VBENGINE 1.0
Working Model

Table of Contents

Introduction

What is the VBENGINE?

The VBENGINE product is a Microsoft Windows compatible dynamic link library (VBENGINE.DLL) designed to provide Visual Basic programmers with a sophisticated, yet easy-to-use tool for building database management applications. Using VBENGINE, Visual Basic programmers can build sophisticated multi-user, network compatible database management applications and distribute the VBENGINE.DLL with those applications on an unlimited, royalty-free basis. The VBENGINE.DLL product presents the Visual Basic programmer with a simple, easy-to-use interface to Borland International's Paradox Engine. The Paradox Engine is a complete multi-user, network compatible API written in the C programming language. The VBENGINE product is a simplified object-oriented interface to the Paradox Engine specifically designed for Visual Basic Programmers. VBENGINE (version 1.0) is compatible with Visual Basic 1.0 and 2.0 and runs in Microsoft Windows 3.0 as well as Microsoft Windows 3.1.

The VBENGINE (version 1.0) working model product itself contains a rather extensive subset of the full-featured VBENGINE product (available only to registered users). Even though it is a subset, it has all the essential functions necessary to design full-featured database applications. As a matter of fact, you can design a very sophisticated package in its entirety using only the working model. I included this functionality in the working model so you could better evaluate the product and its capabilities before registering. However, you are not allowed to distribute applications designed around the working model for profit. You must first obtain a registered copy of the VBENGINE product before you are allowed to distribute the VBENGINE with your commercial applications (this includes Shareware products).

The VBENGINE working model has been called a "NagWare" product. There is a pop-up window in the working model that constantly reminds users that the product is for pre-registration evaluation only and cannot be distributed as part of any product. It is displayed throughout portions of the calling programs code. When you register your copy of the VBENGINE, you get a version of the VBENGINE.DLL without the "NagWare" pop-up window suitable for product distribution.

This documentation describes the VBENGINE product. It specifically describes the VBENGINE (version 1.0) working model which is currently being distributed over computer bulletin board services in the United States of America. The VBENGINE (version 1.0) working model is a Shareware product, it is copyrighted and I reserve all rights to it. You may distribute it to others, through any means, as long as you do not charge others for the product itself, or alter the product in any way.

Douglas A. Bebber
March 31, 1993

How to Register

You can obtain a registered copy of the VBENGINE (version 1.0) product for only $49.95. The package includes:

- A full-featured VBENGINE.DLL (access to all functions minus the "NagWare" pop-up).

- VBENGINE users manual complete with example Visual Basic programs and source code.

- VBENGINE Technical Reference Manual describing all VBENGINE functions in detail.

- Unlimited, royalty-free rights to distribute the VBENGINE.DLL with your applications.

- Notice of product updates.

- Free telephone technical support.

To register send check or money order to:

Douglas A. Bebber
1834 37th Street
Rock Island, Illinois 61201
(309) 786-9602

(make notes payable to: Douglas A. Bebber)

Trademarks

PARADOX Engine is a registered trademark of Borland International.

VBENGINE was written in Borland C++ (version 3.0) by Douglas A. Bebber. Address inquiries and bug reports (preferably Dr. Watson along with a listing of the suspected code) to

Douglas A. Bebber

Internet mail address:
bebberd@rr.bhc.edu

U.S. Postal Address:
1834 37th Street
Rock Island, Illinois 61201

Testing

VBENGINE was written and tested on a variety of 286, 386, and 486 PCs. Record and file locking functions were tested and verified on Lantastic and Novell based ethernet LANs as well as in the standard Windows environment between multiple applications.

If your LAN hardware or software differs significantly and VBENGINE does not run properly, I would appreciate a Dr. Watson UAE (General Protection Fault) report sent to my Internet address. Please describe your operating environment in detail and include a listing of your CONFIG.SYS and WIN.INI files.

Note: VBENGINE based Visual Basic programs will not be able to execute properly if **VBENGINE.DLL** and **PXENGWIN.DLL** files are not in directories included in your MSDOS PATH statement.

Note: VBENGINE will only execute in Windows Standard and 386 Enhanced modes.

Compatability and New Releases

The VBENGINE (version 1.0) is compatible with Borland International's Paradox Engine version 2.0. VBENGINE (version 1.5) is now in it's final test phase and will be released soon (expecting documentation to be complete and ready for release in June 1993). The VBENGINE (version 1.5) will contain all the functionality of the version 1.0 product (existing code will be compatible) and will contain multi-media enhancements such as storing Windows bitmap, and .WAV files in databases for Visual Basic's multi-media enhancements. Registered users of the VBENGINE 1.0 product will receive a **free** upgrade to the version 1.5 product when it is released (free upgrade offer expires June 1, 1993).

Database Fundamentals

What is a Database?

For our purposes we will limit this discussion to the world of IBM PC compatible database management systems. Specifically, relational database management systems designed around Borland International's Paradox Engine and the user friendly Visual Basic API (the VBENGINE).

In this context, we can say that a database consists of one or more related files (tables in VBENGINE terminology) that hold information in an orderly, efficient manner. The database tables consist of several rows and columns into which, information is placed. The columns are generally referred to as "Fields" and the rows as "Records".

We will not delve into a lot of theoretical concepts in this section, rather we will present concepts in order to promote a general understanding of database principles. Just enough to give the beginner a kick-start into the world of database programming. (the VBENGINE User's manual covers a little more theory and provides references to more extensive texts.)

To illustrate the principles involved in simple database design we will present a model which we will build on in the VBENGINE Programming Examples section of this manual. To start, we will design a simple database which will hold information concerning the customers of a small business owner. We will design the database from scratch and will detail its structure in this section.

Our small business owner tells us that he would like to maintain a certain set of facts concerning each one of his customers. Specifically, he would like to have the following information concerning each customer in his database:

1.) The customer's Name
2.) The customer's street address
3.) The customer's city of residence
4.) The state that the customer lives in
5.) The customers zip code
6.) The customer's telephone number

This listing of required information is the first essential step in the design of database systems. It is absolutely essential that you compile the needed sets of information that must be maintained. In the world of database programming this step is called "compiling a data dictionary". Over time additional items of information get added to the list. Sometimes, certain items need to be broken down into several smaller parts so a more detailed or "higher resolution" picture can emurge.

The items listed above are the essential pieces of information our small business owner requires for each of his customers. Each piece of information is needed for every customer. If we think of this conceptually, the required information expands in only one direction. Every time we compile the required set of information for a customer our information grows. Its not that we get more information or details concerning the customer, but that we get more customers with the same set of information (Name, Address, etc,).

In a computer-based database system we generally define a set of data that we would like to obtain for each new entry into the database system. This set of information is called the database field set, consisting of a finite set of fields. The six items listed above are our database fields. The entries in our database, each consisting of the same set of fields, are our records. Each and every one of our customers constitute an individual record in our data base. As you can see, our set of required fields should remain more or less constant. But we hope that our customer base continues to expand. In general databases expand in one direction, in our terms, horizontal, record expansion.

Given the above, we can now look at the structure of our database diagramed below (fields vertically, records horizontally):

Name	Address	City	State	Zip	phone
Bob Smith	1111 3rd St.	Denver	CO	11276	323-998-9987
June Day	220 8th Ave	Moline	IL	61265	309-762-1100
Tom Leaky	11 3rd St.	Milan	IL	61201	309-753-0098
		etc.			

It starts out just that simple. A Visual Basic program to maintain just this sort of minimal database would consist of nothing more than a form consisting of Labels and Text boxes designed to aid the user in data entry and a few buttons to facilitate database functions. The DEMO1 example program is a close approximation of this sort of application.

Database Field Types

Each field in a database has a corresponding data type. The available field types in the VBENGINE (version 1.0) release are listed below:

Alphanumeric (A) field type permits the full ASCII character set (except ASCII 0) and is used for entry of string data types. Fields of this type are specified as **Axxx**, where the **xxx** represents the maximum length of the field in characters. For example, if you were to create a field in a table which is intended to hold a maximum of 50 characters you would specify the field as an **A50**.

Number (N) and **currency ($)** field types permit up to 15 significant digits (including the decimal point) in the range of real numbers from $\pm 10^{-307}$ to $\pm 10^{307}$. Number field values which are greater than 15 significant digits are rounded and stored in scientific notation. Currency field values are stored in a default predefined format.

Short Number (S) field types permit values in the range of signed integers. (-32,767 to 32,767).

Date (D) field types permit any vaild dates between January 1, 100 A.D. to December 31, 9999. Date values are stored as long integers which represent the number of days since January 1, A.D.

VBENGINE programming involves handling database field values as strings only! Regardless of the actual data type in the database file. This is mandated by the

VBENGINE data structures (Visual Basic User Defined Types). VBENGINE programmers receive field values from data table files as String values and write database field values to the VBENGINE API as String values regardless of the actual field value type present in the database table file. The VBENGINE automatically performs data type conversions based on the data type of the field in the database table file. This data type conversion process is transparent to the Visual Basic programmer and provides a much simpler interface to database programming.

Indexes and Searching

Database files generally have some sort of indexing scheme in order to facilitate quick searching capabilities. As stated previously, the VBENGINE API (version 1.0) works with Paradox database files. Paradox database files have the capability of supporting multiple indexes. These database indexes are classified into two categories:

Primary indexes
Secondary indexes (maintained and non-maintained)

The Primary index is the default index used in database searches, however, you are able to create and use secondary indexes in your applications. In these database indexes, you specify **key** fields for the index. The **key** fields are the fields you wish to search on or order data by (Primary indexes must have all **key** fields one right after the other with the first key field being the first field in the database) . Primary indexes can have multiple **key** fields.

Just like searching for specific topics in a book, searching a database for specific information is done much more quickly when there is an index present. Indexes order data viewed in a database table. For example, if you have a database table with a single **key** field of type **Number (N)** in the Primary index. Database records viewed through that index will be ordered sequentially in ascending order based on the numeric values present in the **key** field i.e., 0,1,2,3,4,5, etc.

Two of the example programs included in this working model package show how to use indexes. The MAKEDB example program shows how to create an index. The DEMO2 example program shows how to search a database using a primary index (see the VBENGINE Example Programs section in this manual).

Database indexing and searching are some of the more complicated concepts when first learning about database systems. VBENGINE programmer's who may need more details concerning indexes, searching via indexes, general searching techniques, and querying database tables should obtain a copy of the VBENGINE User's Manual.

Multi-User Environments

The VBENGINE API can be used in Local Area Network environments consisting of multiple users sharing the same database. The VBENGINE working model comes with file and record locking facilities. VBENGINE API functions specific to network file sharing environments are **LockRecord, UnlockRecord, LockFile, UnlockFile, GetUserName**, etc.

For a complete description of the concepts intoduced in this section and information on other VBENGINE database related information please see the VBENGINE User's Manual. For specific information on the Paradox database file structure and concepts relevant specifically to the Pardox Engine see the Paradox Engine User's Guide available from Borland International.

VBENGINE Data Structures

The VBENGINE data structures are Visual Basic User Defined Types. They are defined in the **VBENGINE.TXT** file. The **VBENGINE.TXT** file's contents must be

ported into a Visual Basic program's Global Module in order to use the VBENGINE for database programming. The **VBENGINE.BAS** file is a module which contains the **VBENGINE.TXT** file's contents.

Understanding the VBENGINE **DataTable** User Defined Type is the key to success in VBENGINE programming. The data structures (User Defined Types) discussed in this document are the bare minimum VBENGINE data types. More complex data structures can be built upon these core data structures to provide more sophisticated structures for large complex programming requirements. Generally the more sophisticated structures require Microsoft Visual Basic 2.0 as your programming platform (since it provides for arrays in user defined types). Extending VBENGINE's data structures is covered in the VBENGINE Technical Reference Manual.

During the following discussion conerning the data structures, the VBENGINE User Defined Types are referred to as objects. The VBENGINE architecture is very similar to the structure of a similar product designed by the author of the VBENGINE (a database engine class library) designed for C++ programmers in which databases are manipulated via DataTable objects.

The VBENGINE User Defined Types are described in this section, for details on how to use these structures in your Visual Basic programs see the **VBENGINE Sample Programs** section.

The DataTable Object

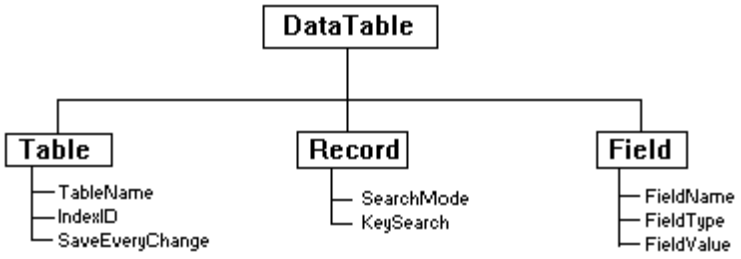


Figure 1.0 VBENGINE DataTable object.

VBENGINE programmers work with databases through **DataTable** objects. The **DataTable** object is a Visual Basic *User Defined Type* which conceptually simplifies database programming.

As you can see in Figure 1.0, the **DataTable** object is made up of three other objects:

- Table object
- Record object
- Field object

Each of these embedded objects consist of a few data members:

Table:	**Record:**	**Field:**
TableName	***SearchMode***	***FieldName***
IndexID	***KeySearch***	
FieldType		
SaveEveryChange		***FieldValue***

In this section we will examine the **DataTable** object in detail. In the following section **VBENGINE Example Programs** we will illustrate how to use the **DataTable** object to manipulate databases using the Visual Basic programming language.

Table Object

The **DataTable** object holds database table specific information. It contains information concerning the table only. The **Table** object has three data members:

TableName is an ASCII string with a length of 255 characters. This string holds the name of a database table, including any MSDOS PATH specifier. Database file names placed in this data member must not include a file extension.

IndexID is an integer data member which holds the identification of the index to be used with the database table (specified in the **TableName** data member).

SaveEveryChange is an integer data member which determines how database changes are saved to disk files. Database changes may be directly written or buffered.

Record Object

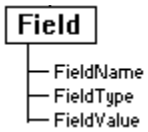
The **Record** object holds database record specific information. The majority of the record structures are internal to the database engine. The **Record** object has two data members:

SearchMode is an integer data member which specifies the search mode used in database searches. It's scope is in relation to it's parent **DataTable** variable only (it does not affect other **DataTable** variables, every **DataTable** variable has it's own table search mode data member). The **SearchMode** data member can have any one of three valid values:

SEARCHFIRST
SEARCHNEXT
CLOSESTRECORD

KeySearch is an integer data member which specifies what portion of the databases primary index to use for index based searches.

Field Object



The **Field** object holds database field specific information. The **Field** object has three data members:

FieldName is an ASCII string with a length of 25 characters. This string holds the name of the target database field.

FieldType is an ASCII string with a length of 30 characters. This string holds the data type of the target database field.

FieldValue is an ASCII string with a length of 255 characters. This string holds the value of the target database field.

To use **DataTable** objects and the library of functions that operate on **DataTable** objects you must include the **VBENGINE.TXT** file in your Visual Basic program's GLOBAL MODULE. The **DataTable** objects are defined in that file. Some additional data members for the **Table, Record,** and **Field** objects are defined in that file, however, those that are not listed above are for VBENGINE's internal use only! You must never alter these data members because they are Visual Basic links to the data base engine environment. Modifying these "Handle" data members will cause unexpected results!

All database programming capabilities (with one exception), are provided to the Visual Basic programmer through the use of **DataTable** objects and the functions that operate on those objects. The one exception is the creation of new database tables. To create a new database table you must use the **NewTable** object (**NewTable** Type). The **NewTable** type is defined in the VBENGINE.BAS file. The **NewTable** Type and the **CreateTable** function are described in the **Function Reference**. An example covering table creation is present in the next section.

VBENGINE Example Programs

In this section detailed examples of how to use the VBENGINE in the Visual Basic programming language will be presented. Details concerning how to use the VBENGINE API to read and write data between Visual Basic programs and database files are covered in detail. Several example programs have been sent along as part of the working model distribution file set. These example programs are discussed here in detail.

This section is a subset of the same section in the registration copy of the VBENGINE User's Manual. There is not as much information concerning complex searching, relational models, and querying present here. There is however, sufficient information to get one started in VBENGINE programming. Specific examples of how to search with an index and how to search on a field are presented here. Also an example is given on how to fill in Visual Basic List and Combo boxes with data from a database table using the VBENGINE. For extensive information concerning database indexing and searching please see the registration copy of the VBENGINE User's Manual.

Installation Note: The example programs presented in this section (including the Visual Basic source code programs included with the working model) expect the files **CUSTOMER.DB** and **CUSTOMER.PX** files to be in the C:\ directory. If you wish to change this location do so in the example programs source code.

DEMO1

This is the first example program. It is very simple and illustrates a few basic VBENGINE programming concepts. It can be found in the working model distribution file set. The files DEMO1.MAK, VBENGINE.BAS, and DEMO1.FRM constitute the file set for the DEMO1 example program. The source code is commented. (The database table and index used in the DEMO1 example program were created using the MAKEDB.MAK project also included in the working model file distribution and discussed at the end of this section).

The DEMO1 example program is a very simple illustration of how easily a database application can be generated in Visual Basic using VBENGINE. The DEMO1 example program consists of one database table "C:\CUSTOMER.DB" with an index file "C:\CUSTOMER.PX" . The structure of the C:\CUSTOMER.DB database is shown below.

Field	Type
Name	A50
Address	A50
City	A30
State	A2
Zip	A10
Phone	A14

The DEMO1.FRM form was designed to be a window into the database. Using this form, users can view the customer data in the database on a record-by-record basis. There is a field on the form for every field in the database. Six Text controls are used to hold database field values, and six Label controls are used to label those fields for the users benefit. There are seven push button controls on the form for database manipulation:

- **Top** moves to the first record in the database.
- **Bottom** moves to the last record in the database
- **Previous** moves to the previous record in the database
- **Next** moves to the next record in the database
- **Update** updates the current record in the database
- **Insert** inserts the form data as a record in the database
- **Delete** deletes the current record from the database

There are two utility push button controls on the form:

- **Clear** clears all information from the form (blank form)
- **Quit** terminates the demo program

We will start the description of this example application in the DEMO1 form's general declarations section. Here a variable of type **DataTable** is declared as:

```
Dim Customer As DataTable
```

This line of code creates a **DataTable** object, which we will refer to by name as "Customer", that will allow us to manipulate the database through the VBENGINE API.

We next see the following code in the Forms load procedure:

```
'Initialize VBENGINE so that database capabilities are enabled.
'Do this by calling OpenEngine with a string representing the program's
        'name.

result = OpenEngine("Visual Basic - VBENGINE DEMO1")

'Now put the database table file name (C:\CUSTOMER.DB) in the
'Customer DataTable object:

Customer.Table.TableName = "C:\CUSTOMER"

'We will use the tables master index
Customer.Table.IndexID = MASTERINDEX

'We will buffer data changes
Customer.Table.SaveEveryChange = FALSE

'Now open the table
result = OpenTable(Customer)

'Now read in the data from the first record and place it in our form.
FillForm
```

This is only six lines of code! In this six lines of code we have:

- Initialized the database engine environment.

- Configured our Customer **DataTable**'s **Table** object to specify what database file we will use,

what index(s) we will use, and that we intend to buffer all data changes to the disk.

- Opened the database file

- Called a Visual Basic subroutine **FillForm** which will read the data from a record and place that data onto our form.

Now let's examine the **FillForm** subroutine to see what it takes to actually read data from the database and place it in our Visual Basic form. The **FillForm** subroutine is a subroutine present in the Form's general section. Here it is in it's entirety:

```
Sub FillForm ()

        Dim result As Integer           'Used to detect errors.

        'Get the record from the table
         result = GetRecord(Customer)

        'Now lets get the customers name and put it in our form.

        'Specify what database field we are interested in by placing the name of the field i~ our
        'Customer DataTable object:
        Customer.Field.FieldName = "Name"

        'Read in the value.
        result = GetField(Customer)

        'Place it in the form.
        Text1.Text = Customer.Field.FieldValue

        'Now do the same thing for every field in our form ...

        Customer.Field.FieldName = "Address"
        result = GetField(Customer)
        Text2.Text = Customer.Field.FieldValue

        Customer.Field.FieldName = "City"
        result = GetField(Customer)
        Text3.Text = Customer.Field.FieldValue

        Customer.Field.FieldName = "State"
        result = GetField(Customer)
        Text4.Text = Customer.Field.FieldValue

        Customer.Field.FieldName = "Zip"
        result = GetField(Customer)
        Text5.Text = Customer.Field.FieldValue

        Customer.Field.FieldName = "Phone"
        result = GetField(Customer)
        Text6.Text = Customer.Field.FieldValue
End Sub
```

Notice that the **FillForm** subroutine is a general purpose subroutine. It simply reads in a record's worth of data and displays that data on our form. It does not in any way position the current record in the database. It reads in the current record and displays the field data on the form. The point here is that we will use other routines to move around in the database and once we position to the desired record we will call **FillForm** to display the information.

At this point in time, our DEMO1 program has opened up our database engine environment, opened up our Customer database, read in the first record and displayed the customer data on our form. The program is now waiting for the user to do something. Let's look at the top row of push button controls on our form:

Top **Bottom** **Previous** **Next**

These push button controls are for movement in the database table. They let our DEMO1 user navigate through our database. Let's take a look at the code attatched to each of these push button controls:

```
Sub TopButton_Click ()

        Dim result As Integer           'For error detection

        'Move to the first record in the table.
        result = FirstRecord(Customer)

        'Now fill in the form
        FillForm
End Sub

Sub BottomButton_Click ()

        Dim result As Integer           'For error detection

        'Move to the first record in the table.
        result = LastRecord(Customer)

        'Now fill in the form
        FillForm
End Sub

Sub PreviousButton_Click ()

        Dim result As Integer           'For error detection

        'Move to the first record in the table.
        result = PreviousRecord(Customer)

        'Now fill in the form
        FillForm
End Sub

Sub NextButton_Click ()

        Dim result As Integer           'For error detection
```

```
        'Move to the first record in the table.
        result = NextRecord(Customer)

        'Now fill in the form
        FillForm
End Sub
```

Pretty simple code! In essence, each one of these positional controls simply calls a single VBENGINE function call to reposition the database's current record pointer. Then calls the **FillForm** subroutine to read the data in and display it on our form.

Now let's take a look at the **Delete** push button's code:

```
        Sub DeleteButton_Click ()

                Dim result As Integer

                'Delete the current record from the data table.
                result = DeleteRecord(Customer)

                'Now fill in the form.
                FillForm

        End Sub
```

It doesn't take a lot of code to delete a record from the database. When this push button is clicked by the user, the record is deleted from the database by calling the **DeleteRecord** function. When the record is deleted from the database the database engine automatically moves the database record pointer to the next available record so all we have to do is to call our **FillForm** subroutine to display the current database record.

Now we only have two more database related push button controls to look at **Update** and **Insert**.

```
Sub UpdateButton_Click ()
        Update
End Sub

Sub InsertButton_Click ()
Insert
End Sub
```

Thats it for the buttons themselves, now let's look at the two subroutines **Update** and **Insert** each present in the Form's general section:

```
Sub Update ()
        Dim result As Integer           'For error detection

        'Here we are transfering information from our form to the table.
        'We must first associate a form value with specific fields in our table.
```

```
        'We will make such associations by first specifying the database field of
        'interest. Then take the data from the corresponding form field and then
        'put the field into the current record.

        'Specify the field in the data table.
        Customer.Field.FieldName = "Name"

        'Place corresponding form data into the database structures FieldValue member.
        Customer.Field.FieldValue = Text1.Text

        'Now put the field structure into the current record.
        result = PutField(Customer)


        'Now repeat the process for all form fields.

        Customer.Field.FieldName = "Address"
        Customer.Field.FieldValue = Text2.Text
        result = PutField(Customer)

        'Customer.Field.FieldName = "City"
        Customer.Field.FieldValue = Text3.Text
        result = PutField(Customer)


        Customer.Field.FieldName = "State"
        Customer.Field.FieldValue = Text4.Text
        result = PutField(Customer)

        Customer.Field.FieldName = "Zip"
        Customer.Field.FieldValue = Text5.Text
        result = PutField(Customer)

        Customer.Field.FieldName = "Phone"
        Customer.Field.FieldValue = Text6.Text
        result = PutField(Customer)

        'All fields have been placed
        'Here we UPDATE the current record in the table.
        result = UpdateRecord(Customer)
End Sub



Sub Insert ()
Dim result As Integer

        'Here we do the same process found in the Update subroutine.
        '(Transfer data from our form fields to the current record)

        Customer.Field.FieldName = "Name"
        Customer.Field.FieldValue = Text1.Text
        result = PutField(Customer)
```

```
Customer.Field.FieldName = "Address"
Customer.Field.FieldValue = Text2.Text
result = PutField(Customer)

Customer.Field.FieldName = "City"
Customer.Field.FieldValue = Text3.Text
result = PutField(Customer)


Customer.Field.FieldName = "State"
Customer.Field.FieldValue = Text4.Text
result = PutField(Customer)

Customer.Field.FieldName = "Zip"
Customer.Field.FieldValue = Text5.Text
result = PutField(Customer)

Customer.Field.FieldName = "Phone"
Customer.Field.FieldValue = Text6.Text
result = PutField(Customer)


'Here we INSERT the current record in the table.
result = InsertRecord(Customer)

End Sub
```

Not to difficult is it! Well that's about it, a few more minor details t cover, like the **Clear** push button. All it does is clear the text values in the Form's Text controls. No VBENGINE functions are associated with the **Clear** button. However, an important concept is associated with the **Quit** button. Remember way back at the Form's Load subroutine, when we set up the **DataTable** object's Table.SaveEveryChange data member to buffer database changes too disk? Well before we quit the DEMO1 example program we want to make sure that any changes are indeed saved to the database disk file. This can be done at any time manually by calling the VBENGINE's **FlushBuffers** function. But in DEMO1, we simply rely on the **CloseTable** function to save all changes before closing the table. We really don't even need to call **CloseTable** (it's a good practice) because directly after calling **CloseTable** we call **CloseEngine** which cleans-up the database engine environment before shutting-down. Part of the **CloseEngine**'s clean-up procedure is to FLUSH all open buffers and to close and release table handles.

Well that's about it for the DEMO1 example application. Let's take a look at another example application MAKEDB, which actually created the table and index for the DEMO1 applications Customer database.

MAKEDB

The MAKEDB example program is included in the VBENGINE working model distribution file set. It is an example program which illustrates the steps and procedures necessary to create database files and indexes. The CUSTOMER.DB and CUSTOMER.PX files used in the example

programs DEMO1 - DEMO4 were created with the MAKEDB program. The MAKEDB example program contains a Visual Basic Form that contains two push button controls - **Create Table** and **Create Index**.

We will begin the examination of the MAKEDB example program by looking at what it takes to create a database table file. Let's take a look at the Visual Basic code attatched to the **Create Table** push button:

```
Sub CreateTable_Click ()
'We will declare a variable of type NewTable to create our
'database table

Dim Customer As NewTable
Dim status As Integer       'Used to detect errors.


'Our database will consist of 6 fields and have
'the following structure:
'
'   Field          Type
'---------------------------------------
'   Name           A50
'   Address        A50
'   City           A30
'   State          A2
'   Zip            A10
'   Phone          A14


'Lets fill in the details of the tables structure here
Customer.TableName = "C:\Customer"      'Specify the database file name
Customer.NFields = 6                    'Six fields:Name,Address,City,State,Zip, and Phone

Customer.FieldNames = "Name,Address,City,State,Zip,Phone"          'Field names must be
                                                                   'separated by
commas

Customer.FieldTypes = "A50,A50,A30,A2,A10,A14"                     'Field types must be
                                                                   'separated by
commas

'Ok, lets go ahead and create the database table using
'VBENGINE function calls

status = OpenEngine("Customer Creation Example")    'Initialize database engine

If (status <> 0) Then           'If an error terminate the program

  MsgBox "Database Engine environment could not be initialized!"
  End

End If

status = CreateTable(Customer)      'Create the database table
```

```
If (status <> 0) Then          'If an error terminate the program

  MsgBox "Customer database could not be created!"
  End

End If

MsgBox "Customer database was successfully created!"    'Tell the user everything is A OK!

status = CloseEngine()          'Now shut-down the database engine environment

If (status <> 0) Then          'If an error terminate the program

  MsgBox "Database engine could not be shut-down!"
  End

End If

End Sub
```

All databases must by created by using variables of type **NewTable.** Of course any external utility or program which creates Paradox Tables can be used. But if you are going to create your own tables using the VBENGINE API you must use the **NewTable** type. For a detailed information on the **NewTable** type and how to create database table files see the **CreateTable** function description in the function reference. The above code is commented to explain the steps necessary to create tables, For more detailed information see the **CreateTable** function in the Function Reference section of this manual.

Now let's look at the code attatched to the **CreateIndex** push button:

```
Sub CreateIndex_Click ()
'Here we create a PRIMARY index for the Customer table
'on the Name field. This will be our only key field
'in the Primary index.
'no two customers will be able to have the exact same name
'and the table will be sorted alphabetically

Dim status As Integer   'for error handling

status = OpenEngine("Customer Creation Example")    'Initialize database engine

If (status <> 0) Then          'If an error terminate the program

  MsgBox "Database Engine environment could not be initialized!"
  End

End If

'We will use the AddKey function to create the index
'The table name is "C:\CUSTOMER"
'We want only one key field.
'It is Name, the first field in the table.
```

```
status = AddKey("C:\CUSTOMER", 1, 1, PRIMARY)     'see the Addkey function in the function
                                                   'reference section of the
VBENGINE                                           'working model manual

If (status <> 0) Then    'If index creation failed

  MsgBox "Failed to create Customer index!"
  End

End If

MsgBox "Customer index successfully created!"


status = CloseEngine()          'Now shut-down the database engine environment

If (status <> 0) Then           'If an error terminate the program

  MsgBox "Database engine could not be shut-down!"
  End

End If

End Sub
```

Here again, the code is commented to explain the steps necessary for creating database indexes. For more information see the **AddKey** and the **RemoveKey** function descriptions in the Function reference section of this manual.

Database Searching Techniques

The DEMO2 Example Program (Searching with an Index)

The DEMO2 example program shows how you can search a database on an index for a specific value. The DEMO2 program is a modified version of the DEMO1 example program. DEMO2 is DEMO1 with one extra subroutine **Text1 LostFocus()**. The Text1 field is a window into our Customer database's Name field. The DEMO2 program is structured to accept keyboard input from the user and when the user types in a customer's name and leaves the Text1 control

(LostFocus), the Customer database is searched (on the PRIMARY index) for the name typed in by the user. If the user typed name is not found, the remaning fields are cleared and we expect to receive information for a new customer. If the user typed name is found in the database, the remaing fields on the form are filled in with that customer's information.

Let's look at the code in the **Text1 LostFocus** subroutine:

```
Sub Text1_LostFocus ()

  Dim status As Integer              'For error handling
  Dim NewName As String              'Used to store user typed name, when search fails


'Set up SearchKey criteria:

  Customer.Field.FieldName = "Name"          'We will search on the Name key field
  Customer.Field.FieldValue = Text1.Text     'For this customer name
  Customer.Table.IndexID = MASTERINDEX       'Using  our Primary index
  Customer.Record.SearchMode = SEARCHFIRST   'Find the first record meeting the criteria
  Customer.Record.KeySearch = 1

  status = PutField(Customer)                'Submit the field for the search

  status = SearchKey(Customer)               'Start the search

  If (status <> 0) Then                      'Search failed to find the customer name
     NewName = Text1.Text                    'Store the new customer name in NewName
     ClearButton_Click                 'Clear the form
     Text1.Text = NewName                    'Put the new name back on the form
     Exit Sub
  End If

  If (status = 0) Then                       'Search was successful
     FillForm                                'Get customer info and put it in the Form
  End If

End Sub
```

Note: Searching techniques using the VBENGINE are discussed in greater detail in the VBENGINE User's Manual.

The DEMO3 Example Program (Searching on a Specific Field)

The DEMO3 example program shows how you can search a database on an index for a specific value. The DEMO3 program is a modified version of the DEMO1 example program. DEMO3 is DEMO1 with one extra subroutine **Text1 LostFocus()**. The Text1 field is a window into our Customer database's Name field. The DEMO3 program is structured to accept keyboard input from the user and when the user types in a customer's name and leaves the Text1 control (LostFocus), the Customer database is searched (on the NAME field) for the name typed in by the user. If the user typed name is not found, the remaning fields are cleared and we expect to receive information for a new customer. If the user typed name is found in the database, the remaing fields on the form are filled in with that customer's information.

Let's look at the code in the **Text1 LostFocus** subroutine:

```
Sub Text1_LostFocus ()

  Dim status As Integer                 'For error detection and correction
  Dim NewName As String                 'Used to store new customer's name


  Customer.Field.FieldName = "Name"                 'Field we wish to search on
  Customer.Field.FieldValue = Text1.Text            'Value we wish to search for
  Customer.Record.SearchMode = SEARCHFIRST          'Find the first matching record


  status = PutField(Customer)           'Submit the search criteria to the database engine

  status = SearchField(Customer)        'Start the search

  If (status <> 0) Then                 'If search failed
     NewName = Text1.Text               'Put new name in NewName variable
     ClearButton_Click          'Clear the form
     Text1.Text = NewName               'Put the new name back on the Form
     Exit Sub                           'Return
  End If

  If (status = 0) Then                  'Search was successfull
     FillForm                           'Put the customer's info on the Form
  End If

End Sub
```

The last example program in the VBENGINE (version 1.0) working model distribution file set is the DEMO4 example program. The DEMO4 program is pretty much the same as the DEMO3 program but it includes an example on how to read data from a database table and place that data into a combo box for database sourced pick lists. The DEMO4 program will not be examined in detail here, the source code is well commented. Look for the new subroutines **FillCustomerCombo** and **EmptyCustomerCombo** in the Form's general section. For more VBENGINE example programs and more detailed information on VBENGINE programming place your order for a registered copy today. From time-to-time new versions of the VBENGINE working model will be released. New example programs covering different aspects of VBENGINE programming will be distributed therein.

VBENGINE Function Reference

AddKey

Description

Creates a primary or secondary index for a table.

Syntax

AddKey(*TableName* **As** String, *NFields* **As** Integer, *FieldHandle* **As** Integer, *Mode* **As** Integer)

Remarks

This function provides for the creation of key indexes (Primary and Secondary). The function accepts four arguments described as follows:

TableName
An ASCII string which holds the name of the data table for which an index is to be made. This variable should contain the table name, including any MSDOS PATH specifier. Note: do not include a file extension (.DB).

NFields
This argument is of type integer and represents the number of fields you wish to make keyed fields for a PRIMARY index and represents the first NFields contiguous fields. For SECONDARY indexes, this argument is always set to 1.

FieldHandle
This is the field position in the table for the key field. For PRIMARY indexes this argument always has a value of 1. If a SECONDARY index is being created, this argument should be set to the number of the field's position in the table, i.e., the **fieldhandle**.

Mode
This argument is used to specify what type of index is being created
PRIMARY,SECONDARY,INCSECONDARY.

For a discussion on the types of indexes and keyed fields available see the topic of **indexing** in the **Database Fundamentals** chapter in this manual. Upon a successfull return the **AddKey** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

Example(s)

This example creates a PRIMARY Index:

```
Sub NewIndex1_Click ()

        DIM status As Integer
        DIM Table As String
        DIM NumberOfFields As Integer
        DIM FldHandle As Integer
        DIM IndexMode As Integer

Table = "C:\Customer"
NumberOfFields = 1              ' Customer table will have only a single key field. The first field in the
                                ' table, the Customer Name field.
FldHandle = 1                   ' Primary index
IndexMode = PRIMARY

status = AddKey(Table, NumberOfFields, FldHandle, IndexMode)

End Sub
```

This example creates a SECONDARY index on the third field in the CUSTOMER table (City):

```
Sub NewIndex2_Click ()

        DIM status As Integer
        DIM Table As String
        DIM NumberOfFields As Integer
        DIM FldHandle As Integer
        DIM IndexMode As Integer

Table = "C:\Customer"
NumberOfFields = 1
FldHandle = 3                   ' Secondary index for the City field.
IndexMode = INCSECONDARY

status = AddKey(Table, NumberOfFields, FldHandle, IndexMode)
```

End Sub

AddPassword

Description

This function enters a password into the system.

Syntax

AddPassword(*Password* **As** String)

Remarks

If database engine resources have been **protected** by a password, users must provide the necessary password to gain access to those resources. The **AddPassword** function call submits a password on your applications behalf. Any database engine resources (Tables) which require the password are **automatically** available for routine manipulation once that password has been submitted with the **AddPassword** function call.

This function call requires a single argument of type String which is the ASCII representation of the password. A sucessfull function call will return an integer value of zero (0), any error will return a non-zero integer value.

See Also

RemovePassword

Example

Sub Password_Click ()

DIM result As Integer

status = AddPassword("Bryan sent me")

End Sub

AppendRecord

Description

This function appends a record to a database table.

Syntax

AppendRecord(*Table* **As** DataTable)

Remarks

This function writes (appends) the record specified in the DataTable argument variable to the database file. If the database file is indexed the **AppendRecord** function works similar to the **InsertRecord** function call

and the record is inserted in the database file at a place specified by the index. If the database file is not indexed the appended record is added to the end of the database file. In both cases the newly appended record becomes the current record. Upon a successfull return the **AppendRecord** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

See Also

InsertRecord, UpdateRecord, DeleteRecord.

Example

```
Sub AddRecord_Click ()

' A variable is dimensioned elsewhere in the program as:
' DIM Customer As DataTable

DIM status As Integer

status = AppendRecord(Customer)  'Append record to table

End Sub
```

ClearRecord

Description

This function clears out the current record for the specified database table.

Syntax

ClearRecord(*Table* **As** DataTable)

Remarks

This function clears the database engine's internal record information for the DataTable argument variable. Specifically all internal information for the **DRecord** data structure is erased. It is a convienient way to clear all the field values for a specific record and is functionally equivalent to calling the **PutBlank** function for each and every field.Upon a successfull return the **ClearRecord** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

See Also **PutBlank**

Example

```
Sub ClearRecord_Click ()

' a variable, declared elsewhere in this program was done like so:
' Dim Customer As DataTable

Dim status As Integer

status = ClearRecord(Customer)

End Sub
```

CloseEngine

Description

This function shuts-down the database engine environment.

Syntax

CloseEngine()

Remarks

When a Visual Basic Program is finished with the database engine and no further database processing is required the program should make a **CloseEngine** function call to clean-up and free memory allocated by the database engine environment. If a database was using table buffering (**.Table.SaveEveryChange = False)** then all buffered data is saved to disk, all open tables closed, etc. before the database engine's environment is shut-down. All programs should call **CloseEngine** when they are finished with database processing. Upon a successfull return the **CloseEngine** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

See Also

OpenEngine

Example

...
...
...
Dim status As Integer

status = CloseEngine()
...
...
...

CloseTable

Description

This function closes a previously opened database table.

Syntax

CloseTable(*Table* **As** DataTable)

Remarks

This function ensures that all buffered data is saved to disk and all memory allocated for the open table is released when the table is properly closed. When a Visual Basic Program is finished with a database table it should make a **CloseTable** call to insure that the table is properly closed and that no data is lost. Upon a successfull return the **CloseTable** function returns an integer value of zero (0). In the event of an error, a

non-zero integer error value is returned.

See Also

OpenTable

Example

```
Sub CloseCustomer ()

'A DataTable variable was declared elsewhere in the table as:
'Dim Customer As DataTable

Dim status As Integer

status = CloseTable(Customer)

End Sub
```

CreateTable

Description

This function is used to create a new database table file.

Syntax

CreateTable(*Table* **As** NewTable)

Remarks

The **CreateTable** function accepts as an argument, a variable of Type **NewTable** (see VBENGINE Data Structures).
You define the structure of the new database table through the **NewTable** data structure:

```
Type NewTable

        TableName As String * 255
        NFields As Integer
        FieldNames As String * 6629
        FieldTypes As String * 1529
End Type
```

You place the name of your new table in the **TableName** member. This member consists of a String type which contains up to 255 characters. Here you place the name of your table, including any MSDOS PATH, but do not include an MSDOS file extension.

You place the total number of fields in your new table in the **NFields** member. This member is an integer and can have a maximum value of 255.

You place the names of your table's fields in the **FieldNames** member. This member is a String type which contains up to 6629 characters. Your table's field names should be placed in this member in the same order you expect them to be found in your table. Each field name is separated by a comma (,). The last field name should **not** be terminated with a comma. Field names can themselves be a maximum of 25 characters in length.

You place the field types of your above defined fields in the **FieldTypes** member. This member is a String type which contains up to 1529 characters. Your field types should be separated from one another by a comma(,). The list of comma separated field types in the **FieldTypes** member string should follow a one-to-one correspondence with the comma separated field names residing in the **FieldNames** member. A field type can be a maximum of five characters and must consist of one of the following field types:

Field Type	Data Type
N	**Numeric**
S	**Short number**

$	**Currency**
Annn	**Alphanumeric**
D	**Date**

Upon a successfull return the **CreateTable** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

Note: The total numer of bytes per record should not exceed 4000. If the table is to be keyed for a primary index the limit is reduced down to 1350 bytes.

See Also

AddKey

Example

```
Sub NewCustomerTable ()

' The new CUSTOMER table will hold data for our customers and will have the following format:
'           Field Name                          Data Type
'-------------------------------------------------------------------------------------------------------
'           Name                                A50
'           Address                             A50
'           City                                A30
'           State                               A2
'           Zip                                 A10
'           Phone                               A14
'           Fax                                 A14
'           Cust As OF                          D
'

Dim Customer As NewTable            'variable of type NewTable for defining the new database table.
Dim status As Integer       ' status will hold the result of function calls (error trapping)

Customer.TableName = "C:\CUSTOMER"
Customer.NFields = 8
Customer.FieldNames = "Name,Address,City,State,Zip,Phone,Fax,Cust As OF"
Customer.FieldTypes = "A50,A50,A30,A2,A10,A14,A14,D"

status = CreateTable(Customer)

End Sub
```

DecryptTable

Description

This function decrypts a previously encrypted table.

Syntax

DecryptTable(*Table* **As** DataTable)

Remarks

Database table files can be encrypted for security purposes. Once a table is encrypted using the **EncryptTable** function call, encryption can only be removed via a **DecryptTable** Function call. A table is encrypted through a password (see the **EncryptTable** function description), if you do not have password rights (if you don't know the password) you can not successfully call **DecryptTable.** Upon a successfull return the **DecryptTable** function returns an integer value of zero (0). In the event of an error, a non-zero

integer error value is returned.

See Also

EncryptTable

Example

```
Sub DecryptCustomer ()
'A variable was declared elsewhere in the program as:
'Dim Customer As DataTable

Dim status As Integer

status = DecryptTable(Customer)

End Sub
```

DeleteRecord

Description

This function deletes the current record from the database table.

Syntax

DeleteRecord(*Table* **As** DataTable)

Remarks

This function deletes the current record in the database table. The database table and the current record are contained inside the passed DataTable argument. Upon a successfull return the **DeleteRecord** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

Example

```
...
...
' Customer is declared elsewhere in the program as:
'Dim Customer As DataTable

Dim status As Integer

status = DeleteRecord(Customer)
...
...
```

DeleteTable

Description

This function deletes a table and its associated family of objects.

Syntax

DeleteTable(*Table* **As** String)

Remarks

When this function is called to delete a table, it will, if successfull, delete the named table, indexes, forms,

reports, graphs, image settings and validity checks (see **Tables and family objects** in the **Database Fundamentals** section of this manual). Upon a successfull return the **DeleteTable** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

Example

```
...
...
...
Dim status As Integer

status = DeleteTable("C:\CUSTOMER")
...
...
...
```

EmptyTable

Description

This function removes all records from the specified table.

Syntax

EmptyTable(*Table* **As** String)

Remarks

When this function is called all records (information) present in the table is removed or erased leaving nothing but an empty database table. Upon a successfull return the **EmptyTable** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

Example

```
...
...
...
Dim status As Integer

status = EmptyTable("C:\CUSTOMER")
...
...
...
```

EncryptTable

Description

This function encrypts a database table.

Syntax

EncryptTable(*Table* **As** DataTable, *Password* **As** String)

Remarks

This function call, when successfull, encrypts the specified table. Once encrypted, the table can only be accessed by users with access to the password. Tables are encrypted for purposes of security. Upon a successfull return the **EncryptTable** function returns an integer value of zero (0). In the event of an error,

a non-zero integer error value is returned.

See Also

DecryptTable and **Passwords and Security** in the **Database Fundamentals** section of this manual.

Example

```
...
...
...
'A variable was declared elsewhere in the program as:
'Dim Customer As DataTable

Dim status As Integer

status = EncryptTable(Customer,"OMEGA")
...
...
...
```

FirstRecord

Description

This function positions the current record on the first record in the database table.

Syntax

FirstRecord(*Table* **As** DataTable)

Remarks

This function, if successfull, moves to the first record in the database table and makes that record the current record. The database table and the current record for that table are passed in the DataTable argument variable. Upon a successfull return the **FirstRecord** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

See Also

LastRecord, NextRecord and PreviousRecord.

Example

```
Sub TableTop_Click ()

'A variable was declared elsewhere in the program as:
'Dim Customer As DataTable
Dim status As Integer

status = FirstRecord(Customer)

End Sub
```

FlushBuffers

Description

This function writes all buffered data to database table files.

Syntax

FlushBuffers()

Remarks

This function is a system level database engine process. It writes all buffered data to disk. If a DataTable variable is set-up to buffer data (**SaveEveryChange** = FALSE) then database changes are not written immediately to disk but are instead buffered generally to increase performance. Even if set-up for buffering, you can force buffered data to be written to disk by calling **FlushBuffers.** Upon a successfull return the **FlushBuffers** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

See Also

SaveEveryChange in the **VBENGINE Data Structure Definition** section of this manual.

Example

```
...
status% = FlushBuffers()
...
```

GetField

Description

This function reads the value of a specified field from the current record of a database table.

Syntax

GetField(*Table* **As** DataTable)

Remarks

This function call reads the value of the field specified by **Table.Field.FieldName** and places that field's value in **Table.Field.FieldValue**. The field value read is that of the current record in the database table of the Table (DataTable) argument passed to the function. All field values placed in **Table.Field.FieldValue** are of type string regardless of the actual data type stored in the table itself. Upon a successfull return the **GetField** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned. The steps required to read a particular field value from a database table and place that information in a Visual Basic Text Box are shown below:

Example

```
Sub GetAField ()

Dim Customer As DataTable                    'Create a DataTable variable to manipulate a
database file.
Dim status As Integer                        'Variable for error handling..

'Set-up the Customer DataTable variable to access the database file.
Customer.Table.TableName = "C:\Customer" 'We will use the C:\CUSTOMER.DB database file.
Customer.Table.IndexID = MASTERINDEX     'We will view the database through its Primary index.
Customer.Table.SaveEveryChange = FALSE   'We will buffer any changes to disk.

'OK, the DataTable variable is set up for Table specific access.
'Now lets initialize the database engine. We will assume that all will go well and will not complicate this
example
'with specific error handling code.

status = OpenEngine("Visual Basic Program")

'Ok the database engine is now up and running, now lets open up our database table:

status = OpenTable(Customer)

'Ok, our table is open, lets get the first record in the Customer table:

status = GetRecord(Customer)

'Now we want to get the NAME of our customer in record#1 and place it in our Text1 control:

Customer.Field.FieldName = "Name"            'Customer name is stored in a field called Name.

'Now get the customer's name:

status = GetField(Customer)

'Ok the customer's name is now in Customer.Field.FieldValue.
'Let's put it in the TextBox

Text1.Text = Customer.Field.FieldValue
'Ok, a job well done. Lets stop, we will need to close our table, and the database engine before we quit:

status = CloseTable(Customer)                'Close the database.
status = CloseEngine()                       'Close the database engine

End Sub
```

GetFieldType

Description

This function returns the data type for a database field.

Syntax

GetFieldType(*Table* **As** DataTable)

Remarks

This function call returns the data type of the field specified in **Table.Field.FieldName**. You use this function when you wish to determine the actual data type of the field as it is stored in the database table. The possible data types returned are as follows:

Field Type	Data Type
N	**Numeric**

S	**Short number**
$	**Currency**
Annn	**Alphanumeric**
D	**Date**

The field type is returned i~ **Table.Field.FieldType**. Upon a successfull return the **GetFieldType** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

Example

```
...
...
...
'A variable was declared elsewhere in this program as:
'Dim Customer As DataTable

Dim status As Integer

Customer.Field.FieldName = "Name"            'Determine the data type of the Name field.
status = GetFieldType(Customer)    'Get the field type
Text1.Text = Customer.Field.FieldType    'Now display the field type in the Text1 control.

...
...
...
```

GetRecord

Description

This function reads the current record in the database table.

Syntax

GetRecord(*Table* **As** DataTable)

Remarks

This function, if successfull, reads the current record in the database table.The database table and the current record for that table are passed in the DataTable argument variable. Upon a successfull return the **GetRecord** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

See Also

FirstRecord, LastRecord, NextRecord and PreviousRecord.

Example

```
Sub GetCustomer_Click ()

'A variable was declared elsewhere in the program as:
'Dim Customer As DataTable
Dim status As Integer

status = GetRecord(Customer)

End Sub
```

GetRecordNumber

Description

This function returns the database record number of the current record.

Syntax

GetRecordNumber(*Table* **As** DataTable, *RecordNumber* **As** Long)

Remarks

The **GetRecordNumber** function returns the record number of the current record. The current record and database table are held in the DataTable argument. The record number is returned in the ***RecordNumber*** argument. The ***RecordNumber*** variable must be of type Long. Upon a successfull return the **GetRecordNumber** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

Example

```
Function GetRecNumber( Table As DataTable) As Long

Dim status As Integer
Dim RecordNumber As Long

status = GetRecordNumber(Table,RecordNumber)
'do error handling here.

GetRecNumber = RecordNumber

End Function
```

GetUserName

Description

This function returns the name of the database engine user.

Syntax

GetUserName(*UserName* **As** String)

Remarks

The **GetUserName** function returns the name of the database engine user. The user name is placed in the *UserName* function argument variable of type String. Upon a successfull return the **GetUserName** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

Example

```
...
...
...
Dim status As Integer
Dim UserName As String

status = GetUserName(UserName)
...
...
...
```

GotoRecord

Description

Goes to the specified record number in the database table and makes that record the current record.

Syntax

GotoRecord(*Table* **As** DataTable, *RecordNumber* **As** Long)

Remarks

This function moves to the *RecordNumber* record in the database table and makes that record the current record.

Example

```
Sub GotoRec (RecordNumber As Long)

'A variable was declared elsewhere in this program as:
'Dim Customer As DataTable
Dim status As Integer

status = GotoRecord(Customer,RecordNumber

End Sub
```

InsertRecord

Description

This function inserts a record into the database table file.

Syntax

InsertRecord(*Table* **As** DataTable)

Remarks

This function inserts a record into the database table file. If the database file is indexed the **InsertRecord** function works similar to the **AppendRecord** function call and the record is inserted in the database file at a location specified by the index. If the database file is not indexed the new record is inserted before the current record. In both cases the newly inserted record becomes the current record. Upon a successfull return the **InsertRecord** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

See Also

AppendRecord, UpdateRecord, DeleteRecord.

Example

```
Sub InsrtRecord_Click ()

' A variable is dimensioned elsewhere in the program as:
' DIM Customer As DataTable

DIM status As Integer

status = InsertRecord(Customer)     'Insert record to table

End Sub
```

IsFieldBlank

Description

This function determines whether or not a field is blank.

Syntax

IsFieldBlank(*Table* **As** DataTable, *Blank* **As** Integer)

Remarks

This function tests a field's value and indicates whether or not the field is blank. If the field's value is indeed blank, the *Blank* argument variable is set to a non-zero falue. If the field's value is not blank, the *Blank* argument variable is set **FALSE** (0). Blank field values represent "values not yet entered" and are valid values for all database table data types. Upon a successfull return the **IsFieldBlank** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

Example

```
Function IsFldBlank (Table As DataTable) As Integer

Dim status As Integer
Dim BlankStatus             'Set False if field is not blank.

status = IsFieldBlank(Table, BlankStatus)
'do any error handling here

IsFldBlank = BlankStatus

End Function
```

IsRecordLocked

Description

This function tests to see if the current record is locked.

Syntax

IsRecordLocked(*Table* **As** DataTable, *Locked* **As** Integer)

Remarks

This function performs a test to see if the current database record is locked. If the current record is locked, the *Locked* argument variable is set to a non-zero value. If the current record is not locked the *Locked* argument variable is set to False (0). Upon a successfull return the **IsRecordLocked** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

See Also

LockRecord

Example

```
Function RecordLocked (Table As DataTable) As Integer

Dim status As Integer
Dim Locked As Integer

status = IsRecordLocked(Table, Locked)
'do IsRecordLocked error handling here.
```

RecordLocked = Locked

End Function

LastRecord

Description

This function moves to the last record in the database table.

Syntax

LastRecord(*Table* **As** DataTable)

Remarks

This function moves to the last record in the database table and makes that record the current record. The database table is specified in the *Table* argument. The Table must have been successfully opened with a previous call to **OpenTable**. Upon a successfull return the **LastRecord** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

See Also

FirstRecord, NextRecord and PreviousRecord

Example

...
...
...
'A variable was declared elsewhere in the program as:
'Dim Customer As DataTable

Dim status As Integer

status = LastRecord(Customer) 'move to the last record in the table specified by Customer.
'd LastRecord error handling here. 'status holds the return vlaue of the LastRecord function call.

...
...
...

LockRecord

Description

This function locks the current database record.

Syntax

LockRecord(*Table* **As** DataTable)

Remarks

This function locks the current record. The database table and it's current record are specified by the *Table* argument variable. Once the record is successfully locked, no other users are able to delete, or otherwise write to the record until the record is unlocked with a call to the **UnlockRecord** function call. Upo~ a

successfull return the **LockRecord** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

See Also

UnlockRecord

Example

```
...
...
...
'A variable was declared elsewhere in the program as:
'Dim Customer As DataTable

Dim status As Integer

status = LockRecord(Customer)
'do LockRecord error handling here.
...
...
...
```

LockTable

Description

This function locks a database table.

Syntax

LockTable(*Table* **As** DataTable, *LockType* **As** Integer)

Remarks

This function locks a database table with the lock type specified by the *LockType* argument. The *LockType* can be one of the following:

- **FULLLOCK**	(Place a read/write lock on the table)
- WRITELOCK	(Place a write lock on the table)
- PREVENTWRITELOCK	(Prevent write locking on a table)

Once successfully locked, the lock is in effect until a call to the **UnlockTable** function call (with the same *Table* and *LockType* arguments) is made to release the lock. You can place more than one lock on a table. Certain types of locks take precedence over others. A **FULLLOCK** overrides a **WRITELOCK**. As you might expect, if one user placed a **PREVENTWRITELOCK** on a table, another user would not be able to successfully place a **WRITELOCK** on the same table. Upon a successfull return the **LockTable** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

See Also

UnlockTable

Example

```
...
...
...
'A variable was declared elsewhere in the program as:
'Dim Customer As DataTable

Dim status As Integer
```

```
status = LockTable(Customer, PREVENTWRITELOCK)   'Prevent other users from placing table based locks.
'do LockTable function error handling here.

...
...
...
```

NRecords

Description

Returns the number of records present in the database table.

Syntax

NRecords(*Table* **As** DataTable, *NRecords* **As** Long)

Remarks

This function returns the total number of records present in the database table specified in the *Table* argument. The number of records is placed in the *NRecords* argument variable. Upon a successfull return the **NRecords** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

Example

```
...
...
...
'A variable was declared elsewhere in the program as:
'Dim Customer As DataTable

Dim status As Integer
Dim Records As Long

status = NRecords(Customer, Records)                    'Get the number of records in the database
Text1.Text = Str$(Records)                      'Display the number of records in a Visual Basic Text box
control.
...
...
...
```

NextRecord

Description

This function moves to the next record in the database table.

Syntax

NextRecord(*Table* **As** DataTable)

Remarks

This function moves to the next record in the database table and makes that record the current record. The database table is specified in the *Table* argument. Upon a successfull return the **NextRecord** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

See Also

FirstRecord, LastRecord, and PreviousRecord.

Example

```
...
...
...
'A variable was declared elsewhere in the program as:
'Dim Customer As DataTable

Dim status As Integer

status = NextRecord(Customer)                'Move to the next record in the database
'do any error handling here.
...
...
...
```

OpenEngine

Description

This function initializes the database engine for subsequent database operations.

Syntax

OpenEngine(*ApplicationName* **As** String)

Remarks

This function initializes the database engine environment and must be successfully called before any other database function can be performed. This function sets-up the database engine, allocates memory and various other internal database engine environmental settings. The *ApplicationName* argument variable of type String should contain the name of your application program. When you are finished with databse engine processing, you should call the **CloseEngine** function to perfom housekeeping clean-up for the database engine environment. Upon a successfull return the **OpenEngine** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

See Also

CloseEngine

Example

```
Function SetupDatabaseEngine (ProgramName As String) As Integer

Dim status As Integer

status = OpenEngine(ProgramName)          'Initialize the database engine environment.

SetupDatabaseEngine = status

End Function
```

OpenTable

Description

This function opens a database table file for subsequent processing.

Syntax

OpenTable(*Table* **As** DataTable)

Remarks

Before you can process information in a database table file, you must first open that file for processing. You open database table files by calling the **OpenTable** function. To successfully open a database table you will need to specify three parameters in the *Table* DataTable argument (for more information on the DataTable data structure see the **VBENGINE Data Structure Definition** section in this manual):

***Table*.Table.TableName =**
***Table*.Table.IndexID =**
***Table.*Table.SaveEveryChange =**

***Table*.Table.TableName** should hold the name of the database table file including any MSDOS PATH specifier. Do not include the file extension.

***Table*.Table.IndexID** should specify the index you wish to use for table operations. **MASTERINDEX** should be used to open the table with all of it's associated indexes. For a specific index, specify the field number of the associated index.

***Table.*Table.SaveEveryChange** should specify whether you wish to save every change to disk or whether you wish to buffer changes to disk. Buffering is faster, but you may lose data if the power goes out (see **FlushBuffers** for information on writing buffered data to disk). To buffer changes set this parameter to FALSE.

Once these three DataTable parameters have been appropriately set, call **OpenTable** to open the database table. Upon a successfull return the **OpenTable** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

See Also

CloseTable, FlushBuffers, and CloseEngine.

Example

```
...
...
...
Dim Customer As DataTable                       'Declare a variable of type DataTable to interface
                                                ' with the database file.

Dim status As Integer                           'Declare a variable to hold VBENGINE function call results.

Customer.Table.TableName = "C:\Customer" 'Specify the data table file name, include PATH specifier C:\
Customer.Table.IndexID = MASTERINDEX     'We will use all table indexes.
Customer.Table.SaveEveryChange = FALSE   'We will buffer data changes to disk for performance reasons.

status = OpenTable(Customer)                    'Ok, open it up!

...
...
...
```

PreviousRecord

Description

This function moves to the previous record in the database table.

Syntax

PreviousRecord(*Table* **As** DataTable)

Remarks

This function moves to the previous record in the database table and makes that record the current record. The database table is specified in the *Table* argument. Upon a successfull return the **PreviousRecord** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

See Also

FirstRecord, LastRecord, and NextRecord.

Example

```
...
...
...
'A variable was declared elsewhere in the program as:
'Dim Customer As DataTable

Dim status As Integer

status = PreviousRecord(Customer)          'Move to the previous record in the database
'do any error handling here.
...
...
...
```

PutBlank

Description

This function places a blank value into the specified field in the database record.

Syntax

PutBlank(*Table* **As** DataTable)

Remarks

This function places a blank value into the field specified in the *Table* argument (*Table.Field.FieldName*). The field value is not written to the database table until the record is written to disk using either **InsertRecord, AppendRecord,** or **UpdateRecord**. A blank value of the appropriate data type is placed in the field automatically. A blank value is a valid value which represents the fact that the value has yet to be entered (a blank value is not zero.) Upon a successfull return the **PutBlank** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

See Also

IsFieldBlank

Example

```
...
...
...
'A variable was declared elsewhere in the program as:
'Dim Customer As DataTable

Dim status As Integer

Customer.Field.FieldName = "Address"                'We will put a blank value in the Address field.
status = PutBlank(Customer)                         'Move to the previous record in the database
'do any error handling here.
...
...
...
```

PutField

Description

This function places a field value into the specified field in the database record.

Syntax

PutField(*Table* **As** DataTable)

Remarks

This function places the value found in ***Table.Field.FieldValue*** for the field ***Table.Field.FieldName*** into the database record. The record in the database table file is not actually modified until a call to **InsertRecord, AppendRecord,** or **UpdateRecord** is called. The table and record for the operation is specified by the *Table* argument variable. All field values to be written to a database field are placed in ***Table.Field.FieldValue*** and are of type String regardless of the actual data type of the field in the database table itself. The **PutField** function automatically converts the value to the appropriate type before placing it in the database record. Upon a successfull return the **PutField** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned. The steps required to store a Visual Basic text string into a field in a database table are shown below:

See Also

GetField, PutBlank.

Example

```
Sub PutAField ()

Dim Customer As DataTable                       'Create a DataTable variable to manipulate a
database file.
Dim status As Integer                       'Variable for error handling..

'Set-up the Customer DataTable variable to access the database file.
Customer.Table.TableName = "C:\Customer"  'We will use the C:\CUSTOMER.DB database file.
Customer.Table.IndexID = MASTERINDEX      'We will view the database through its Primary index.
Customer.Table.SaveEveryChange = FALSE    'We will buffer any changes to disk.

'OK, the DataTable variable is set up for Table specific access.
'Now lets initialize the database engine. We will assume that all will go well and will not complicate this
example
'with specific error handling code.

status = OpenEngine("Visual Basic Program")

'Ok the database engine is now up and running, now lets open up our database table:
```

```
status = OpenTable(Customer)

'Ok, our table is open, lets get the first record in the Customer table:

Customer.Field.FieldName = "Name"                    'Customer name is stored in a field called Name.

status = GetRecord(Customer)                         'Tables are automatically at the first record when
initially opened.

'Now get the customer's name from a Visual Basic Text control.

Customer.Field.FieldValue = Text1.Text               'Get the value from a Text box control.


'Ok the customer's name is no in Customer.Field.FieldValue.
'Let's put it in the Database

status = PutField(Customer)                          'Place the field in the record
status = UpdateRecord(Customer)              'Update the record

'Ok, a job well done. Lets stop, we will need to close our table, and the database engine before we quit:

status = CloseTable(Customer)                        'Close the database.
status = CloseEngine()                       'Close the database engine

End Sub
```

RefreshTable

Description

This function refreshes or updates a table image to reveal up-to-the minute changes.

Syntax

RefreshTable(*Table* **As** DataTable)

Remarks

This function updates the table image to reflect any changes to data that other users may have made since your last table refresh. The following functions automatically refresh a table image **RecordLock, UpdateRecord, InsertRecord, AppendRecord, and DeleteRecord**. Upon a successfull return the **PutField** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

Example

```
...
...
...
'A variable was declared elsewhere in the program as:
'Dim Customer As DataTable

Dim status As Integer

status = RefreshTable(Customer)
...
...
...
```

RemoveKey

Description

This function deletes a specified database index.

Syntax

RemoveKey(*TableName* **As** String, *IndexID* **As** Integer)

Remarks

This function removes or deletes a database index. The index to be removed is specified by the *IndexID* argument. If *IndexID* = 0 (PRIMARY) then the Primary as well as all Secondary indexes will be removed since Secondary indexes are based on the Primary index. The *IndexID* should equal the field number of the index to be removed.

If a database table (C:\EXAMPLE) has three fields; Number **(1)**, Name **(2)** and Phone number **(3)** (in that order), the Number field is the only key field in the Primary index, and there are secondary indexes (1 for name and 1 for Phone number). To remove the Secondary index for the Phone number field the function would be called as:

status = RemoveKey("C:\EXAMPLE",3)

When this function is called, a **FULLLOCK** is placed on the table during the idex removal process . If the lock attempt fails, so does the function call. Upon a successfull return the **RemoveKey** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

See Also

AddKey

Example

```
...
...
...
Dim status As Integer

status = RemoveKey("C:\CUSTOMER",2)
...
...
...
```

RemovePassword

Description

Removes a password from the database engine environment.

Syntax

RemovePassword(Password **As** String)

Remarks

If database engine resources have been **protected** by a password, users must provide the necessary password to gain access to those resources. The **AddPassword** function call submits a password on your applications behalf. Any database engine resources (Tables) which require the password are **automatically** available for routine manipulation once that password has been submitted with the **AddPassword** function call. The **RemovePassword** function removes the password from the system. Any

resources you had access to through the password are then unavailable once the password has been removed via the **RemovePassword** function call. Upon a successfull return the **RemovePassword** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

See Also

AddPassword

Example

```
Sub RMPassword_Click ()

DIM result As Integer

status = RemovePassword("Bryan sent me")

End Sub
```

SearchField

Description

This function searches a database table file on a specified field.

Syntax

SearchField(*Table* **As** DataTable)

Remarks

This function searches through the database table for a value in a field. The database field searched on is specified by ***Table.*Field.FieldName** the field value to search for is specified by ***Table.*Field.FieldValue**. You need to set these two parameters or data structure members and then call the **PutField** function. After that you need to specify your search mode preference by setting ***Table.*Record.SearchMode** to one of three values:

- SEARCHFIRST
- SEARCHNEXT
- CLOSESTRECORD

SEARCHFIRST begins the search at the first record in the database, the record position of the current record is not changed if a search attempt fails to find a match.

SEARCHNEXT begins with the record following the current record in the database, the record position of the current record is not changed if a search attempt fails to find a match.

CLOSESTRECORD begins to search at the first record in the database, if a record is not found (search attempt fails), one of two possibilities exist:

-If there is no exact match, there happens to be a record which has a value lexically greater than the search value. The current record in the database will be the record with the first such instance and a record not found error (89) returned.

- There is no record in the database that has a value greater or equal to the search value. The current record will be the last record in the database and a record not found error (89) returned.

A search can then be started with a call to the **SearchField** function.

The available search modes rely on the index on which the table is currently using. **SearchField** always searches for the first record which fullfills the search criteria. On non-indexed database tables **SearchField** searches via a sequential scan. The order of the records searched through the sequntial scan is that of the physical order of the records in the table itself. In non-indexed tables **CLOSESTRECORD** is not supported. Upon a successfull return the **SearchField** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

See Also

SearchKey

Example

```
'This example searches the Customer database for a specific customers name.
'The data for the Customer table is passed in the Customer argument of type DataTable.
'The customer name to search for is passed in the CustomerName argument of type String.

Function CustomerSearch(Customer As DataTable, CustomerName As String) As Integer
Dim status As Integer

Customer.Field.FieldName = "Name"                    'We will search on the Name field.
Customer.Field.FieldValue = CustomerName             'For the name in CustomerName string.
Customer.Record.SearchMode = SEARCHFIRST             'Start searching from the first record.

status = PutField(Customer)                          'Submit the search criteria.
'do any desired error handling for the PutField function call here

status = SearchField(Customer)                       'Start the search.
'do any desired error handling for the SearchField function call here

CustomerSearch = status                              'Return the search status

End Function
```

SearchKey

Description

This function searches a database table for a key match.

Syntax

SearchKey(*Table* **As** DataTable)

Remarks

This function searches the table specified in ***Table.*Table.TableName** on the Primary index. A search match is sought on the key field(s) of the table specified by ***Table.*Record.SearchKey**. The key to be matched must be the primary key or a subset of the primary key. The fields to be matched are the fields which have been placed into the database engine's record buffer via calls to **PutField**.

If there are five key fields and you are only interested in finding records which have specific values in the first two key fields lets say "Date" and "Customer Name", you want to search for records in the database that have 12/12/92 for the "Date" value and "Robert Smith" for the "Customer Name" you would set the criteria for those fields and place them in the database engine via calls to **PutField.** Your KeySearch would be set up as ***Table.*Record.KeySearch** = 2.

You need to specify your search mode preference by setting ***Table.*Record.SearchMode** to one of three values:

- SEARCHFIRST

- SEARCHNEXT
- CLOSESTRECORD

SEARCHFIRST begins the search at the first record in the database, the record position of the current record is not changed if a search attempt fails to find a match.

SEARCHNEXT begins with the record following the current record in the database, the record position of the current record is not changed if a search attempt fails to find a match.

CLOSESTRECORD begins to search at the first record in the database, if a record is not found (search attempt fails), one of two possibilities exist:

-If there is no exact match, there happens to be a record which has a value lexically greater than the search value. The current record in the database will be the record with the first such instance and a record not found error (89) returned.

- There is no record in the database that has a value greater or equal to the search value. The current record will be the last record in the database and a record not found error (101) returned.

The available search modes rely on the index on which the table is currently using. **SearchKey** always searches for the first record that fullfills the search criteria. Once the desired key fields have been set-up and submitted via calls to **PutField**, the desired search mode specified, along with the keysearch specification, you can then call **SearchKey**. Upon a successfull return the **SearchKey** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

See Also

SearchField and Database Searching in the **Database Fundamentals** section of this manual.

Example

```
'This example searches the Customer database for a specific customers name.
'The data for the Customer table is passed in the Customer argument of type DataTable.
'The customer name to search for is passed in the CustomerName argument of type String.

Function CustomerSearch(Customer As DataTable, CustomerName As String) As Integer
Dim status As Integer

Customer.Field.FieldName = "Name"                    'We will search on the Name field.
Customer.Field.FieldValue = CustomerName             'For the name in CustomerName string.
Customer.Record.SearchMode = SEARCHFIRST             'Start searching from the first record.
Customer.Record.KeySearch = 1                        'Customer name field only keyed field in table

status = PutField(Customer)                          'Submit the search criteria.
'do any desired error handling for the PutField function call here

status = SearchKey(Customer)                         'Start the search.
'do any desired error handling for the SearchKey function call here

CustomerSearch = status                              'Return the search status

End Function
```

UnlockRecord

Description

This function unlocks a previously locked record.

Syntax

UnlockRecord(*Table* **As** DataTable)

Remarks

This function unlocks a previously locked record. You are only able to unlock records that you have previously locked. You can not unlock records locked by other users. A locked record can also be unlocked under the following conditionss:

- You delete the record by calling **DeleteRecord**.

- You call **CloseTable** which unlocks all the records in that table before closing the table.

- You call **CloseEngine** which unlocks all records in your tables.

Upon a successfull return the **UnlockRecord** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

See Also

LockRecord

Example

```
...
...
...
'A variable was declared elsewhere in this program as:
'Dim Customer As DataTable

Dim status As Integer

status = LockRecord(Customer)                'Lock the Record

If (status = 0) Then                                    'If locked, then unlock

        status = UnlockRecord(Customer)
End If
...
...
...
```

UnlockTable

Description

This function unlocks a previously locked table.

Syntax

UnlockTable(*Table* **As** DataTable, *LockType* **As** Integer)

Remarks

This function unlocks a previously locked database table. The target table is specified by ***Table.*Table.TableName**. To unlock the table the *LockType* argument must be the same value used when locking the table. The *LockType* can be one of the following:

- **FULLLOCK** (Place a read/write lock on the table)
- **WRITELOCK** (Place a write lock on the table)

- PREVENTWRITELOCK (Prevent write locking on a table)

Once successfully locked, the lock is in effect until a call to the **UnlockTable** function (with the same *Table* and *LockType* arguments) is made to release the lock. You can place more than one lock on a table. Certain types of locks take precedence over others. A **FULLLOCK** overrides a **WRITELOCK**. As you might expect, if one user placed a **PREVENTWRITELOCK** on a table, another user would not be able to successfully place a **WRITELOCK** on the same table. Upon a successfull return the **UnlockTable** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

See Also

LockTable

Example

```
...
...
...
'A variable was declared elsewhere in the program as:
'Dim Customer As DataTable

Dim status As Integer

status = LockTable(Customer, PREVENTWRITELOCK)   'Lock the table
'do LockTable function error handling here.

status = UnlockTable(Customer, PREVENTWRITELOCK)          'Now unlock it
'do UnlockTable function error handling here.

...
...
...
```

UpdateRecord

Description

This function updates a record in a database table.

Syntax

UpdateRecord(*Table* **As** DataTable)

Remarks

This function updates the record specified in the DataTable argument variable to the database file. There must be a current database record to update. Upon a successfull return the **UpdateRecord** function returns an integer value of zero (0). In the event of an error, a non-zero integer error value is returned.

See Also

AppendRecord, InsertRecord, DeleteRecord.

Example

```
Sub UpdateRecord_Click ()

' A variable is dimensioned elsewhere in the program as:
' DIM Customer As DataTable

DIM status As Integer
```

```
status = UpdateRecord(Customer)   'Append record to table

End Sub
```

VBENGINE / PARADOX ENGINE ERROR CODES

Error Code	Description
1	Drive not ready
2	Directory not found
3	File is busy
4	File is locked
5	File not found
6	Table damaged
7	Primary index damaged
8	Primary index is out of date
9	Record is locked
10	Sharing violation - directory busy
11	Sharing violation - directory locked
12	No access to directory
13	Sort for index different from table
14	Single user but directory is shared
15	Multiple PARADOX.NET files found
21	Insufficient password rights
22	Table is write-protected
30	Data type mismatch
31	Argument is out of range
33	Invalid argument
40	Not enough memory to complete operation
41	Not enough disk space to complete operation
50	Another user deleted record
70	No more file handles available
72	No more table handles available
73	Invalid date given
74	Invalid field name
75	Invalid field handle
76	Invalid table handle
78	Engine not initialized
79	Previous fatal error, cannot proceed
81	Table structures are different
82	Engine already initialized
83	Unable to perform operation on open table
86	No more temporary names available
89	Record was not found
94	Table is indexed
95	Table is no| indexed
96	Secondary index is out of date
97	Key violation
98	Could not login on network
99	Invalid table name
101	End of table
102	Start of table
103	No more record handles available
104	Invalid record handle
105	Operation on empty table
106	Invalid lock code
107	Engine not initialized
108	Invalid file name

109	Invalid lock
110	Invalid lock handle
111	Too many locks on table
112	Invalid sort-order table
113	Invalid net type
114	Invalid directory name
115	Too many passwords specified
116	Invalid password
117	Buffer too small for result
118	Table is busy
119	Table is locked
120	Table was not found
121	Secondary index was not found
122	Secondary index is damaged
123	Secondary index is already open
124	Disk is write-protected
125	Record is too big for index
126	General hardware error
127	Not enough stack space to complete operation
128	Table is full
129	Not enough swap buffer space to complete operation
130	Table is SQL replica
131	Too many clients for Engine DLL
132	Exceeds limits specified in WIN.INI
133	Too many files open simultaneously (includes all clients)
134	Can't lock PARADOX.NET - is SHARE.EXE loaded
135	Can't run Engine in Windows real mode
136	Can't modify unkeyed table with non-maintained secondary index