023
```

For simpler results, pass all the Xbase values to **SQLFormat()**. (See page 100.)

```
? SQLFormat(ctot("10:10:10")) // Result: {t 10:10:10}
? SQLFormat(ctot("10/10/00")) // Result: {d 1900-10-10}
```

- To ensure predictable results and to provide database interoperability, we recommend the use of **SQLFormat()**. (See page 100.)

NOTE: Further information on date/time escape sequences can be found at **http://msdn.microsoft.com/library/psdk/dasdk/odch5wfn.htm** and **http://msdn.microsoft.com/library/psdk/dasdk/odap4gfn.htm**.

Outer Joins

Note the difference in native syntax for outer joins in Oracle and Microsoft SQL Server and the database-independent equivalent for ODBC:

Oracle	SELECT STUDENT.SSN, FNAME, LNAME, CCODE, GRADE FROM STUDENT, GRADE WHERE STUDENT.SSN = GRADE.SSN(+)
Microsoft SQL Server	SELECT STUDENT.SSN, FNAME, LNAME, CCODE, GRADE FROM STUDENT LEFT OUTER JOIN GRADE ON STUDENT.SSN = GRADE.SSN
ODBC (interoperable)	SELECT STUDENT.SSN, FNAME, LNAME, CCODE, GRADE FROM {oj STUDENT LEFT OUTER JOIN GRADE ON STUDENT.SSN = GRADE.SSN}

NOTE: Further discussion of escape sequences for outer joins may be found at **http://msdn.microsoft.com/library/psdk/dasdk/odch1c37.htm**.

SQL Functions

SQL provides a set of functions for data transformation. A further discussion of the ODBC escape sequences can be found at **http://msdn.microsoft.com/library/psdk/dasdk/odch90xf.htm**.

LIKE Clauses

SQL provides the LIKE clause as a way of performing substring comparisons within a WHERE search predicate. A further discussion of the ODBC escape sequence can be found at **http://msdn.microsoft.com/library/psdk/dasdk/odch478z.htm**.

SQL Procedures

Many SQL databases provide a facility for precompiling, storing and executing SQL operations on demand. Stored procedures are common to the commercial databases. A discussion of the ODBC escape sequences for these can be found at **http://msdn.microsoft.com/library/psdk/dasdk/odch7h4j.htm**.

Frequently Asked Questions

Q: Let's say that I open an ODBC connection in work area 5 and then successfully load a vDBF there. What happens if I open something else in that same work area? Is this the same as issuing a **SQLDisconnect()**?

A: No, it is not the same thing. In your example, the first vDBF is closed, all related data buffers are released. Then the new vDBF is opened using the existing ODBC connection. (Please remember that re-opening connections is resource intensive. Read documentation for **SQLDisconnect()**. (See page 89.)

Q: Is SQLRowCount() the same as lastrec()/reccount()?

A: **SQLRowCount()** is used only for SQL operations like INSERT and UPDATE that do NOT return record sets. This is one way to determine whether the operation was successful because the RDBMS typically indicates whether any rows were affected by the operation. Don't use **`SQLRowCount()`** for SELECT operations! The way to get the count is to call lastrec()/reccount().

Q: Does a vDBF store NULL values coming from a SQL database?

A: vDBF tables support NULL values. Use **SQLNull()** to control the behavior.

Q: Can I store NULL values in a DBF data table?

- The native PlugSys DBF file format does not support NULL values.
- Some DBF implementations do support NULL values. It is possible to copy from a vDBF to a physical DBF file. If you use **`COPY`** to create a physical version of the table, make sure the target database is capable of storing NULLs. (PlugSys offers the CodeBase OXBC driver to support NULL storage for the major DBF file formats.)

More Resources

SQL

- http://sqlcourse.com/
- http://www.sqlcourse2.com/
- http://users.neca.com/ltruett/sql.html
- http://www.sql-zone.com/sites.asp
- http://www.cs.unibo.it/~ciaccia/COURSES/RESOURCES/SQLTutorial/sqlcont.htm
- http://www.dcs.napier.ac.uk/~andrew/qisq/

ODBC

- http://msdn.microsoft.com/library/psdk/dasdk/odin8w4s.htm
- http://www.microsoft.com/data/odbc/
- http://www.microsoft.com/data/doc.htm#odbc
- http://www.unixodbc.com

ODBC Connector Language Reference

This chapter provides an overview of functions relevant to ODBC operations using the PlugSys ODBC Connector.

General Language Constructs

The following are commands and functions appearing in the Max Language Reference. However they are of additional interest when used in accessing SQL databases.

NOTE: Be sure to consult the Max Language Reference for the most complete information on these.

USE

- opens a DBF data table, its associated memo file and its associated indexes.
- passes a SQL SELECT statement to an ODBC data source and opens a virtual DBF in the specified work area. ***(Requires PlugSys ODBC Connector.)***

Platforms:	
☑DOS32/W **☑Linux**	USE [data_table] [IN work_area_number / NEW] [ALIAS database_alias] [INDEX idxfile_1 [KEY Index_Exp1 [FOR ForCond_1] [UNIQUE]] [, idxfile_2 [KEY Index_Exp2 [FOR ForCond_2] [UNIQUE]] [, ...]]] [DBTYPE <datasource_type>] [ORDER index_tag] [QUERY sql_query] [NOUPDATE] [EXCLUSIVE] [SHARED]
☑OXBC: ODBC	*USE <tablename|viewname> [QUERY <cSqlQuery>] [IN <nWorkArea>] [ALIAS <cAlias>] [DBTYPE <dbtype>]

NOTE: *This is designed to document how **USE** works when accessing ODBC SQL data sources in conjunction with the PlugSys ODBC Connector. Please see the Max Language Reference for baseline documentation of this command and its Xbase-oriented behavior.

isnull()

tests whether a variable or field contains a null value.

```
isnull( <VarField> )
```

Platforms:

☑ **OXBC: ODBC**

VarField
any variable name or field name

Return Value:

returns .T. if the variable/field contains a null value; otherwise returns .F.

unnull()

returns the appropriate empty value of a variable or vDBF field according to its data type.

```
unnull( FieldVar )
```

Platforms:

☑ **OXBC: ODBC**

FieldVar
fieldname or variable name

Return Value:

- If the variable/field passed to UNNULL( ) contains a null, UNNULL( ) returns the appropriate “empty value.”
- The default empty value is produced according to the type of the variable/field provided.
- If a variable was assigned with the .NULL. constant, UNNULL( ) returns .F.

Remarks:

- The **QUERY** clause is only valid when applied to ODBC connections with SQL databases.
- Whenever you know you will require a join for common operations, consider creating a SQL view at the database. This is handy for frequent reuse. Do this also insulates your application code from any changes to the database schema. (As long as the viewname returns the same columnnames and data types, you can make significant changes to the underlying SQL database schema while preserving the code in your application.) And most database engines provide improved performance. Once you have created the join, your application can open the connection and call the viewname: **USE ViewName DBTYPE ODBC**.

Examples:

```
* This exploits default behaviors and values:
* ODBC Connector opens view or table called “productlist”
* and assigns the default alias  “productlist”
* query is omitted: connector submits “SELECT * from productlist”
USE ProductList DBTYPE ODBC // Alias is "PRODUCTLIST"
                            // Use default SELECT * query
                            // "SELECT * from ProductList"

* Submit an explicit query
USE ProductList QUERY [SELECT ProdID, PriceUSD FROM Products] in 2
                            // uses default alias “productlist”
                            // returned set has 2 columns
```

Also see:

SQL-Specific Language Constructs

The following are relevant only when you are using the ODBC Connector. These language elements are not available unless the ODBC Connector has been loaded.

SQLAutoCommit()

used to retrieve or change the AutoCommit mode for an ODBC connection.

```
SQLAutoCommit( newState )
```

Platforms:

☑ **OXBC: ODBC**

newState
a logical value (.T. or .F.) indicating the new state of AutoCommit.

Return Value:

the current AutoCommit state. If the function is used to set a new state, it returns the previous state.

Remarks:

- You must load the ODBC connector with **`SET DBTYPE TO ODBC`** for this function to be available to the application at runtime.
- When AutoCommit mode is ON (default), each statement is committed immediately after it is executed. When in manual-commit mode, the application must explicitly commit or roll back transactions with **SQLAutoCommit()** or **SQLRollback()**.

The default value for AutoCommit is ON.

Example:

```
SET DBTYPE TO ODBC    // Load ODBC Connector to make SQLxxx()
functions available
SQLAutoCommit(.F.)    // Turning off AutoCommit mode:
Connected = SQLConnect("MS-SQLServer", "sa", "plugsys")

if ! Connected
   FatalError("Could Not Connect to ODBC Data Source",
SQLErrorMessage())
end

USE CUSTOMERS QUERY [SELECT ID, NAME, CITY, STATE from CUSTOMERS]

SQLExec([UPDATE CUSTOMERS SET name = 'new name' WHERE ID =
']+mID+['])
.
.
.
if TransactionCompleted
    SQLCommit()          // Commit transaction
else
    SQLRollBack()        // Roll back
end
.
.
.
close
SQLDisconnect()
```

Also see:

SQLCommit()

commits all transactions on the current work area.

```
SQLCommit( )
```

Return Value:

The function returns .T. if successful or .F. upon failure.

Notes:

This function is only effective when **SQLAutoCommit()** is OFF.

Remarks:

- You must load the ODBC connector with **SET DBTYPE TO ODBC** for this function to be available to the application at runtime.
- Use **SQLError()** and **SQLErrorMessage()** to retrieve information about the error upon failure.
- The virtual DBF is a “snapshot” of the data at the time the last query was executed. Your application is not notified about subsequent changes occur to the SQL database.
- When AutoCommit mode is OFF, the application must explicitly commit or roll back transactions with **SQLCommit()** or SQLRollBack( ). (The default value for **SQLAutoCommit()** is ON.)

Example:

```
SET DBTYPE TO ODBC // Load ODBC Connector, make SQLxxx() funcs
available

SQLAutoCommit(.F.)    // Turning off AutoCommit mode:

Connected = SQLConnect("MS-SQLServer", "sa", "plugsys")

if ! Connected
   FatalError("Could Not Connect to ODBC Data Source",
SQLErrorMessage())
end

USE CUSTOMERS QUERY [SELECT ID, NAME, CITY, STATE from CUSTOMERS]

SQLExec([UPDATE CUSTOMERS SET name = 'new name' WHERE ID =
']+mID+['])
.
.
.
if TransactionCompleted
    SQLCommit()          // Commit transaction
else
    SQLRollBack()        // Roll back
end
.
.
.
close
SQLDisconnect()
```

Also see:

SQLConnect()

establishes an ODBC connection with the specified ODBC data source name.

```
SQLConnect( DataSourceName [, UserID [, Password ] ] )
```

Platforms:

☑ **OXBC: ODBC**

DataSourceName
is the ODBC data source as defined using an ODBC management tool.

UserID
If the data source requires authentication, supply a valid RDBMS user account id as a string. (Optional)

Password
If the data source requires runtime authentication, supply a valid password for the UserID as a string. (optional)

Return Value:

Returns returns .T. if the connection was successfully established or .F. if an error occurred.

Remarks:

- You must load the ODBC connector with **`SET DBTYPE TO ODBC`** for this function to be available to the application at runtime.
- If the ODBC driver for this data source requires more parameters than provided for in the above syntax, use **SQLStringConnect()** instead.
- Use **SQLDisconnect()** and **SQLErrorMessage()** to retrieve information about the error upon failure.
- Before being able to connect to a data source, it is necessary to run an ODBC manager/ configuration utility to create and/or configure a data source.
- The Data Source Name (DSN) is the name of the ODBC connection to be used. An ODBC data source contains all details about the connection.
- Data sources that describe connections with RDBMSs are capable of storing information about the authentication method to be used. This may override the UserID and Password parameters. If the data source is not properly configured, the connection will not work.

Certain DBMS engines (such as Microsoft SQL Server) offer the option of logging in using operating system authentication. If your application is running on a remote server and particularly if you are running separate operating systems, it may be necessary to configure the SQL server to use the native DBMS authentication instead.

SQLConnect() establishes a connection within the current work area. After the connection has been successfully established, the application may open a virtual DBF with the **USE** command, specifying the SQL SELECT query necessary to return the result table to be mapped into a Virtual DBF.

SQLCommit() establishes a connection on the current work area. After the connection has been successfully established, the application may open a virtual DBF with **USE**.

The connection with the ODBC data source is not terminated if the Virtual DBF is closed with **CLOSE** or **USE**. Once the connection has been established, it remains available until you issue a **SQLDisconnect()**. Establishing a connection demands more processing than most other operations. So keep connections open as much as possible.

Call **SQLDisconnect()** only when you absolutely need to explicitly close a connection.

ODBC connections are automatically disconnected when an application (or web connection) terminates, so it is not mandatory to call **SQLDisconnect()** in many cases.

Example:

```
SET DBTYPE TO ODBC // Load ODBC Connector, make SQLxxx() funcs
available

SELECT 10    // Choosing a work area
Connected = SQLConnect("SQLServer", "sa", "plugsys")

if ! Connected
   FatalError("Could Not Connect to ODBC Data Source",
SQLErrorMessage())
end
close
SQLDisconnect()
```

Also see:

SQLDisconnect()

disconnects an ODBC connection in the current work area established previously with SQLConnect( ).

```
SQLDisconnect()
```

Platforms:

☑ **OXBC: ODBC**

Return Value:

None

Remarks:

You must load the ODBC connector with **SET DBTYPE TO ODBC** for this function to be available to the application at runtime.

SQLCommit( ) establishes a connection on the current work area. After the connection has been successfully established, the application may open a virtual DBF with **USE**.

The connection with the ODBC data source is not terminated if the Virtual DBF is closed with **CLOSE** or **USE**. Once the connection has been established, it remains available until you issue a **SQLDisconnect()**. Establishing a connection demands more processing than most other operations. So keep connections open as much as possible.

Call **SQLDisconnect()** only when you absolutely need to explicitly close a connection.

ODBC connections are automatically disconnected when an application (or web connection) terminates, so it is not mandatory to call **SQLDisconnect()** in many cases.

Example:

```
SET DBTYPE TO ODBC  // Load ODBC Connector, make SQLxxx() funcs
available

SELECT 10    // Choosing a work area
Connected = SQLConnect("SQLServer", "sa", "plugsys")

if ! Connected
   FatalError("Could Not Connect to ODBC Data Source",
SQLErrorMessage())
end

close
SQLDisconnect()
```

Also see:

SQLError()

retrieves the ODBC error code from the last ODBC operation performed.

```
SQLError()
```

Platforms:

☑ **OXBC: ODBC**

Return Value:

Returns a status code indicating success or error, according to the following table:

-1 = Error

0 = Success

1 = Success with info (Warning)

Remarks:

- You must load the ODBC connector with **`SET DBTYPE TO ODBC`** for this function to be available to the application at runtime.
- Use **`SQLError()`** to determine if the last operation performed by the ODBC Connector succeeded. Most operations return codes indicating success or failure. SQLError( ) can be used concurrently to retrieve further information. Operations may succeed but generate warning messages. **`SQLError()`** returns 1 indicating that the operation succeeded but generated warning messages. Use **SQLErrorMessage()** to retrieve further information.
- Error values are always returned as negative numbers, so SQLError( ) >= 0 indicates that the operation was successfully executed.

Example:

```
SET DBTYPE TO ODBC // Load ODBC Connector, make SQLxxx() funcs
available

Connected = SQLConnect("SQLServer", "sa", "plugsys")

if SQLError() < 0
   ? "ERROR: Failed to connect to SQLServer data source!"
   ? "Error Description = " + SQLErrorMessage()
   Quit
end

if SQLError() > 0
   ? "WARNING: Successfully connected to SQLServer"
   ? "But driver issued a warning: " + SQLErrorMessage()
   wait "Press any key to proceed..."
end
.
.
.
close
SQLDisconnect()
```

Also see:

SQLErrorMessage()

retrieves the error string triggered by the last database operation through the ODBC connector.

```
SQLErrorMessage()
```

Platforms:

☑ **OXBC: ODBC**

Return Value:

Returns one or more error messages separated by semi-colons (;)

Remarks:

You must load the ODBC connector with **SET DBTYPE TO ODBC** for this function to be available to the application at runtime.

When an operation fails, the ODBC Connector notes the original ODBC error code. This value is returned by **SQLDisconnect()**. **SQLErrorMessage()** returns more detailed, user-readable information about the error.

Example:

```
SET DBTYPE TO ODBC  // Load ODBC Connector, make SQLxxx() funcs
available
Connected = SQLConnect("SQLServer", "sa", "plugsys")

if ! Connected
   ? "ERROR: Could not connect to SQLServer data source!"
   ?
   ? "ODBC State/Error Code = " | SQLError()
   ? "Error Description = " | SQLErrorMessage()
   Quit
end

RecordSet = [SELECT NAME, CITY, STATE from CUSTOMERS]
USE &RecordSet alias CUST
.
.
.
select CUST
bResult = SQLExec([UPDATE CUSTOMERS set NAME = '] + mNewName + ['
WHERE NAME = '] + CUST->NAME + ['])
if !bResult
   FatalError("Could not update customers table: ",
SQLErrorMessage())
end

? "Total Records Updated: " | SQLRowCount()
.
.
.

close
SQLDisconnect()
```

Also see:

SQLExec()

executes a SQL query in the current work area.

```
SQLExec( SQLStatement )
```

Platforms:

☑ **OXBC: ODBC**

SQLStatement
a string containing a valid SQL operation

Return Value

returns .T. if the SQL statement was successfully executed or .F. upon failure.

Notes:

- Do not use SQLExec() to submit statements that return result sets! Therefore SQL SELECT statements are not allowed.. Submitted SQL SELECT statements with the Xbase **USE** command. This produces a Virtual DBF for your application to access as needed.
- Do not use this function to invoke commit and rollback operations. Instead use **SQLCommit()** and **SQLRollback()** or **SQLAutoCommit()**.

Remarks:

- You must load the ODBC connector with **SET DBTYPE TO ODBC** for this function to be available to the application at runtime.
- A connection with an ODBC data source must have been successfully established with **SQLConnect()** or **SQLStringConnect()** first or **SQLExec()** will fail.
- Call **SQLRowCount()** to determine the number of rows affected by the SQL statement after execution.
- Use **SQLError()** and **SQLErrorMessage()** to retrieve information about errors.
- **SQLRefresh()** should be called immediately following **SQLExec()** to read any data changes into the local data buffer.

Example:

```
SET DBTYPE TO ODBC  // Load ODBC Connector, make SQLxxx() funcs
available

Connected = SQLConnect("SQLServer", "sa", "plugsys")
if ! Connected
   FatalError("Could Not Connect to ODBC Data Source",
SQLErrorMessage())
end

RecordSet = [SELECT NAME, CITY, STATE from CUSTOMERS]
USE &RecordSet alias CUST
.
.
.
select CUST
bResult = SQLExec([UPDATE CUSTOMERS set NAME = '] + mNewName + ['
WHERE NAME = '] + CUST->NAME + ['])
if !bResult
   FatalError("Could not update customers table: ",
SQLErrorMessage())
end

? "Total Records Updated: " | SQLRowCount()
.
.
.

close
SQLDisconnect()
```

Also see:

SQLFetchOnDemand()

specifies whether the ODBC connector will retrieve result rows as needed (or fetches the entire data set when SQL SELECT is executed).

```
SQLFetchOnDemand( lAllowFlexibleFetching )
```

SQLFieldName()

retrieves the original field/column name returned by the ODBC driver.

```
SQLFieldName( nFieldNumber )
```

Platforms:

☑ **OXBC: ODBC**

nFieldNumber
is a numeric expression of the column number in the sequence submitted to the ODBC connector.

Return Value:

Returns the original field name (column name) returned by the ODBC driver.

Remarks:

You must load the ODBC connector with **SET DBTYPE TO ODBC** for this function to be available to the application at runtime.

Xbase semantics may differ from the ODBC data source field name conventions. And some queries may not return names for columns. The ODBC Connector inspects column names from the data source before mapping them into Virtual DBF field names. The field order is preserved to guarantee correspondence between Xbase field names and the columns in the query result set.

Example:

```
SET DBTYPE TO ODBC  // Load ODBC Connector, make SQLxxx() funcs
available
Connected = SQLConnect("SQLServer", "sa", "plugsys")

if ! Connected
   FatalError("Could Not Connect to ODBC Data Source: ",
SQLErrorMessage())
end

RecordSet = [SELECT NAME, CITY, STATE from CUSTOMERS]
USE &RecordSet alias CUST
nFields = afields()
declare FieldNames[nFields]
afields(FieldNames)

for n = 1 to nFields
    ? "Xbase field " + FieldNames[n] + " = SQL column " +
SQLFieldName(n)
next
close
SQLDisconnect()
```

Also see:

SQLFormat()

returns a properly formatted ODBC value for use in SQL statements.

```
SQLFormat( value )
```

Platforms:

☑ **OXBC: ODBC**

value
a character, date, datetime, numeric or logical value to be formatted.

Return Value:

a valid ODBC escape string appropriate to the input data format and data type.

Remarks:

- You must load the ODBC connector with **`SET DBTYPE TO ODBC`** for this function to be available to the application at runtime.

SQLFormat( ) follows the ODBC standard for Date, Time and DateTime values, returning values that are guaranteed valid for submission to any data source.

Formatting criteria:

- Character values are returned within single-quotes (')
- Numeric values are formatted with 6-digit precision.
- Logical values are formatted as 0 or 1
- Date and DateTime fields are formatted as ODBC Escape Sequences {t }, {d } or {ts } (See **ODBC Escape Sequences** on page 71.)
- The default precision for numeric values is 6 digits. This cannot be changed. If your app needs a different format, do not use SQLFormat( ) for that value.

NOTE: We strongly recommend the use of **`SQLFormat()`** when building SQL statements to be submitted through **`SQLExec()`**.

Example:

```
SET DBTYPE TO ODBC // Load ODBC Connector, make SQLxxx() funcs
available

set century on

? SQLFormat(10.2093)            // Result: 10.209300  (Default
Precision = 6 digits)
? SQLFormat("PlugSys")          // Result: 'PlugSys'
? SQLFormat(datetime())         // Result: {ts 2000-09-12
14:14:23}
? SQLFormat(ctot("10:10:10"))   // Result: {t 10:10:10}
? SQLFormat(ctot("10/10/00"))   // Result: {d 1900-10-10}
? SQLFormat(.F.)                // Result: 0
```

Also see:

SQLMaxVarLen()

restricts the maximum data length when variable length fields are retrieved from a given ODBC data source.

```
SQLMaxVarLen( newMaxLength )
```

Platforms:

☑ **OXBC: ODBC**

newMaxLength
is a numeric expression speficying the maximum length of variable length fields.

Return Value:

Returns the current setting. If the function is used to set a new value, it returns the previous setting.

Remarks:

- You must load the ODBC connector with **`SET DBTYPE TO ODBC`** for this function to be available to the application at runtime.
- Use **SQLError()** and **SQLErrorMessage()** to retrieve information about the error upon failure.

Most RDBMSs support a series of variable length fields. These fields may contain text, character or binary data. The most commonly used variable length SQL datatype is varchar.

SQLMaxVarLen( ) allows the application to specify a maximum length for all of the variable-length fields returned from a given data source. This the ODBC Connector, when accessing such fields, will download only that number of bytes. If necessary, the field contents will be truncated to the maximum length you specify (newMaxLength).

This function is a tool for conserving memory. When a query is returned as a virtual DBF, the data is cached in memory. The contents of all variable-length database fields can take an enormous chunk of memory-- especially if the query returns a high number of rows and/or variable length fields.

The default value for MaxVarLen is 512K.

Example:

```
SET DBTYPE TO ODBC    // Load ODBC Connector to make SQLxxx()
functions available

Connected = SQLConnect("MS-SQLServer", "sa", "plugsys")

if ! Connected
   FatalError("Could Not Connect to ODBC Data Source",
SQLErrorMessage())
end

? "Preventing huge memos from being downloaded"
? "Displaying only the first page of each article"
SQLMaxVarLen(2048)

RecordSet = [SELECT AUTHOR, ARTICLE from KNOWLEDGEBASE]
USE &RecordSet alias KB
.
.
.
close
SQLDisconnect()
```

Also see:

SQLNull()

specifies how the ODBC Connector will return NULL values.

```
SQLNull( newState )
```

Platforms:

☑ **OXBC: ODBC**

newState
a logical value (.T. or .F.) indicating the new state.

Return Value:

returns the current state. If the function is used to set a new state, it returns the previous state.

Remarks:

- You must load the ODBC connector with **`SET DBTYPE TO ODBC`** for this function to be available to the application at runtime.
- SQLNull(.F.) instructs the ODBC Connector to automatically convert all NULL values to the appropriate empty values for the data type of the field. For instance, NULL character fields return an empty string (""), numeric fields return 0.
- The status of SQLNull( ) can be changed at any time.
- The default value for SQLNull is ON.

Example:

```
SET DBTYPE TO ODBC    // Load ODBC Connector to make SQLxxx()
functions available

SelectStmt := [ SELECT NAME, CITY, STATE, COUNTRY  FROM CUSTOMERS
]

Connected = SQLConnect("MyODBCConnection", "max", "plugsys")

if ! Connected
   FatalError("Could Not Connect to ODBC Data Source",
SQLErrorMessage())
end

use CUSTOMERS query SelectStmt
if ! USED( )
  FatalError("Could Not Retrieve Data", SQLErrorMessage())
end

SQLNull(.F.)    // NULL fields are returned as empty/printable
values

do while ! eof()
   ? CUSTOMERS->NAME
   SKIP
end

close
SQLDisconnect()
```

Also see:

SQLRefresh()

reissues the ODBC connection's query and updates the Virtual DBF image with current data.

```
SQLRefresh()
```

Platforms:

☑ **OXBC: ODBC**

Return Value:

- If the SQL query for the virtual DBF (VDBF) in the current work area was successfully executed, returns .T.
- Returns .F. upon failure of execution.

Remarks:

- You must load the ODBC connector with **`SET DBTYPE TO ODBC`** for this function to be available to the application at runtime.
- Use **SQLError()** and **SQLErrorMessage()** to retrieve information about the error upon failure.
- The virtual DBF is a "snapshot" of the data at the time the last query was executed. Your application is not notified about subsequent changes occur to the SQL database.
- **`SQLRefresh()`** should be called immediately following **SQLExec()** to read any data changes into the local data buffer.
- Call **SQLRowCount()** to determine the number of rows affected by the SQL statement after execution.

Example:

```
SET DBTYPE TO ODBC    // Load ODBC Connector to make SQLxxx()
functions available
Connected = SQLConnect("SQLServer", "sa", "plugsys")

if ! Connected
   FatalError("Could Not Connect to ODBC Data Source",
SQLErrorMessage())
end

RecordSet = [SELECT ID, NAME, CITY, STATE from CUSTOMERS]
USE &RecordSet alias CUST
.
.
.
UpdateStmt = [UPDATE customers SET NAME = '] + mNewName + [' WHERE
ID = '] + CUST->ID + [']
bResult = SQLExec(UpdateStmt)
if !bResult
   FatalError("Could not update customers table: ",
SQLErrorMessage())
end

? "Total Records Updated: " | SQLRowCount()
? "Showing cached records: VDBF cache still not updated"
go top
do while ! eof()
   ? recno(), CUST->ID, CUST->NAME, CUST->CITY, CUST->STATE
   SKIP
end

? "Updating the Virtual DBF..."
SQLRefresh()
? "Now showing all records including latest changes:"
go top
do while ! eof()
   ? recno(), CUST->ID, CUST->NAME, CUST->CITY, CUST->STATE
   SKIP
end
.
```

```
.
.
close
SQLDisconnect()
```

Also see:

SQLRollback()

rolls back all transactions on the current work area. It requires AutoCommit mode to be turned OFF.

```
SQLRollBack()
```

Platforms:

☑ **OXBC: ODBC**

Return Value:

Returns .T. if successful or .F. upon failure.

Remarks:

- You must load the ODBC connector with **`SET DBTYPE TO ODBC`** for this function to be available to the application at runtime.
- Use **SQLError()** and **SQLErrorMessage()** to retrieve information about the error upon failure.
- The virtual DBF is a “snapshot” of the data at the time the last query was executed. Your application is not notified about subsequent changes occur to the SQL database.
- When AutoCommit mode is OFF, the application must explicitly commit or roll back transactions with **SQLCommit()** or SQLRollBack( ). (The default value for **SQLAutoCommit()** is ON.)

Example:

```
SET DBTYPE TO ODBC    // Load ODBC Connector to make SQLxxx()
functions available

SQLAutoCommit(.F.)    // Turning off AutoCommit mode:

Connected = SQLConnect("MS-SQLServer", "sa", "plugsys")

if ! Connected
   FatalError("Could Not Connect to ODBC Data Source",
SQLErrorMessage())
end

USE CUSTOMERS QUERY [SELECT ID, NAME, CITY, STATE from CUSTOMERS]

SQLExec([UPDATE CUSTOMERS SET name = 'new name' WHERE ID =
']+mID+['])
.
.
.
if TransactionCompleted
    SQLCommit()          // Commit transaction
else
    SQLRollBack()        // Roll back
end
.
.
.
close
SQLDisconnect()
```

Also see:

SQLRowCount()

identifies the number of rows affected by the the last invoked SQLExec( ) operation.

```
SQLRowCount()
```

Platforms:

☑ **OXBC: ODBC**

Return Value:

Returns the number of rows affected by the the last invoked **SQLExec()** operation.

Remarks:

- You must load the ODBC connector with **SET DBTYPE TO ODBC** for this function to be available to the application at runtime.
- Call SQLRowCount( ) to determine the number of rows affected by the last executed SQL INSERT OR UPDATE operation.
- SQLRowCount() is used only for SQL operations like INSERT and UPDATE that do NOT return record sets. This is one way to determine whether the operation was successful because the RDBMS typically indicates whether any rows were affected by the operation.
- Don't use SQLRowCount() to test the result of SELECT operations. The way to get the count is to call lastrec() or reccount( ).
- Use **SQLDisconnect()** and **SQLErrorMessage()** for diagnostic purposes if the function fails.

Example:

```
Connected = SQLConnect("SQLServer", "sa", "plugsys")

if ! Connected
   FatalError("Could Not Connect to ODBC Data Source",
SQLErrorMessage())
end

RecordSet = [SELECT NAME, CITY, STATE from CUSTOMERS]
USE &RecordSet alias CUST
.
.
.
select CUST
bResult = SQLExec([UPDATE CUSTOMERS set NAME = '] + mNewName + ['
WHERE NAME = '] + CUST->NAME + ['])
if !bResult
   FatalError("Could not update customers table: ",
SQLErrorMessage())
end

? "Total Records Updated: " | SQLRowCount()
.
.
.
close
SQLDisconnect()
```

Also see:

SQLState()

returns SQLSTATE value strings as defined by X/Open Data Management: SQL Version 2.

```
SQLState()
```

Platforms:

☑ **OXBC: ODBC**

Return Value:

Returns a string that contain five characters indicating the SQLSTATE, according to the table below.

Remarks:

- You must load the ODBC connector with **`SET DBTYPE TO ODBC`** for this function to be available to the application at runtime.

SQLSTATE Codes

- The character string value returned for an SQLSTATE consists of a two-character class code followed by a three-character subclass value.
- A class value of “01” indicates a warning.
- Class values other than 01 and IM indicate an error.
- The class IM is specific to warnings and errors generated by ODBC itself.
- The subclass value “000” in any class indicates that there is no subclass for that SQLSTATE.
- The assignment of class and subclass values is defined by the ANSI SQL92.standard.

SQLSTATE Value	Status
00000	**Success**
01000	General warning
01001	Cursor operation conflict
01002	Disconnect error
01003	NULL value eliminated in set function
01004	String data, right truncated
01006	Privilege not revoked
01007	Privilege not granted
01S00	Invalid connection string attribute
01S01	Error in row
01S02	Option value changed
01S06	Attempt to fetch before the result set returned the first rowset
01S07	Fractional truncation
01S08	Error saving File DSN
01S09	Invalid keyword
07001	Wrong number of parameters
07002	COUNT field incorrect
07005	Prepared statement not a *cursor-specification*
07006	Restricted data type attribute violation
07009	Invalid descriptor index
07S01	Invalid use of default parameter

08001	Client unable to establish connection
08002	Connection name in use
08003	Connection does not exist
08004	Server rejected the connection
08007	Connection failure during transaction
08S01	Communication link failure
21S01	Insert value list does not match column list
21S02	Degree of derived table does not match column list
22001	String data, right truncated
22002	Indicator variable required but not supplied
22003	Numeric value out of range
22007	Invalid datetime format
22008	Datetime field overflow
22012	Division by zero
22015	Interval field overflow
22018	Invalid character value for cast specification
22019	Invalid escape character
22025	Invalid escape sequence
22026	String data, length mismatch
23000	Integrity constraint violation
24000	Invalid cursor state
25000	Invalid transaction state
25S01	Transaction state
25S02	Transaction is still active
25S03	Transaction is rolled back
28000	Invalid authorization specification
34000	Invalid cursor name
3C000	Duplicate cursor name
3D000	Invalid catalog name
3F000	Invalid schema name
40001	Serialization failure

40002	Integrity constraint violation
40003	Statement completion unknown
42000	Syntax error or access violation
42S01	Base table or view already exists
42S02	Base table or view not found
42S11	Index already exists
42S12	Index not found
42S21	Column already exists
42S22	Column not found
44000	WITH CHECK OPTION violation
HY000	General error
HY001	Memory allocation error
HY003	Invalid application buffer type
HY004	Invalid SQL data type
HY007	Associated statement is not prepared
HY008	Operation canceled
HY009	Invalid use of null pointer
HY010	Function sequence error
HY011	Attribute cannot be set now
HY012	Invalid transaction operation code
HY013	Memory management error
HY014	Limit on the number of handles exceeded
HY015	No cursor name available
HY016	Cannot modify an implementation row descriptor
HY017	Invalid use of an automatically allocated descriptor handle
HY018	Server declined cancel request
HY019	Non-character and non-binary data sent in pieces
HY020	Attempt to concatenate a null value
HY021	Inconsistent descriptor information
HY024	Invalid attribute value
HY090	Invalid string or buffer length

HY091	Invalid descriptor field identifier
HY092	Invalid attribute/option identifier
HY095	Function type out of range
HY096	Invalid information type
HY097	Column type out of range
HY098	Scope type out of range
HY099	Nullable type out of range
HY100	Uniqueness option type out of range
HY101	Accuracy option type out of range
HY103	Invalid retrieval code
HY104	Invalid precision or scale value
HY105	Invalid parameter type
HY106	Fetch type out of range
HY107	Row value out of range
HY109	Invalid cursor position
HY110	Invalid driver completion
HY111	Invalid bookmark value

HYC00	Optional feature not implemented
HYT00	Timeout expired
HYT01	Connection timeout expired
IM001	Driver does not support this function
IM002	Data source name not found and no default driver specified
IM003	Specified driver could not be loaded
IM004	Driver's **SQLAllocHandle** on SQL_HANDLE_ENV failed
IM005	Driver's **SQLAllocHandle** on SQL_HANDLE_DBC failed
IM006	Driver's **SQLSetConnectAttr** failed
IM007	No data source or driver specified; dialog prohibited
IM008	Dialog failed
IM009	Unable to load translation DLL
IM010	Data source name too long
IM011	Driver name too long
IM012	DRIVER keyword syntax error
IM013	Trace file error
IM014	Invalid name of File DSN
IM015	Corrupt file data source

Example:

```
SET DBTYPE TO ODBC    // Load ODBC Connector to make SQLxxx()
functions available

Connected = SQLConnect("SQLServer", "sa", "plugsys")

if SQLError() < 0
   ? "ERROR: Failed to connect to SQLServer data source!"
   ? "Error Description = " + SQLErrorMessage()
   ? "SQLSTATE = " + SQLState()
   Quit
end

if SQLError() > 0
   ? "WARNING: Successfully connected to SQLServer"
   ? "But driver issued a warning: " + SQLErrorMessage()
   ? "SQLSTATE = " + SQLState()
   wait "Press any key to proceed..."
end
.
.
.
close
SQLDisconnect()
```

Also see:

SQLStringConnect()

establishes an ODBC connection with the specified ODBC data source name. (To be used when the data source depends on an ODBC driver requiring more data parameters than expected by **SQLConnect()**.)

```
SQLStringConnect( ODBCConnectionString )
```

Platforms:

☑ **OXBC: ODBC**

ODBCConnectionString
the ODBC connection string as described in ODBC's specification. This should include:

- data source name (DSN)
- userID (one more more)
- password (one or more)
- other parameters required by the ODBC driver
- this is model syntax map for ODBC connection strings:
  "DSN=data-source-name[;SERVER=server-name][;PWD=passwordString] [;UID=value] [;<Attribute>=<value>]"

Return Value:

SQLConnect returns .T. if the connection was successfully established or .F. if an error occurred.

Remarks:

- You must load the ODBC connector with **SET DBTYPE TO ODBC** for this function to be available to the application at runtime.
- Use **SQLDisconnect()** and **SQLErrorMessage()** for diagnostic purposes if the function fails.
- With most ODBC drivers, you can omit certain parameters and the user will be prompted for these at runtime. (Do not depend on this behavior for applications requiring total automated operation such as web applications.)

The Data Source Name (DSN) is the name of the ODBC connection to be used (or in the case of a file DSN, its filename). It contains all details about the connection. Before being able to connect to a data source, it is necessary to run the ODBC configuration utility to create or configure a data source. Data sources that describe connections with RDBMSs usually contain information about the authentication method to be used. This may override the UserId and Password parameters. If the data source is not properly configured the connection will not work.

Certain DBMS engines (such as Microsoft SQL Server) offer the option of logging in using operating system authentication. If your application is running on a remote server and particularly if you are running separate operating systems, it may be necessary to configure the SQL server to use the native DBMS authentication instead.

SQLConnect() establishes a connection within the current work area. After the connection has been successfully established, the application may open a virtual DBF with the **USE** command, specifying the SQL SELECT query necessary to return the result table to be mapped into a Virtual DBF.

SQLCommit() establishes a connection on the current work area. After the connection has been successfully established, the application may open a virtual DBF with **USE**.

The connection with the ODBC data source is not terminated if the Virtual DBF is closed with **CLOSE** or **USE**. Once the connection has been established, it remains available until you issue a **SQLDisconnect()**. Establishing a connection demands more processing than most other operations. So keep connections open as much as possible.

Call **SQLDisconnect()** only when you absolutely need to explicitly close a connection.

ODBC connections are automatically disconnected when an application (or web connection) terminates, so it is not mandatory to call **SQLDisconnect()** in many cases.

Notes:

SQLStringConnect() allows the driver to define an arbitrary number of keyword-value pairs as required by an ODBC driver. For example, a certain driver requires a data source name, a user ID and password for the server, and a user ID and password for the DBMS. A program accessing the XYZ Corp data source might prompt the user for IDs and passwords and build the following set of keyword-value pairs, or connection string, to pass to SQLStringConnect():

```
DSN=XYZ Inc;UID=Adam;PWD=Sesame;UIDDBMS=QAdams;PWDDBMS=Shazam;
```

NOTE: This manual is not intended to document the entirety of the ODBC specifications or features of particular ODBC drivers. For driver information, consult the vendor. For further information on ODBC connection string syntax, see **http://msdn.microsoft.com/library/psdk/dasdk/orcd1i2b.htm**

More background information on ODBC can be found at the Microsoft web site (**http://www.microsoft.com/data/odbc/** and **http://www.microsoft.com/data/doc.htm#odbc**) and the web site for the UNIXodbc project. (**http://www.unixodbc.com**)

Example:

```
SET DBTYPE TO ODBC    // Load ODBC Connector to make SQLxxx()
functions available

SELECT 10    // Choosing a work area

conStr:=
"DSN=Inventory;UID=HWang;PWD=mypw;UIDDBMS=JGom;PWDDBMS=Shazam;"
Connnected := SQLStringConnect( ConStr )

if ! Connected
   FatalError("Could Not Connect to ODBC Data Source",
SQLErrorMessage())
end

close
SQLDisconnect()
```

Also see:

PlugSys OXBC Data Access Guide 1

Introduction 9

DBF Extender 11

The PlugSys ODBC Connector 51

PlugSys OXBC Data Access Guide

Introduction

DBF Extender

The PlugSys ODBC Connector

ODBC Connector Language Reference