Rocket

UniData

UniObjects Developer's Guide

UDT-720-UOBJ-1

Notices

Edition

Publication date: July 2008

Book number: UDT-720-UOBJ-1

Product version: UniData 7.2

Copyright

© Rocket Software, Inc. 1988-2008. All Rights Reserved.

Trademarks

The following trademarks appear in this publication:

Trademark	Trademark Owner
Rocket Software™	Rocket Software, Inc.
Dynamic Connect®	Rocket Software, Inc.
RedBack®	Rocket Software, Inc.
SystemBuilder™	Rocket Software, Inc.
UniData®	Rocket Software, Inc.
UniVerse™	Rocket Software, Inc.
U2™	Rocket Software, Inc.
U2.NET™	Rocket Software, Inc.
U2 Web Development Environment™	Rocket Software, Inc.
wIntegrate®	Rocket Software, Inc.
Microsoft® .NET	Microsoft Corporation
Microsoft® Office Excel®, Outlook®, Word	Microsoft Corporation
Windows®	Microsoft Corporation
Windows® 7	Microsoft Corporation
Windows Vista®	Microsoft Corporation
Java™ and all Java-based trademarks and logos	Sun Microsystems, Inc.
UNIX®	X/Open Company Limited

The above trademarks are property of the specified companies in the United States, other countries, or both. All other products or services mentioned in this document may be covered by the trademarks, service marks, or product names as designated by the companies who own or market them.

License agreement

This software and the associated documentation are proprietary and confidential to Rocket Software, Inc., are furnished under license, and may be used and copied only in accordance with the terms of such license and with the inclusion of the copyright notice. This software and any copies thereof may not be provided or otherwise made available to any other person. No title to or ownership of the software and associated documentation is hereby transferred. Any unauthorized use or reproduction of this software or documentation may be subject to civil or criminal liability. The information in the software and documentation is subject to change and should not be construed as a commitment by Rocket Software, Inc.

Restricted rights notice for license to the U.S. Government: Use, reproduction, or disclosure is subject to restrictions as stated in the “Rights in Technical Data-General” clause (alternate III), in FAR section 52.222-14. All title and ownership in this computer software remain with Rocket Software, Inc.

Note

This product may contain encryption technology. Many countries prohibit or restrict the use, import, or export of encryption technologies, and current use, import, and export regulations should be followed when exporting this product.

Please be aware: Any images or indications reflecting ownership or branding of the product(s) documented herein may or may not reflect the current legal ownership of the intellectual property rights associated with such product(s). All right and title to the product(s) documented herein belong solely to Rocket Software, Inc. and its subsidiaries, notwithstanding any notices (including screen captures) or any other indications to the contrary.

Contact information

Rocket Software
275 Grove Street Suite 3-410
Newton, MA 02466-2272
USA
Tel: (617) 614-4321 Fax: (617) 630-7100
Web Site: www.rocketsoftware.com

Table of Contents

Preface

This book describes UniObjects, an interface to UniVerse and UniData databases from Visual Basic (or any other program development environment that uses the Microsoft OLE Automation interface). The book is intended for experienced programmers and application developers who want to write Visual Basic programs that access UniVerse and UniData databases. The book assumes that you are familiar with your server database and with Visual Basic. If you are new to Visual Basic, read one or more of the books in "Additional Reference" on page xvii. If you are new to UniVerse or UniData, you should read at least "The Database Environment" in Chapter 2, "Using UniObjects."

Organization of This Manual

This manual contains the following:

Chapter 1, "Your First UniObjects Application," shows how easy it is to write your first application using UniObjects.

Chapter 2, "Using UniObjects," outlines the database environment and explains how to use UniObjects to connect to the database, open files, access records, and so forth.

Chapter 3, "A Tour of the Objects," is a guide to all the objects, methods and properties that are available with UniObjects, and includes code examples for most objects.

Chapter 4, "Distributing Your Application," tells you how to package your UniObjects application for distribution.

Appendix A, "Error Codes and Replace Tokens," lists error codes you may encounter when programming with UniObjects.

Appendix B, "The Demo Application," gives a sample application created with UniObjects.

Appendix C, "Data Conversion Functions," describes some Visual Basic data conversion functions that are supplied with UniObjects.

Documentation Conventions

This manual uses the following conventions:

Convention	Usage
Bold	Bold indicates objects, methods, properties, and Visual Basic keywords.
UPPERCASE	Uppercase indicates database commands, file names, keywords, BASIC statements and functions, MS-DOS paths, and text that must be input exactly as shown.
Italic	Italic in a syntax line or an example indicates information that you supply.
`Courier`	Courier indicates examples of source code and system output.
`This line ?continues`	The continuation character is used in source code examples to indicate a line that is too long to fit on the page, but must be entered as a single line on the screen.
[ ]	Brackets enclose optional items. Do not type the brackets unless indicated.

Documentation Conventions

The following conventions are also used:

- Syntax definitions and examples are indented for ease in reading.
- All punctuation marks included in the syntax—for example, commas, parentheses, or quotation marks—are required unless otherwise indicated.

Help

To get Help about UniObjects, choose **Start ?Programs ?IBM U2 ?UniDK ?UniObjects – Help**.

API Documentation

The following books document application programming interfaces (APIs) used for developing client applications that connect to UniVerse and UniData servers.

Administrative Supplement for Client APIs: Introduces IBM's seven common APIs, and provides important information that developers using any of the common APIs will need. It includes information about the UniRPC, the UCI Config Editor, the *ud_database* file, and device licensing.

UCI Developer's Guide: Describes how to use UCI (Uni Call Interface), an interface to UniVerse and UniData databases from C-based client programs. UCI uses ODBC-like function calls to execute SQL statements on local or remote UniVerse and UniData servers. This book is for experienced SQL programmers.

IBM JDBC Driver for UniData and UniVerse: Describes UniJDBC, an interface to UniData and UniVerse databases from JDBC applications. This book is for experienced programmers and application developers who are familiar with UniData and UniVerse, Java, JDBC, and who want to write JDBC applications that access these databases.

InterCall Developer's Guide: Describes how to use the InterCall API to access data on UniVerse and UniData systems from external programs. This book is for experienced programmers who are familiar with UniVerse or UniData.

UniObjects Developer's Guide: Describes UniObjects, an interface to UniVerse and UniData systems from Visual Basic. This book is for experienced programmers and application developers who are familiar with UniVerse or UniData, and with Visual Basic, and who want to write Visual Basic programs that access these databases.

UniObjects for Java Developer's Guide: Describes UniObjects for Java, an interface to UniVerse and UniData systems from Java. This book is for experienced programmers and application developers who are familiar with UniVerse or UniData, and with Java, and who want to write Java programs that access these databases.

UniObjects for .NET Developer's Guide: Describes UniObjects, an interface to UniVerse and UniData systems from .NET. This book is for experienced programmers and application developers who are familiar with UniVerse or UniData, and with .NET, and who want to write .NET programs that access these databases.

Using UniOLEDB: Describes how to use UniOLEDB, an interface to UniVerse and UniData systems for OLE DB consumers. This book is for experienced programmers and application developers who are familiar with UniVerse or UniData, and with OLE DB, and who want to write OLE DB programs that access these databases.

Additional Reference

The following manual provides more information about Visual Basic.

> ***Microsoft Visual Basic Programmer's Guide*** (the appropriate edition for your version of Visual Basic)

Either of the following books may be useful if you are new to programming with Visual Basic:

> ***Visual Basic 6 Database Programming for Dummies***, by Richard Mansfield. ISBN: 0764506250
>
> ***Teach Yourself Visual Basic 6 in 24 Hours***, by Greg Perry with Snjaya Hettihewa. ISBN 0672315335

Chapter

1

Your First UniObjects Application

This chapter shows how to write a simple program with UniObjects in just five minutes. UniObjects forms part of the UniDK (Uni Development Kit). UniObjects provides a fast and simple way to access a UniVerse or UniData database on a server from a client program written in Visual Basic.

All the examples in this book use Visual Basic, but you can also use UniObjects in any program development environment that uses the Microsoft OLE Automation interface.

If you have not already installed UniObjects, install it as described in the installation instructions that came with your installation CD. Also read the online release notes that are installed with the UniDK.

Adding a Component to Visual Basic

If you use Visual Basic Version 4.0 or later, you can add a custom control for UniObjects to the toolbox, as follows:

1. From the Visual Basic Project menu, choose **Components**. You see the Components dialog box.
2. Under the **Controls** tab, select **UniObjects Control 3.x** from the scroll box.

For more information about custom controls, see *Microsoft Visual Basic Programmer's Guide*.

The UniObjects Include File

UniObjects has an include file that contains tokens for error codes, Visual Basic registered object names, and other constants that you may need in your programs. The file is called UVOAIF.TXT and is located in the UniDK installation directory. You can either add the file to your Visual Basic project, or cut and paste the tokens you require onto the form where they are used.

A Five-Minute Program

Once you install UniObjects, you can start programming immediately. This section tells you how to construct a Visual Basic program that starts a database session, opens a database file, and looks at a record. You can try this program to check that your installation is working. If you already know how to use the Visual Basic toolbox, it should take just five minutes. If you have any problems when you try out the program, see "Troubleshooting" on page 1-9.

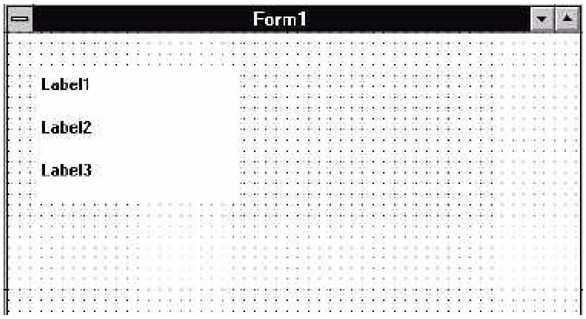
Construct the User Interface

The user interface for the program is a single form with three text boxes and a command button. You enter a file name and a record ID in two of the text boxes, and when the command button is pushed, the program retrieves the specified record from the server and displays it in the third text box.

You construct the user interface for the program as follows:

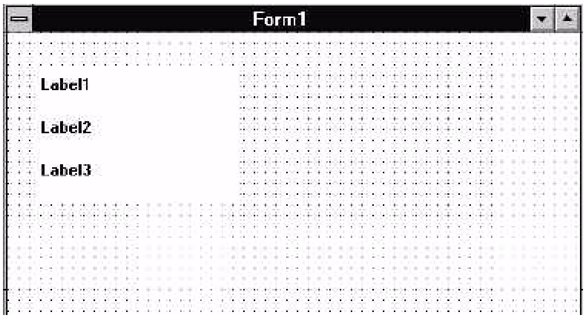
1. Double-click the **Visual Basic** icon to open a new project and form.
2. Draw three labels on the left side of the form as follows:
   - Click the label control on the toolbox.
   - Move the cursor to the form.
   - Drag the cross hair to form a box shape.
   - Repeat twice more.

   Your form now looks similar to this:



3. Add captions to the three labels that read Filename, Record ID, and Result as follows:

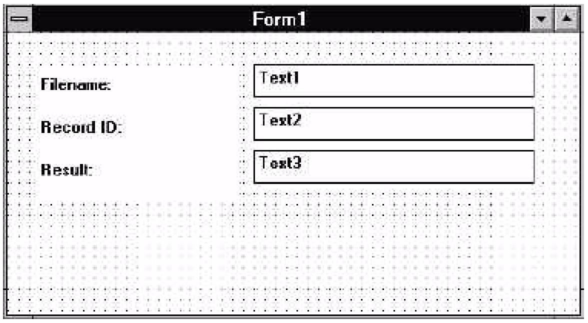
- Click one label.
- Press **F4** to show the Properties window.
- Scroll down to **Caption**, and enter your caption in the Settings box.
- Click back on the form and repeat for the other two labels.
- Your form now looks similar to this:



4. Make three text boxes opposite the labels as follows:

- Click the **Text Box** control on the toolbox.
- Move the cursor to the form.
- Drag the cross hair to form a box shape.
- Repeat twice more.

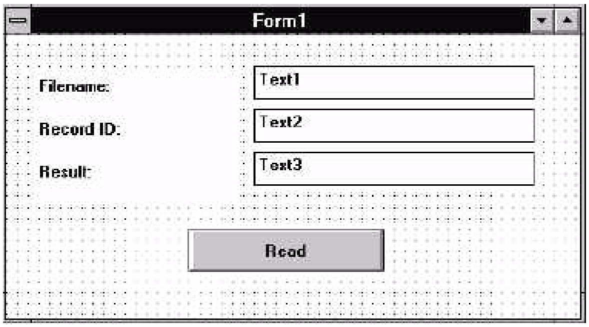
Your screen now looks similar to this:



5. Add a command button at the bottom of the form and change its caption to say Read as follows:

   - Click the **Command** button control.
   - Move the cursor to the form and drag the cross hair to form a box shape.
   - Press **F4** to display the Properties window, scroll to Caption, and enter `Read` in the settings box.

   Your screen now looks similar to this:



At this stage, the user interface is complete. You can test what you have done so far by clicking the **Run** button on the toolbar or by pressing **F5**. You can then see your interface working. You can enter text in the text boxes and press the command button, although, of course, nothing happens yet. Close the window to get back to the Visual Basic development environment.

Write the Program

The next stage is to write the program. This involves three short snippets of code that:

- Start the database session when the form is loaded
- Define a variable for the database session
- Specify the action for the command button, that is, open the specified database file and read the specified record

You only need to enter 10 lines of code.

1. Double-click the white space of your form to open the code window. This automatically opens the Form_Load subroutine which already has the following lines:

   ```
   Sub Form_Load()

   End Sub
   ```

   You need to add four lines as follows:

   ```
   Sub Form_Load()
       Set Session = CreateObject ("UniObjects.unioaifctrl")
       Session.UserName = InputBox ("User Name:","Login")
       Session.Password = InputBox ("Password:","Password")
       Session.Connect
   End Sub
   ```

   ***Note**: "UniObjects.unioaifctrl" is the registered object name for a database **Session** object. If you add the UVOAIF.TXT file to the Visual Basic project, you can specify this as the replace token UNI_SESSION_OBJECT..*

2. In the same editor window, click the object box and choose **general**. The editor window should be empty. Enter the following line:

   Dim Session As Object

3. From the design window containing the labels, text boxes and command button, double-click the command button. The following code appears:

   ```
   Sub Command1_Click ()

   End Sub
   ```

   You need to add lines to read as follows:

   ```
   Sub Command1_Click ()
       Dim FileObj As Object

       If Session.IsActive Then
           Set FileObj = Session.OpenFile (Text1.Text)
           FileObj.RecordId = Text2.Text
           FileObj.Read
           Text3.Text = FileObj.Record
       End If
   End Sub
   ```

Test the Program

That completes the program. You can test it by doing the following:

1. Click **Run** on the toolbar or press **F5**.
2. You are prompted to enter the database account you want to use for the session. Enter the name of the computer on which you installed the database server, and either the path of an account, or a valid account name. (That is, one that is used in the ACCOUNTS file on the server.)
3. This step depends on the kind of server you connect to.
   - **On UniVerse systems:** When the form you designed appears, try entering **`VOC`** as the filename and **`RELLEVEL`** for the record ID.
   - **On UniData systems:** When the form you designed appears, try entering **`VOC`** as the filename and **`VERSION`** for the record ID.
4. Click **Read**. The RELLEVEL or VERSION record appears in the Result box. Try other records in the VOC file, or other files you have on the server.
5. Close the application to return to the Visual Basic development environment.

Troubleshooting

This section gives some pointers to help you if you cannot get the test program to run.

"My user interface does not work…"

If you have completed step 5 of the program design and your user interface does not work when you try it, it is likely that Visual Basic is not installed correctly or is incomplete. Try reinstalling Visual Basic.

"I get an error box saying: cannot create object…"

If you see this error when you try to run the test program, it is likely that UniObjects itself is not installed properly. Try repeating the installation steps described in the installation instructions.

“The test program runs OK but I do not get a result…”

If your test program seems to work but does not return anything in the result box, this suggests a problem with the server. Possible causes are:

- The database is not installed or is not running correctly on the server.
- The user name you specified is incorrect.
- The password you specified is incorrect.
- The host name you specified is incorrect.
- The account path you specified is incorrect.
- The file name or the record ID you specified does not exist.

You can use the Visual Basic debugger to find out where the problem is happening. Put break points in after these lines:

```
Set Session = CreateObject ("UniObjects.unioaifctrl")

Session.Connect

FileObj.Read
```

Then run the program and check the **Error** properties of the **Session** and **File** objects. The error codes you might see are listed in Appendix A, “Error Codes and Replace Tokens,” or in UniObjects Help.

“It still will not work…”

Telephone the customer support number if you are in the U.S. or contact your local IBM supplier.

Chapter

Using UniObjects

2

This chapter explains how to use UniVerse or UniData in a Visual Basic program. The topics covered include:

- An overview of the database environment
- Opening and controlling a database session
- Accessing files
- Locking records
- Handling errors
- Using dictionaries
- Accessing UniVerse sequential files (that is, text files and binary files)
- Executing database commands
- Running subroutines on the server

If you are new to Visual Basic, read *Microsoft Visual Basic Programmer's Guide* or one of the other books listed under "Additional Reference" in the "Preface" before you start this chapter.

For full details and more examples of all the objects, methods, and properties mentioned in this chapter, see Chapter 3, "A Tour of the Objects."

For information about packaging your application for distribution, see Chapter 4, "Distributing Your Application."

The Database Environment

This section tells you just enough about the database environment to enable you to understand the rest of the chapter. If you already know about UniVerse or UniData, skip to "UniObjects Concepts" on page 2-7. To learn more about UniVerse, read *The Database Environment*. To learn more about UniData, read *Using UniData* and *Administering UniData*.

A database user logs on to an account. An account is an operating system directory containing system files, database files, and possibly operating system files and directories too.

Each database file comprises a data file, which contains the data records, and a file dictionary, which defines the structure of the data records, how they are displayed, and so forth. Each record in a file is uniquely identified by a record ID, which is stored separately from the data to which it refers.

The VOC file in an account contains a record for every file used in the database. This record provides a cross-reference between the file name, which is the record ID, and the path of the file, which is contained in another field.

***Note**: UniVerse has several account flavors. The following sections describe IDEAL flavor, which is the recommended flavor to use with UniObjects. UniData uses ECLTYPE and BASICTYPE to specify account flavors. See Using UniData for information about UniData flavors.*

UniVerse and UniData Data Structure

In a business application, each file holds one type of record only. For example, a file called CUSTOMER might hold one record for each customer, while another file called ORDERS holds one record for each order placed by any customer. The records and the fields they contain are not fixed in size, and the file itself can grow or shrink according to the amount of data it holds.

The data is held in fields within a record. For example, a record in the CUSTOMER file might have fields containing the name, address, and telephone number of the customer concerned. A field can hold multiple values, for example, the separate elements of an address can be stored as multivalues of one field, rather than as separate fields in the record. A field in one record can form the basis of a cross-reference to data held in another file. For example, to link a customer with the orders made by that customer, records in the CUSTOMER file might have a multivalued field containing a list of the corresponding record IDs in the ORDERS file.

File Dictionaries

The file dictionary holds information about the structure of a record and its relationship to other files. Within a record, each field is identified by a number, and the dictionary acts as a cross-reference between that number and the name of the field. For example, the customer's phone number might be held in a field called CUST.PHONE, which is field 3 in the record.

The file dictionary also defines the way the data in the field is formatted and displayed for output, for example, the column heading and the column width used in a report. All data is stored as character strings. Some data, such as monetary amounts and dates is stored in a compact, internal format. For these fields the dictionary holds a conversion code, which specifies a conversion to be applied before the data is displayed.

Types of Dictionary Record

There are two main types of dictionary record that are used to define fields:

- D-descriptors, which define the data actually held in a field
- I-descriptors and V-descriptors, which are calculated fields that are evaluated whenever the value is required

I-descriptors are versatile and powerful. They can perform calculations on data stored in one record, or retrieve data from other files using a function called TRANS. For example, as described previously, records in the CUSTOMER file have a field which lists related record IDs in the ORDERS file. The CUSTOMER file dictionary could contain I-descriptors which use the TRANS function to retrieve fields from those related records.

Locks

When a program makes changes to the database, it sets a lock on each record involved in the update. This means that no other user can modify the record until the lock is released. Locks and locking strategy are described in “Record Locks” on page 2-18.

Data Retrieval

UniVerse contains a suite of programs to use with the database, including:

- RetrieVe, a data query and reporting language
- ReVise, a menu-based data entry and modification program
- Editor, a line editor you can use to add, change, or delete records in a UniVerse file

UniData provides a set of similar programs:

- UniQuery, a data query and reporting language
- UniEntry, a menu-based data entry and modification program
- Editor and AE Editor, line editors you can use to add, change, or delete records in a UniData file
- UniData SQL, UniData’s version of the SQL language

Both databases also have many commands and keywords that are used to administer and maintain the database. All these programs and commands can be accessed by a Visual Basic program through UniObjects. For more information, see “Using Database Commands” on page 2-23.

UniObjects Concepts

If you already know UniVerse or UniData, you will find that UniObjects uses some new terms to define familiar database features. This section defines those terms and shows how they map to the database.

Objects

An object is one instance of a group, or class, that shares the same characteristics. The objects that you can use with UniObjects are shown in the following table.

Object	Description
Session	A **Session** object is a reference to a connection between your client program and the database running on the server. You normally access the other objects through the **Session** object.
File	A **File** object is a reference to a data file.
Dictionary	A **Dictionary** object is a reference to a file dictionary.
DynamicArray	A **DynamicArray** object is a reference to a dynamic array, such as a record.
SelectList	A **SelectList** object is a reference to a select list.
SequentialFile	A **SequentialFile** object is a reference to an operating system directory that is used for storing text, programs, or other data.
Command	A **Command** object is a reference to a database command executed on the server.
Subroutine	A **Subroutine** object is a reference to a BASIC subroutine that is called by the client program but runs on the server. For BASIC users, this is the familiar cataloged subroutine.
Transaction	A **Transaction** object is a reference to a transaction for a session.
NLSLocale	(UniVerse only) An **NLSLocale** object is a reference to the locale information for a session.
NLSMap	(UniVerse only) An **NLSMap** object is a reference to the map information for a session.

Objects

Properties

Properties are values that have been assigned to a particular object. For example, a **Session** object has a property called **AccountPath** which contains the name of the database account that the session logs on to.

Methods

Methods are the procedures that can be used with a particular object. Many of the methods used in UniObjects are equivalent to BASIC statements and functions and InterCall functions. For example, the **ClearFile** method is equivalent to the BASIC CLEARFILE statement, and the **ClearList** method is equivalent to the InterCall **ic_clearselect** function.

Accessing Objects

A Visual Basic program accesses objects through the database **Session** custom control. This control represents your connection to the server, and you can add it to your application in one of two ways:

- Select a database **Session** control from the Visual Basic toolbox and drag it onto one of your application's forms at design time. The control is invisible at run time; you just use its properties and methods in your Visual Basic code. (See also "Adding a Component to Visual Basic" on page 1-3.)
- Declare a variable of type **Object**, and then use the **CreateObject** function to create a database **Session** object, as follows:

  ```
  Dim UVSession As Object
  Set UVSession = CreateObject(UV_SESSION_OBJECT)
  ```

Opening a Database Session

You must connect to a database server before you can access files or records. You establish a server session with the **Connect** method of the **Session** object. The server can be the same computer that the client application is running on, or it can be a different computer linked by a network. A connected session is like a login session that would be established by a terminal user.

Once the session is active, you can use it to create other objects. For example, if you want to open a file, or execute a database command, or run a subroutine on the server, you start the operation using the methods provided by the **Session** object.

You must ensure that the **Session** object exists for as long as your application needs access to the server. When a **Session** object is no longer active, the connection with the database server ends. This means that although the objects created through a **Session** object are still available, you may not be able to use them. For example, if you have a **File** object, you can access the last record that was read from the file, but you cannot read another record.

If you add a control to a form, choose a form that will not be unloaded while your application is active. If you declare an object, either declare it globally, or declare it in a form that will not be unloaded.

Methods and Properties Used to Create Objects

The following table shows the UniObjects and the methods and properties that you use to create or access them. The methods and properties all belong to the **Session** object unless otherwise stated.

UniObject	Method or Property
File	**OpenFile** method.
Dictionary	**OpenDictionary** method.
SequentialFile	**OpenSequential** method.
DynamicArray	**Record** property of **File** or **Dictionary** object, or **ReadList** method of **SelectList** object. Can also be created independently.

Methods and Properties Used to Create Objects

UniObject	Method or Property
SelectList	**SelectList** method.
Command	**Command** property.
Subroutine	**Subroutine** method.
Transaction	**Start** method.
NLSLocale	(UniVerse only) **NLSLocale** property.
NLSMap	(UniVerse only) **NLSMap** property.

Methods and Properties Used to Create Objects (Continued)

Using the @TTY Variable

During normal server operations, the @TTY variable on the server is set to the terminal number. If the process is a phantom, @TTY returns the value `phantom`. If the process is an API such as UniObjects or InterCall, @TTY returns the value `uvcs` on UniVerse servers and `udcs` on UniData servers.

You can use this returned value by adding a paragraph entry to the VOC file. For example:

```
PA
IF @TTY = 'uvcs' THEN GO END:
START.APP
END:
```

Using Files

Before you can use a database file you must create a **File** object that refers to the file. To do this, you first declare a variable of type **Object** using either of the following examples:

```
Global CustomerFile As Object
Dim CustomerFile As Object
```

You then open the file to the variable using the **OpenFile** method of the **Session** object as follows:

```
Set CustomerFile = UVSession.OpenFile("CUSTOMER")
```

Note: *When you have opened a file, you should ensure it opened successfully, either by checking the* **Error** *property of the* **Session** *object (which is used like an ELSE clause in a BASIC statement), or by checking if the* **File** *object is* **Nothing**. *For more information, see "Error Handling" on page 2-17.*

Reading and Writing Records

Once you have an open file, you can read data from it and write data to it.

To read a record, set the **RecordId** property to the ID of the record you want, and then call the **Read** method. If the read is successful, the **Record** property contains the required record, for example:

```
Dim CustomerRec As Object
   .
   .
   .
CustomerFile.RecordID = "12345"
CustomerFile.Read
If CustomerFile.Error <> 0 Then
   ' ... error handling here...'
End If
CustomerRec = CustomerFile.Record
```

To write the current data (that is, the current value of the **Record** property) back to the file, you call the **Write** method, as follows:

```
CustomerFile.Write
If CustomerFile.Error <> 0 Then
   ' ... do some error handling'
End If
```

Fields, Values, and Subvalues

When you read a record it is returned as a DynamicArray object. This means you can access and change the fields and values in the record exactly as they are stored in the file.

For example, your application might include lines like these:

```
Const CUSTOMER_NAME = 14
   ' other constants representing the layout of the customer
   ' record
Set CustomerRec = CustomerFile.Record
CustomerNameBox.Text = CustomerRec.Field(CUSTOMER_NAME)
```

This defines the variable CustomerRec as a reference to the CustomerFile.Record.

You can then use the **Field** method to extract or change the value of a specified field, and the **Value** and **SubValue** methods to access a specific value or subvalue. For example, you could append a new value to field 2 of a record as follows:

```
CustomerRec.Value(2, -1) = "some new value"
```

When you make changes to the variable, they are immediately reflected in the **Record** property. If you want a separate copy of the record, you can create an independent **DynamicArray** object using the **CreateObject** function, as follows:

```
Set NewRecord = CreateObject(UV_DARRAY_OBJECT)
```

The result of the **Field**, **Value**, and **SubValue** methods are also **DynamicArray** objects. This means that you can apply other methods to one field or value, or to the entire array, by invoking the method at the correct level, as shown in the following examples.

To count the number of values in field 2 of a dynamic array:

```
NumValues = DynArray.Field(2).Count()
```

To count the number of fields in the entire array:

```
NumValues = DynArray.Count()
```

To insert a new field before field 5 on the dynamic array:

```
DynArray.Field(5).Ins "new value"
```

To delete the fourth subvalue of the first value of field 3:

```
DynArray.SubValue(3, 1, 4).Del
```

The methods you can use in this way are **Count**, **Del**, **Ins**, **Length**, and **Replace**.

Data Conversion

When you read and write an entire record, as described in the previous section, your program must handle conversion of data to and from its internal storage format. You do this through the **Iconv** (input convert) and **Oconv** (output convert) methods of the **Session** object.

For example:

```
DateBox.Text = UVSession.OConv(CustomerRec.Field(LAST_ORDER_DATE), "D")
```

In most cases, the position of the field in the record and the conversion code to apply need to be written into your program. This means that your program may need to change if the structure of the record changes.

As an alternative, you can read or write to a named field, rather than the entire record, and let UniObjects consult the file dictionary and perform any data conversion for you. You do this through the **ReadNamedField** and **WriteNamedField** methods.

***Note**: BASIC does not have equivalents to the* **ReadNamedField** *and* **WriteNamedField** *methods.*

The **ReadNamedField** method of the **File** object lets a program request a data field by name, in its converted form. **ReadNamedField** can also evaluate I-descriptors. For example, the code to read the LAST.ORDER.DATE field might look like this:

```
CustomerFile.RecordID = "12345"
CustomerFile.ReadNamedField "LAST.ORDER.DATE"
If CustomerFile.Error <> UVE_NOERROR Then
   '... do some error handling'
End If
DateBox.Text = CustomerFile.Record
```

The **WriteNamedField** method does the converse, that is it takes a data value, applies an input conversion to it, and then writes it to the appropriate location in the record. It does not support I-descriptors.

Error Handling

Visual Basic does not have a direct equivalent to the THEN and ELSE clauses that a BASIC programmer uses to specify different actions depending on the success of an operation. Instead, all UniObjects have an **Error** property, which is set by every method. If the method completes successfully, the **Error** property is set to 0; any other value indicates an error. For a list of error codes, see Appendix A, "Error Codes and Replace Tokens."

For example, if you call the **Read** method of a **File** object the operation will fail if the record does not exist. In this case, the **Error** property of the **File** object is set to a value indicating `record not found.`

Once an object has set its **Error** property to a nonzero value, it will not process any method until the **Error** property has been examined. This means that you can code blocks of similar actions, for example, a sequence of calls to the **Read** method, and then handle errors for the whole block by looking at the error property at the end of the block. You do not need to examine the **Error** property after each call.

The **Error** property is set only for nonfatal errors; fatal errors raise an exception condition. In Visual Basic, you trap the exception condition with an **On Error** statement; if it is not trapped, the exception makes the program terminate. If you want your program also to raise exceptions when a nonfatal error occurs, you can set the **ExceptionOnError** property to **True**. For examples of error handling, see the entry for the **File** object.

Record Locks

***Note**: BASIC programmers should read this section carefully. Locking is handled differently in UniObjects to make coding easier in the event-driven environment of a client application.*

UniVerse and UniData have a system of locks to prevent potential problems when several users try to access the same data at the same time. The three types of locks you can use in programs are task locks, file locks, and record locks. This section discusses only record locks, which are used most often. For information on task locks and file locks, refer to the descriptions of the **SetTaskLock** and **ReleaseTaskLock** methods of the **Session** object, and to the **LockFile** and **UnlockFile** methods of the **File** object, in Chapter 3, "A Tour of the Objects." Also see *UniVerse System Description* for more information on UniVerse file locks, or *Using UniData* and *Administering UniData* for more information about UniData file locks.

A record lock prevents other users from:

- Setting a file lock on the file containing the locked record.
- Setting a record lock on the locked record.
- Writing to the locked record.
- Creating a record with the same ID. In this case you set a lock on the record before it has been created.

There are two types of record lock:

- Exclusive update locks (READU locks), which prevent other users from reading or writing to the record
- Shared read locks (READL locks), which allow other users to read the record, but not to update it

Setting and Releasing Locks

Setting and releasing record locks is controlled by three properties of the **File** or **Dictionary** object:

- **BlockingStrategy** specifies whether to wait if the record is already locked (equivalent to a BASIC LOCKED clause).
- **LockStrategy** specifies what kind of lock to set when reading.

- **ReleaseStrategy** specifies when a lock is released, for example:
    - When a record is written or deleted.
    - When you assign a new value to the **RecordId** property. This provides a simple way to set the lock release strategy for a program that edits a sequence of records, without having to code lock handling every time a record is read or written.
    - Only by the **UnlockRecord** method.

***Note**: All locks are released when the session is closed.*

You can set these properties for each file, or you can use the defaults associated with the **Session** object. These defaults are specified in the **DefaultBlockingStrategy**, **DefaultLockStrategy**, and **DefaultReleaseStrategy** properties of the **Session** object. In either case the properties remain set for all subsequent reads on that file during the session; you do not need to set them again. For examples, see the entries for the **File** and **Session** objects in Chapter 3, "A Tour of the Objects."

Select Lists

In UniVerse and UniData, you can retrieve a specified set of records and then save their record IDs as a select list. You can then either use the select list immediately in a program or command, or give it a name and save it for future use.UniVerse database sessions can have up to 11 select lists (numbered 0 through 10) active at the same time; UniData sessions can have up to 10 active select lists (numbered 0 through 9).

Accessing Select Lists

A Visual Basic program can use select lists by defining **SelectList** objects. You obtain a reference to one of the numbered select lists using the **SelectList** method of the **Session** object, for example:

```
Dim UVSelectList As Object
Set UVSelectList = UVSession.SelectList(0)
```

Creating Select Lists

The methods you can use to create a select list are **FormList**, **Select**, **SelectAlternateKey**, or **SelectMatchingAk**. You can also create a select list by executing a database command that creates one, for example, SELECT or SSELECT.

Reading and Clearing Select Lists

You can read a select list in two ways:

- One ID at a time using the **Next** method
- All at once using the **ReadList** method

If you just want to read part of a list, you can discard the unwanted part by calling the **ClearList** method.

For more information and examples, see the entry for the **SelectList** object.

Using a Dictionary

For most application programs it is economical to build a record's structure and field types into the program. This avoids having to look up the format of the record in the file dictionary. If you want your program to process different types of record, you need to look in the file dictionary to see how the records are structured. In a Visual Basic program you do this through the **OpenDictionary** method of the **Session** object. This returns a **Dictionary** object, which has properties for reading and writing individual fields from the dictionary. These properties are **ASSOC**, **CONV**, **FORMAT**, **LOC**, **NAME**, **SM**, **SQLTYPE**, and **TYPE**. For more information about these properties, see the entry for the **Dictionary** object on.

Here is an example that finds the type of a particular field:

```
FileDictPart.RecordID = NewField
FieldType = FileDictPart.TYPE
If FileDictPart.Error UVE_NOERROR Then
'... handle the error'
End If
```

Using Binary and Text Files

You can use operating system files to store text or binary data that you want to include in a program. UniVerse handles an operating system *directory* as a type 1 or type 19 UniVerse file. Operating system *files* are handled as records in the database file. The name of the file is its record ID. For small text files, you can open the type 1 file with the **OpenFile** method, and then read an entire text file with the **Read** method. See "Using Files" on page 2-11.

On UniData systems you can read and write operating system files as records to an existing operating system directory, but you cannot process such files sequentially (see the next section).

Accessing Files Sequentially

If a file is large or contains binary data, it is better to read and write the file sequentially, that is, in manageable sections. You can do this by using the **OpenSequential** method of the **Session** object. This returns a **SequentialFile** object, whose methods allow sequential access to the data. The **SequentialFile** object uses an internal file pointer to track read and write operations (equivalent to BASIC's sequential file variable). You can:

- Read and write lines of text with the **ReadLine** and **WriteLine** methods
- Read and write binary data with the **ReadBlk** and **WriteBlk** methods
- Change the position of the file pointer with the **FileSeek** method
- Truncate an existing file with the **WriteEOF** method

For more information, see the entry for the **SequentialFile** object.

Using Database Commands

You can run most database commands from a Visual Basic program through the **Command** object, which is equivalent to the BASIC EXECUTE statement.

The **Command** object can be used for:

- Creating or deleting a database file.
- Making a select list of records that meet your requirements. (See "When to Use Database Commands" on page 2-25.)
- Running a program on the server to save processing power on the client.

***Note**: The **Command** object may not always be the most efficient way to use resources in a client/server program. For more information, see "When to Use Database Commands" on page 2-25.*

You can issue only one command at a time. You obtain a reference to the **Command** object from the **Command** property of the **Session** object. For example:

```
Dim UVCommand As Object
Set UVCommand = UVSession.Command
```

You specify the command that you want to execute by setting the **Text** property, and then execute it by calling the **Exec** method. For example:

```
UVCommand.Text = "some command"
UVCommand.Exec
```

The result of a command is held in the **CommandStatus**, **Error**, and **Response** properties as follows:

- If the command ran to completion, **CommandStatus** is set to UVS_COMPLETE and any output generated by the command is held in the **Response** property.
- If the command did not complete, or if all the output was not retrieved, the **CommandStatus** property shows what happened. You can use the **Reply** or **NextBlock** method to continue processing. For an example, see the entry for the **Command** object.

Client/Server Design Considerations

When you design your application, you should avoid unnecessary interaction between the client and the server. This has two main benefits:

- Performance: reducing network traffic improves performance.
- Scalability: if more clients and servers are added to the network your application's performance will still be acceptable.

To use the client and server efficiently, you must know which operations need to communicate with the server, and when those operations take place. If necessary, you can then change the design of the application to reduce the interaction with the server. The following sections describe some ideas for using the client and server economically.

Calling Server Subroutines

You can reduce network traffic by running parts of your application on the server as BASIC subroutines. Server subroutines run in an area called catalog space that is available to any program on the server.

***Note**: A server subroutine must be cataloged before you can call it from UniObjects. This must be done on the server. For more information about cataloging UniVerse subroutines, see the entry for the CATALOG command in UniVerse User Reference, and the discussion of subroutines in UniVerse BASIC. For more information about cataloging UniData subroutines, see UniData Commands Reference and Administering UniData.*

You can call a cataloged subroutine from Visual Basic through the **Subroutine** object, which you obtain through the **Subroutine** method of the **Session** object. For example:

```
Dim GetOrderData As Object
Set GetOrderData = UVSession.Subroutine("*GET.ORDER.DATA", 4)
```

You must supply the name of the cataloged subroutine and the number of arguments that it takes to the **Subroutine** method. Once your program has obtained the **Subroutine** object, you use the **SetArg** method to supply values for arguments, the **Call** method to call the subroutine, and the **GetArg** method to retrieve any argument values returned. For example:

```
GetOrderData.SetArg 0, OrderNumber
GetOrderData.SetArg 1, DisplayType
GetOrderData.Call
OrderData = GetOrderData.GetArg (2)
ErrorCode = GetOrderData.GetArg (3)
```

When to Use Database Commands

You can save client processing by executing database commands on the server. The most effective commands to use are those that do not generate any output, such as SELECT.

Some commands may increase network traffic because they generate prompts or messages that your program must then handle. If your program cannot cope with an unexpected request for input from a command, it will hang with no indication of what went wrong. In particular, you should avoid using interactive commands such as CREATE.FILE or REFORMAT which have many possible prompts and error conditions. (In most cases it should not be necessary to create or reformat files as part of your application.)

***Note**: You cannot cancel a command that is executed through UniObjects.*

Task Locks

You can protect a process running on the server from interruption by other users or programs by setting a task lock. UniVerse and UniData have 64 task locks that you can assign to events or processes. For example, if your application uses a resource such as a printer, you can set a task lock to prevent another database user from accessing the printer during your print run.

You set and release task locks with the **SetTaskLock** and **ReleaseTaskLock** methods of the **Session** object. The task locks have no predefined meanings. You must ensure that your application sets and releases task locks efficiently. You can use the LIST.LOCKS command to check which locks are in use, and which users hold them.

Chapter

A Tour of the Objects

3

This chapter describes the objects used in UniObjects, together with their associated methods and properties, in the order in which you are most likely to use them in an application.

Object	Description
Session Object	The **Session** object is the starting point for all applications, and is used to access the other objects.
File Object and Dictionary Object	Next, your program is likely to access a database file on the server through the **File** or **Dictionary** object.
SequentialFile Object	If you want to use data in an operating system file, you use the **SequentialFile** object.
DynamicArray Object	You can then address records in database files through the **DynamicArray** object. This object may also be used independently of a session.
SelectList Object	You read and manipulate select lists of records through the **SelectList** object.
Command Object	You can execute a database command through the **Command** object.
Subroutine Object	You can execute a cataloged BASIC subroutine on the server through the **Subroutine** object.
Transaction Object	You can create and manipulate transactions through the **Transaction** object.
NLSLocale Object (UniVerse Only)	You can define which locale setting to use through the **NLSLocale** object.
NLSMap Object (UniVerse Only)	You can determine which maps to use through the **NLSMap** object.

Code Examples

The following objects contain a short program example that illustrates many of the methods and properties associated with the object: **Session**, **File**, **Dictionary**, **SequentialFile**, **DynamicArray**, **SelectList**, **Command**, and **Subroutine**. If you want to include the examples in your programs, files containing the code are located in the directory called UNIDK\SAMPLES\MANUAL, and are included in the project SAMP1.

Note*: The sample programs are in Visual Basic 4.0 format.*

Replace Tokens

Some methods and properties use replace tokens to represent global constants, integer values, and error codes. You can include a file containing these tokens with your application. For more details about these tokens, see Appendix A, “Error Codes and Replace Tokens.”

Boolean Values

In the following pages the Boolean values used are those defined for Visual Basic. That is, a value of 0 indicates **False**, and a value of –1 indicates **True**.

Case-Sensitivity

The names of the objects, methods, and properties used in UniObjects are not case-sensitive.

Account Flavors

UniObjects works best with IDEAL flavor UniVerse accounts. With other UniVerse account flavors, status or error codes returned by some methods may vary from those documented.

UniObjects works best with UniData accounts when ECLTYPE and BASICTYPE are both set to U.

Quick Reference

The following table gives a quick reference to the methods and properties that are available with each object.

Objects	Methods	Properties
Session	Connect	AccountPath
	Disconnect	Command
	DynamicArray	ConnectionString
	GetAtVariable	DatabaseType
	Iconv	DefaultBlockingStrategy
	IsActive	DefaultLockStrategy
	Oconv	DefaultReleaseStrategy
	OpenDictionary	Error (read-only)
	OpenFile	ExceptionOnError
	OpenSequential	FM (read-only)
	ReleaseTaskLock	HelpFile (read-only)
	SelectList	HostName
	SetAtVariable	HostType (read-only)
	SetTaskLock	Identifier (read-only)
	Subroutine	IM (read-only)
		NLSLocale (read-only)
		NLSMap (read-only)
		Password
		ShowConnectDialog
		SQLNULL (read-only)
		Status (read-only)
		Subkey
		SVM (read-only)
		Timeout
		TM (read-only)
		Transaction (read-only)
		Transport
		UserName
		VM (read-only)

Objects and Their Methods and Properties

Objects	Methods	Properties
File	ClearFile	BlockingStrategy
	CloseFile	Error (read-only)
	DeleteRecord	ExceptionOnError
	GetAkInfo	FileName (read-only)
	IsOpen	FileType (read-only)
	IType	Identifier (read-only)
	LockFile	LockStrategy
	LockRecord	Record
	Read	RecordId
	ReadField	ReleaseStrategy
	ReadFields	Status (read-only)
	ReadNamedField	
	UnlockFile	
	UnlockRecord	
	Write	
	WriteField	
	WriteFields	
	WriteNamedField	
Dictionary	ClearFile	ASSOC
	CloseFile	BlockingStrategy
	DeleteRecord	CONV
	GetAkInfo	Error (read-only)
	IsOpen	ExceptionOnError
	LockFile	FileName (read-only)
	LockRecord	FileType (read-only)
	Read	FORMAT
	ReadField	Identifier (read-only)
	ReadNamedField	LOC
	UnlockFile	LockStrategy
	UnlockRecord	NAME
	Write	Record
	WriteField	RecordId
	WriteNamedField	ReleaseStrategy
		SM
		SQLTYPE
		Status (read-only)
		TYPE

Objects and Their Methods and Properties (Continued)

Objects	Methods	Properties
SequentialFile	CloseSeqFile	Error (read-only)
	FileSeek	ExceptionOnError
	IsOpen	FileName (read-only)
	ReadBlk	Identifier (read-only)
	ReadLine	ReadSize
	WriteBlk	RecordId (read-only)
	WriteEOF	Status (read-only)
	WriteLine	Timeout
DynamicArray	Count	Error (read-only)
	Del	ExceptionOnError
	Field	StringValue (default)
	Ins	TextValue
	Length	
	Replace	
	SubValue	
	Value	
SelectList	ClearList	Error (read-only)
	FormList	ExceptionOnError
	GetList	Identifier (read-only)
	Next	LastRecordRead (read-only)
	ReadList	
	SaveList	
	Select	
	SelectAlternateKey	
	SelectMatchingAk	
Command	Cancel	AtSelected (read-only)
	Exec	BlockSize
	NextBlock	CommandStatus (read-only)
	Reply	Error (read-only)
		ExceptionOnError
		Response (read-only)
		SystemReturnCode (read-only)
		Text (default)
Subroutine	Call	Error (read-only)
	GetArg	ExceptionOnError
	ResetArgs	RoutineName (read-only)
	SetArg	

Objects and Their Methods and Properties (Continued)

Objects	Methods	Properties
Transaction	Commit IsActive Rollback Start	Error (read-only) ExceptionOnError Level (read-only)
NLSLocale (UniVerse only)	SetName	ClientNames (read-only) Error (read-only) ExceptionOnError ServerNames (read-only)
NLSMap (UniVerse only)	SetName	ClientName (read-only) Error (read-only) ExceptionOnError ServerName (read-only)

Objects and Their Methods and Properties (Continued)

BASIC and InterCall Equivalents

The following table shows the UniObjects methods and their equivalents in BASIC and InterCall.

Method	BASIC Equivalent	InterCall Equivalent
Call	CALL	ic_subcall
Cancel	No direct equivalent	No direct equivalent
ClearFile	CLEARFILE	ic_clearfile
ClearList	CLEARSELECT	ic_clearselect
CloseFile	CLOSE, reassignment to file variable	ic_close
CloseSeqFile	CLOSESEQ	ic_closeseq
Commit	COMMIT	ic_trans
Connect	No direct equivalent	ic_opensession
Count	DCOUNT( )	No direct equivalent
Del	DEL	ic_strdel
DeleteRecord	DELETE, DELETEU	ic_delete
Disconnect	No direct equivalent	ic_quit
Exec	EXECUTE	ic_execute
Field	No direct equivalent	No direct equivalent
FileSeek	SEEK	ic_seek
FormList	FORMLIST	ic_formlist
GetAkInfo	INDICES( )	ic_indices
GetArg	No direct equivalent	No direct equivalent
GetAtVariable	No direct equivalent	ic_getvalue

UniObjects Methods and Their Equivalents

Method	BASIC Equivalent	InterCall Equivalent
GetList	No direct equivalent	ic_getlist
Iconv	ICONV( )	ic_iconv
Ins	INS	ic_insert
IsActive	No direct equivalent	No direct equivalent
IsOpen	No direct equivalent	No direct equivalent
IType	ITYPE( )	ic_itype
Length	No direct equivalent	No direct equivalent
LockFile	FILELOCK	ic_filelock
LockRecord	RECORDLOCKL, RECORDLOCKU	ic_recordlock
Next	READNEXT	ic_readnext
NextBlock	No direct equivalent	No direct equivalent
Oconv	OCONV( )	ic_oconv
OpenDictionary	OPEN DICT	ic_open
OpenFile	OPEN	ic_open
OpenSequential	OPENSEQ	ic_openseq
Read	READ, READL, READU	ic_read
ReadBlk	READBLK	ic_readblk
ReadField	READV, READVL, READVU	ic_readv
ReadLine	READSEQ	ic_readseq
ReadList	READLIST	No direct equivalent
ReadNamedField	No direct equivalent, see Appendix C, "Data Conversion Functions."	No direct equivalent
ReleaseTaskLock	UNLOCK	ic_unlock

UniObjects Methods and Their Equivalents (Continued)

Method	BASIC Equivalent	InterCall Equivalent
Replace	REPLACE( )	ic_replace
Reply	No direct equivalent	ic_inputreply
ResetArgs	No direct equivalent	No direct equivalent
Rollback	ROLLBACK	ic_trans
SaveList	No direct equivalent	No direct equivalent
Select	SELECT	ic_select
SelectAlternateKey	SELECTINDEX	ic_selectindex
SelectList	No direct equivalent	ic_select
SelectMatchingAk	SELECTINDEX	ic_selectindex
SetArg	No direct equivalent	ic_setvalue
SetAtVariable	No direct equivalent	ic_setvalue
SetName	No direct equivalent	ic_set_locale, ic_set_map
SetTaskLock	LOCK	ic_lock
Start	BEGIN TRANSACTION	ic_trans
Subroutine	No direct equivalent	No direct equivalent
SubValue	No direct equivalent	No direct equivalent
UnlockFile	FILEUNLOCK	ic_fileunlock
UnlockRecord	RELEASE	ic_release
Value	No direct equivalent	No direct equivalent
Write	WRITE, WRITEU	ic_write
WriteBlk	WRITEBLK	ic_writeblk
WriteEOF	WEOFSEQ	ic_weofseq

UniObjects Methods and Their Equivalents (Continued)

Method	BASIC Equivalent	InterCall Equivalent
WriteField	WRITEV, WRITEVU	ic_writev
WriteLine	WRITESEQ	ic_writeseq
WriteNamedField	No direct equivalent, see Appendix C, "Data Conversion Functions."	No direct equivalent

UniObjects Methods and Their Equivalents (Continued)

Session Object

The **Session** object defines and manages a session with the server database. The **Session** object is a custom control that is invisible when you run your application. You access other objects on the server, such as **File** or **Dictionary** objects through the **Session** object. See Opening a Database Session and Accessing Objects in Chapter 2, " Using UniObjects," for information about using the **Session** object. For a program example, see "Example Using the Session Object" on page 3-54.

The methods and properties you can use with the **Session** object are described in the following sections.

Session Object Methods

These are the methods that you can use with the **Session** object:

? Connect	? Iconv	? OpenFile
? Disconnect	? IsActive	? OpenSequential
? Dynamic Array	? Oconv	? ReleaseTaskLock
? GetAtVariable	? OpenDictionary	? Select List
? SetAtVariable		

Session Object Properties

These are the properties of the **Session** object:

? AccountPath

? Command

? ConnectionString

? DatabaseType

? DefaultBlockingStrategy

? DefaultLockStrategy

? DefaultReleaseStrategy

? Error (read-only)

? ExceptionOnError

? FM (read-only)

? HelpFile (read-only)

? HostName

? HostType (read-only)

? Identifier (read-only)

? IM (read-only)

? NLSLocale (read-only)

? NLSMap (read-only)

? Password

? ShowConnectDialog

? SQLNULL (read-only)

? Status (read-only)

? Subkey

? SVM (read-only)

? Timeout

? TM (read-only)

? Transaction (read-only)

? Transport

? UserName

? VM (read-only)

Connect Method

Syntax

Bool = *SessObj*.**Connect**

Description

This method opens a session on the server that is identified by the **HostName** property. It uses the values set up in the **HostName** and **AccountPath** properties to create a connection. If either **HostName** or **AccountPath** do not have an assigned value, a dialog box prompts the user for the value at run time.

Bool is the returned integer that indicates whether or not the session was opened. *Bool* has the following values:

- **True** indicates the connection was established and the **HostName** and **AccountPath** properties were set.
- **False** indicates the connection was not established or the user cancelled the operation in the dialog box.

SessObj is an inactive **Session** object. It must either exist as a control (in Visual Basic Version 4.0) or be created with the following statement:

```
Set SessObj = CreateObject(UV_SESSION_OBJECT)
```

UV_SESSION_OBJECT is a replace token for the session. For more information about replace tokens, see Appendix A, "Error Codes and Replace Tokens."

If the **Connect** method returns **False**, check the **Error** property for possible causes. If you cancel the Connect Details dialog box, no error occurs, and the session remains inactive. If a connection is established, the **IsActive** method returns **True**. For an example of error handling during the connection process, see "Example Using the Session Object" on page 3-54.

This method corresponds to the InterCall **ic_opensession** function.

Example

```
'open a session to the chosen server
   SessObj.AccountPath = "d:\uv\accounts\mkt"
   SessObj.HostName = "mktg_nt"
   Success = SessObj.Connect
```

Disconnect Method

Syntax

SessObj.**Disconnect**

Description

This method closes the object's active session, closes any open files and releases any locks associated with the session.

SessObj is a **Session** object.

After calling this method, the **IsActive** method is set to **False**, and any operation performed on this session, other than **Connect** or **Disconnect**, results in an error, and returns an object variable whose value is **Nothing**.

***Note**: Other objects that were created by the session, or are associated with the session are still available, but using them may cause an error. For example, if you have a* **File** *object created by the* **Session** *object, you can access the last record that was read from the file, but you cannot read another record.*

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_quit** function.

Example

```
If SessObj.IsActive = True Then
    SessObj.Disconnect
End If
```

DynamicArray Method

Syntax

DynArrayObj = *SessObj*.**DynamicArray**

Description

This method creates a **DynamicArray** object that inherits the current values of all delimiters from the **Session** object.

DynArrayObj is a **DynamicArray** object.

SessObj is a **Session** object.

The new dynamic array inherits the current values of all delimiters from the **Session** object. The values of the delimiters for these dynamic arrays do not change if the NLS map used by the session changes to one that uses delimiters with other values.

GetAtVariable Method

Syntax

String = *SessObj*.**GetAtVariable**(*@variableID*)

Description

This method returns the value of a BASIC @variable as a string.

String is the returned value of the @variable.

SessObj is a **Session** object.

@variableID identifies the @variable whose value is to be retrieved. *@variableID* should be one of the following values or tokens:

Value	Token	BASIC @variable
1	AT_LOGNAME	@LOGNAME
2	AT_PATH	@PATH
3	AT_USERNO	@USERNO
4	AT_WHO	@WHO
5	AT_TRANSACTION	@TRANSACTION
6	AT_DATA_PENDING	@DATA.PENDING
7	AT_USER_RETURN_CODE	@USER.RETURN.CODE
8	AT_SYSTEM_RETURN_CODE	@SYSTEM.RETURN.CODE
9	AT_NULL_STR	@NULL.STR
10	AT_SCHEMA	@SCHEMA

@variableID Values or Tokens

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_getvalue** function.

Example

```
MsgBox "Operating in account" & SessObj.GetAtVariable(4)
```

Iconv Method

Syntax

String = *SessObj*.**Iconv**(*InputString*, *ConvCode*)

Description

This method converts an input string to an internal storage format using the conversion code specified.

String is the converted string.

SessObj is a **Session** object.

InputString is the string to be converted.

ConvCode is a conversion code that defines the conversion to be performed. You can use any of the conversion codes that are available with the BASIC ICONV function.

The **Iconv** method sets the **Session** object's **Status** property to one of the following values:

Value	Meaning
0	The conversion was successful.
1	The string supplied was invalid.
2	The conversion code supplied was invalid.
3	Successful conversion of possibly invalid data.

Status Values

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_iconv** function and the BASIC ICONV function.

Example

```
DayCount = SessObj.Iconv(txtDate.Text, "D2/")
```

IsActive Method

Syntax

Bool = *SessObj*.**IsActive**

Description

This method returns **True** if the session has been successfully established by calling the **Connect** method; otherwise it is **False**.

Bool is set to **True** or **False** to indicate whether the session is active.

SessObj is a **Session** object.

***Note**: The* **IsActive** *method does not change the setting of the* **Error** *property, and is not affected by the state of the* **Error** *property.*

Example

```
If SessObj.IsActive = True Then
   SessObj.Disconnect
```

Oconv Method

Syntax

String = *SessObj*.**Oconv**(*InputString*, *ConvCode*)

Description

This method converts a string from the internal storage format to an external format defined by a conversion code.

String is the converted string.

SessObj is a **Session** object.

InputString is the string to be converted.

ConvCode is a conversion code that defines the conversion to be performed. You may use any of the conversion codes that are available with the BASIC ICONV function.

The **Error** property is set if an error occurs.

The **Oconv** method sets the **Session** object's **Status** property to one of the following values:

Value	Meaning
0	The conversion was successful.
1	The string supplied was invalid.
2	The conversion code supplied was invalid.
3	Successful conversion of possibly invalid data.

This method corresponds to the InterCall **ic_oconv** function and the BASIC OCONV function.

Example

```
txtDate.Text = SessObj.Oconv(Project(i).StartDate, "D2/")
```

OpenDictionary Method

Syntax

Set *DictObj* = *SessObj*.**OpenDictionary**(*Filename*)

Set *DictObj* = *SessObj*.**OpenDictionary**(*FileObject*)

Description

This method opens a **Dictionary** object that accesses a file dictionary.

DictObj is the returned **Dictionary** object.

SessObj is a **Session** object.

Filename is a the name of the database file whose dictionary is to be opened.

FileObject is the **File** or **Dictionary** object for a file or dictionary that has already been opened.

If *Filename* or *FileObject* corresponds to a file dictionary, the file DICT.DICT is opened.

If an error occurs, *DictObj* is set to **Nothing** and the **Session** object's **Error** property indicates the error.

When the method completes successfully, the **Status** property contains the file type of the dictionary. See the **FileType** property of the **File** object for a list of file types.

This method corresponds to the InterCall **ic_open** function and the BASIC OPEN DICT statement.

Examples

```
Set DictObj = SessObj.OpenDictionary('CUSTOMERS')
Set DictObj = SessObj.OpenDictionary(FileObj)
```

OpenFile Method

Syntax

Set *FileObj* = *SessObj*.**OpenFile**(*FileName*)

Description

This method opens an existing database file and returns an object that allows access to the file.

FileObj is the returned **File** or **Dictionary** object representing an open database file.

SessObj is a **Session** object.

FileName is the name of the database file to be opened.

If an error occurs, *FileObj* is set to **Nothing** and the **Session** object's **Error** property indicates the error. If no error occurs, the **Status** property is set to the file type of the opened file. See the **FileType** property of the **File** object for a list of file types.

This method corresponds to the InterCall **ic_open** function and the BASIC OPEN statement.

Example

```
    Set FileObj = SessObj.OpenFile("PRODUCTS")
```

OpenSequential Method

Syntax

Set *SeqFileObj* = *SessObj*.**OpenSequential**(*FileName, RecordId, CreateFlag*)

Description

This method returns a **SequentialFile** object.

SeqFileObj is the returned **SequentialFile** object.

SessObj is a **Session** object.

FileName is the name of a type 1 or 19 file.

RecordId refers to a record within the file, optionally created if it does not exist.

CreateFlag is a Boolean value. Setting *CreateFlag* to **True** creates the file if it does not already exist. Set this flag to **True** only when the next operation on the file is a **WriteLine** method or **WriteBlk** method.

If the file cannot be opened, *SeqFileObj* is set to **Nothing** and the **Error** property indicates the error or status. If the file cannot be opened owing to an error on the server, the **Session** object's **Status** property displays one of the following values:

Value	Meaning
0	No record ID was found.
1	The specified file is not type 1 or type 19.
2	The specified file was not found.

Status Values

This method corresponds to the database CREATE command (if the create flag is set to **True**), the InterCall **ic_openseq** function, and the BASIC OPENSEQ statement.

Example

```
Set SeqFileObj = SessObj.OpenSequential("TESTS", "TEST1", False)
```

ReleaseTaskLock Method

Syntax

SessObj.**ReleaseTaskLock** *TaskLockNumber*

Description

This method releases one of the 64 UniVerse task locks. For more information about task locks, see Task Locks in Chapter 2, "Using UniObjects."

SessObj is a **Session** object.

TaskLockNumber is the number of the task lock to be released.

The **Error** property is set if an error occurs.

Example

```
    SessObj.ReleaseTaskLock 4
```

SelectList Method

Syntax

Set *SelectListObj* = *SessObj*.**SelectList**(*ListNumber*)

Description

This method returns a **SelectList** object representing one of the 11 select lists.

SelectListObj is the returned **SelectList** object representing a select list.

SessObj is a **Session** object.

ListNumber is the number, 0 through 10, of the select list to use.

If an error occurs, *SelectListObj* is set to **Nothing** and the **Session** object's **Error** property indicates the error.

Example

```
'open a select list to find a primary key in the file
   Set SelectListObj = SessObj.SelectList(0)
   SelectListObj.Select FileObj
```

SetAtVariable Method

Syntax

SessObj.**SetAtVariable** *@variableId, String*

Description

This method sets the value of a BASIC @variable to a string.

SessObj is a **Session** object.

String is the value to which the @variable is to be set.

@variableId is the @variable name to be set. *@variableId* is:

7	AT_USER_RETURN_CODE	@USER.RETURN.CODE

@variableID Value

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_setvalue** function.

Example

```
SessObj = SetAtVariable AT_USER_RETURN_CODE, -2
```

SetTaskLock Method

Syntax

SessObj.**SetTaskLock** *TaskLockNumber*

Description

This method locks one of the 64 task locks in UniVerse. For more information about task locks, see Task Locks in Chapter 2, "Using UniObjects."

SessObj is a **Session** object.

TaskLockNumber is the number of the task lock to be set.

The **Error** property is set if an error occurs.

This method is equivalent to the BASIC LOCK statement and corresponds to the InterCall **ic_lock** function.

Example

```
SessObj.SetTaskLock 5
```

Subroutine Method

Syntax

Set *SubrObj* = *SessObj*.**Subroutine**(*ServerSubroutine*, *ArgCount*)

Description

This method returns a **Subroutine** object that calls a cataloged subroutine on the server.

SubrObj is the returned **Subroutine** object representing a BASIC cataloged subroutine.

SessObj is a **Session** object.

ServerSubroutine is the name of the subroutine which *SubrObj* represents. This should be the name used when the subroutine was cataloged on the server.

ArgCount is the number of arguments that the server subroutine uses.

If an error occurs, *SubrObj* is set to **Nothing** and the **Session** object's **Error** property indicates the error.

Example

```
Set SubrObj = SessObj.Subroutine("CALCDISCOUNT", 3)
```

Session Object Properties

AccountPath

This property contains the name of the account to which the session is to be connected on the database server. You can specify the account as:

- A full path, for example, on a Windows server:

  d:\uv\sales\customer

  Or on a UNIX server:

  /usr/uv/sales/CUSTOMER
- A valid account name, as specified in the ACCOUNTS file on the server.

You can set this property at design time or run time. If you do not set it before calling the **Connect** method, you are prompted for this value in a dialog box. See also "Connect Method" on page 3-25 and "HostName" on page 3-49.

Command

This property contains the single **Command** object for the session. It is set to **Nothing** when the session is not active. This property cannot be modified. See "Command Object" on page 3-136.

ConnectionString

Use this property to specify the name of a server process to connect to when you want to connect to a server other than the one specified by the DatabaseType property.

DatabaseType

Use this property to specify the database type to which you want to connect. This property has one of the following values:

Value	Token	Meaning
0	DEFAULT	Opens a session on either a UniVerse or a UniData system. The application connects to a server process called *defcs*, which is defined in the *unirpcservices* file.
1	UNIVERSE	Opens a session on a UniVerse system, connecting to a server process called *uvcs*.
2	UNIDATA	Opens a session on a UniData system, connecting to a server process called *udcs*.

Database Type Values

DefaultBlockingStrategy

Use this property to set the value of the **BlockingStrategy** property for all **File** and **Dictionary** objects created by the **OpenFile** and **OpenDictionary** methods.

This property has one of the following values:

Value	Token	Meaning
1	WAIT_ON_LOCKED	If the record is locked, wait until it is released (see Note).
2	RETURN_ON_LOCKED	Return a value to the **Status** property to indicate the state of the lock. This is the default.

DefaultBlockingStrategy Values

Altering this property does not affect **File** or **Dictionary** objects that have already been created.

See also “DefaultLockStrategy” on page 3-47 and “DefaultReleaseStrategy” on page 3-47.

DefaultLockStrategy

Use this property to set the value of the **LockStrategy** property for all **File** and **Dictionary** objects created by **OpenFile** and **OpenDictionary** methods.

This property has one of the following values:

Value	Token	Meaning
0	NO_LOCKS	No locking. This is the default.
1	EXCLUSIVE_UPDATE	Sets an exclusive update lock (READU).
2	SHARED_READ	Sets a shared read lock (READL).

DefaultLockStrategy Values

Altering this property does not affect files or dictionaries that are already opened.

DefaultReleaseStrategy

Use this property to set the value of the **ReleaseStrategy** property for all **File** and **Dictionary** objects created by the **OpenFile** and **OpenDictionary** methods. Whenever the **RecordId** property is changed or explicitly released, the property reverts to the initial value. Altering this property does not affect files or dictionaries that are already opened.

DefaultReleaseStrategy can have one of the following values:

Value	Token	Meaning
1	WRITE_RELEASE	Releases the lock when the record is written. This is the property's initial value.
2	READ_RELEASE	Releases the lock when the record is read.
4	EXPLICIT_RELEASE	Maintains locks as specified by the **LockStrategy** property. Locks can be released only with the **UnlockRecord** method.
8	CHANGE_RELEASE	Releases the lock whenever a new value is assigned to the **RecordId** property.

DefaultReleaseStrategy Values

All the values are additive. If you specify EXPLICIT_RELEASE with WRITE_RELEASE and READ_RELEASE, it takes a lower priority. The initial value of **DefaultReleaseStrategy** is set to 12, that is, release locks when the value of the **RecordId** property changes or when locks are released explicitly.

Error

This read-only property contains a code for the last error that occurred. If no error has occurred, it contains 0. For a list of error codes and their meanings, see Appendix A, "Error Codes and Replace Tokens." For more information about error conditions, see Error Handling. in Chapter 2, "Using UniObjects."

ExceptionOnError

Use this property to generate an exception that can be handled by the Visual Basic **On Error** statement. Set this property to **True** to force all errors to generate an exception. The default is **False**, causing only fatal errors to generate an exception.

All other objects created as a result of the **Session** object inherit the setting of this property as their initial value. Any objects that were created before the session are not affected. If you change the setting while a session is active, it does not modify the setting of the **ExceptionOnError** property for any other object.

FM

This read-only property contains the current character value of the field mark used on the server. If NLS is not enabled, FM contains a value of 254. If NLS is enabled, FM gets its value from the server.

This property corresponds to the InterCall **ic_get_mark_value** function.

HelpFile

This read-only property contains the full name of the help file for UniObjects, including the drive letter and directory path.

The full help file name is derived from information in the registry regarding the location of the installed product, and product knowledge of the help file name.

HostName

This property contains the name of the database server.

You can set this property at design time or run time. If you do not set the value before calling **Connect**, the **Connect** method prompts the user for the value so that it can establish a connection to the server, then writes the value entered to the **HostName** property.

If you used the **Transport** property to specify a TCP/IP connection, you can use the **HostName** property to specify the IP address and/or port number to use for the connection. For example:

- If you enter *server name* (for example, server1), a TCP/IP connection is made to that node name.
- If you enter *server name:port number* (for example, server1:396), a TCP/IP connection is made to the specified port on the server.
- If you enter *IP address* (for example, 192.34.56.94), a TCP/IP connection is made to the specified address.
- If you enter *IP address:port number* (for example, 192.34.56.94:396), a TCP/IP connection is made to the specified port number at the given IP address.

HostType

This read-only property contains a value that represents the type of host the session is connected to. This property can have only one of the following values:

Value	Token	Meaning
0	UVT_NONE	The host system cannot be determined, the session is not connected.
1	UVT_UNIX	The host is a UNIX system.
2	UVT_NT	The host is a Windows system.

HostType Values

Identifier

This read-only property contains the **Session** object's InterCall session ID. It is used only in applications that call an InterCall function that requires a session identifier.

IM

This read-only property contains the current character value of the item mark used on the server. If NLS is not enabled, IM contains a value of 255. If NLS is enabled, IM gets its value from the server.

This property corresponds to the InterCall **ic_get_mark_value** function.

NLSLocale

This read-only property contains the **NLSLocale** object for this session. If NLS is not enabled, this property contains a NULL reference.

This property corresponds to the InterCall **ic_session_info** function.

NLSMap

This read-only property contains the **NLSMap** object for this session. If NLS is not enabled, this property contains a NULL reference.

This property corresponds to the InterCall **ic_session_info** function.

Password

This property contains a password (if required) for the user specified in the **UserName** property. On Windows servers, this property is ignored, but should be specified as an empty string if you want your code to be portable to both Windows and UNIX systems.

ShowConnectDialog

This property determines whether the **Session** object presents a dialog box when it does not have enough information to make a connection.

The default value of the property is **True**. This maintains backward compatibility.

SQLNULL

This read-only property contains the current character value for the null value that is used on the server. If NLS is not enabled, SQLNULL contains a value of 128. If NLS is enabled, the SQLNULL gets its value from the server.

This property corresponds to the InterCall **ic_get_mark_value** function.

Status

This read-only property contains a status code returned by certain methods. Refer to each method for a description of the status values that are returned.

Subkey

This property contains the device subkey string used when an application connects to a database server through a multiple-tier connection.

SVM

This read-only property contains the current character value of the subvalue mark used on the server. If NLS is not enabled, SVM contains a value of 252. If NLS is enabled, SVM gets its value from the server.

This property corresponds to the InterCall **ic_get_mark_value** function.

Timeout

This property specifies the length of the timeout for a connected session. The timeout period is used by the UniVerse remote procedure call utility (UniRPC).

The default value of this property is 0 (no timeout period). For values greater than 0, you must specify seconds.

***Note**: If you enter a value that is too small, a running process (for example, the* **Read** *method) may time out. If this occurs, an error code is returned and the connection to the server is dropped.*

TM

This read-only property contains the current character value of the text mark used on the server. If NLS is not enabled, TM contains a value of 251. If NLS is enabled, TM gets its value from the server.

This property corresponds to the InterCall **ic_get_mark_value** function.

Transaction

This read-only property contains the **Transaction** object for this session.

Transport

This property specifies the transport type to use when a connection is made to a server. This property can have one of the following values:

Value	Token	Meaning
0	NETWORK_DEFAULT	There is no preferred transport type. The default network is used.
1	NETWORK_LANMAN	The preferred transport is LAN Manager Named Pipes.
2	NETWORK_TCP	The preferred transport type is TCP/IP.

Transport Values

***Note**: If you make a connection using TCP/IP, you must enter security information to connect to the server, for example, user name and password.*

Once a session is connected, setting this property has no effect on the transport type used. To use a new transport type, disconnect this session and make a new session connection.

UserName

This property contains the user name to be used to log in to a database server. On Windows servers, this property is ignored, but should be specified as an empty string if you want your code to be portable to both Windows and UNIX.

VM

This read-only property contains the current character value of the value mark used on the server. If NLS is not enabled,VM contains a value of 253. If NLS is enabled, VM gets its value from the server.

This property corresponds to the InterCall **ic_get_mark_value** function.

Example Using the Session Object

This example creates a session, handles errors and closes the session.

```
 Dim objSession As object ' The Session to the database

   Dim objFile As object ' The file to open (VOC)

   Const UVE_NOERROR = 0       ' From UVOAIF.TXT - no error
   Const NETWORK_TCP = 2' From UVOAIF.TXT
   Const NETWORK_LANMAN = 1' From UVOAIF.TXT
   Const NETWORK_DEFAULT = 0' No preference
   ' The registered name of a database Session - Version 1
   Const UV_SESSION_OBJECT = "UniObjects.unioaifctrl"
   '
   ' Create a Session object to work with
   '  - This is a contrived sample, in a full application the
Session object
   '  - would typically be a Global variable that is set once
maybe in
   '  - response to a menu selection (e.g. Connect) on the main
form.
   '
   Set objSession = CreateObject(UV_SESSION_OBJECT)
   If objSession Is Nothing Then
      ' NB. Errors will be reported by Visual Basic
      Exit Sub ' End the program
   End If
   objSession.UserName = Input.Box ("User Name:","Login")
   objSession.Password = Input.Box ("Password:","Password")
   objSession.Transport = NETWORK_LANMAN

   '
   ' Establish a connection to the database server. By default it
displays
   ' a dialog box to request the HostName and AccountPath property
values.
   '
   objSession.Connect
   If objSession.IsActive Then
      '
      ' Continue with the program, then close the session...
      '
      If objSession.HostType = UVT_UNIX Then
         MsgBox "You are connected to a UNIX server"
      End If
         objSession.Disconnect
   Else
      '
      ' Check for Session errors - display message box with error
code
      ' No error means the user cancelled the connection dialog
```

```
box
      '
      If objSession.Error <> UVE_NOERROR Then
         MsgBox "Unable to open connection:- " & objSession.Error
      End If
   End If
End Sub
```

File Object

The **File** object defines and manages a data file on the server. You define the **File** object through the **OpenFile** method of the **Session** object. See Using Files in Chapter 2, " Using UniObjects," for more information about creating and using a **File** object.

The methods and properties that you can use with the **File** object are described in the following sections.

***Note**: All the methods and properties, except for the* **IType** *method, used with the* **File** *object can also be used with the* **Dictionary** *object.*

File Object Methods

These are the methods that you can use with the **File** object:

- **ClearFile**
- **CloseFile**
- **DeleteRecord**
- **GetAkInfo**
- **IsOpen**
- **IType**
- **LockFile**
- **LockRecord**
- **Read**
- **ReadField**
- **ReadFields**
- **ReadNamedField**
- **UnlockFile**
- **UnlockRecord**
- **Write**
- **WriteField**
- **WriteFields**
- **WriteNamedField**

ClearFile Method

Syntax

FileObj.**ClearFile**

Description

This method clears a file, deleting all its records.

FileObj is a **File** or **Dictionary** object.

If the file is locked by another session or user, the **BlockingStrategy** property determines the action to be performed.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_clearfile** function and the BASIC CLEARFILE statement.

Example

```
'clear any old contents of the file
AuditObj.ClearFile
'check for failure
If AuditObj.Error <> UVE_NOERROR Then
   MsgBox "File ClearFile Error" & AuditObj.Error
End If
```

CloseFile Method

Syntax

FileObj.**CloseFile**

Description

This method closes a file.

FileObj is a **File** or **Dictionary** object.

Any file or record locks are released. If you try to use *FileObj* following a **CloseFile** call, an error results.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_close** function and the BASIC CLOSE statement.

Example

```
' close the files
  FileObj.CloseFile
  AuditObj.CloseFile
```

DeleteRecord Method

Syntax

FileObj.**DeleteRecord**

Description

This method deletes the record identified by the **RecordId** property.

FileObj is a **File** or **Dictionary** object.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_delete** function and the BASIC DELETE and DELETEU statements.

Example

```
Set CustomerFileObj = SessionObj.OpenFile("CUSTOMER")
CustomerFileObj.RecordId = "49748870081"
CustomerFileObj.DeleteRecord
```

GetAkInfo Method

Syntax

FileObj.**GetAkInfo** *IndexName*

Description

This method obtains information about the secondary key indexes in a **File** object and returns the result as a **DynamicArray** object.

FileObj is a **File** or **Dictionary** object.

IndexName is the field name of the secondary key for which information is required.

The returned information is placed in the **File** object's **Record** property as a dynamic array. The elements of the array are separated by value marks. The meaning of the result depends on the type of index, as follows:

- For D-type indexes: field 1 contains D as the first character and field 2 contains the location number of the indexed field.
- For I-type indexes: field 1 contains I as the first character, field 2 contains the I-type expression, and the compiled I-type code occupies fields 19 onward.
- For both D-type and I-type indexes:
  - The second value of field 1 is 1 if the index needs to be rebuilt, or an empty string otherwise.
  - The third value of field 1 is 1 if the index is null-suppressed, or an empty string otherwise.
  - The fourth value of field 1 is 1 if automatic updates are disabled, or an empty string otherwise.
  - Field 6 contains an S if the index is singlevalued and an M if it is multivalued.

If *IndexName* is an empty string, a list of secondary keys on the file returns as a dynamic array of fields.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_indices** function and the BASIC INDICES function.

Example

```
FileObj.GetAkInfo "POSTCODE"
```

IsOpen Method

Syntax

Bool = *FileObj*.**IsOpen**

Description

This method checks to see if a file is open. It does not change the setting of the **Error** property, and is not affected by the current state of the **Error** property.

Bool is **True** if the file is open, **False** if it is closed.

FileObj is a **File** or **Dictionary** object.

Example

```
Sub MyForm_Unload ()
   ' Window is closing
   ' Close the file if it's still open
   '
   If FileObj is Nothing Then
   Else
      'File was previously opened
      If FileObj.IsOpen Then
         'close the file
         FileObj.CloseFile
      End If
   End If
End Sub
```

IType Method

Syntax

String = FileObj.**IType** *RecordID, ITypeID*

Description

This method evaluates the specified I-descriptor and returns the evaluated string. This method applies no conversions to the data.

String is the returned data.

FileObj is a **File** object.

RecordID is the record ID of the record supplied as data.

ITypeID is the record ID of the I-descriptor record to be evaluated.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_itype** function and the BASIC ITYPE function.

LockFile Method

Syntax

FileObj.**LockFile**

Description

This method locks an associated database file. It does not rely on any of the locking strategy properties such as **BlockingStrategy**, **LockStrategy**, or **ReleaseStrategy**. If a file is already locked by another user it returns an error.

FileObj is a **File** or **Dictionary** object.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_filelock** function and the BASIC FILELOCK statement. See also "UnlockFile Method" on page 3-71.

Example

```
FileObj.LockFile
If FileObj.Error <> UVE_NOERROR Then
   MsgBox "Error attempting to lock file" & FileObj.FileName
   '
   'Error recovery code here
   '
Else
   ' Process the file
End If
FileObj.UnlockFile
```

LockRecord Method

Syntax

FileObj.**LockRecord** *LockType*

Description

This method locks the record identified by the **RecordId** property using the type of lock specified by *LockType*. Use this method to override the current locking strategy.

FileObj is a **File** or **Dictionary** object.

LockType can have one of the following values:

Value	Token	Meaning
1	EXCLUSIVE_UPDATE	Sets an exclusive update lock (READU).
2	SHARED_READ	Sets a shared read lock (READL).

LockType Values

Using this method is equivalent to calling the **Read**, **ReadField**, **ReadFields**, or **ReadNamedField** methods with the **LockStrategy** property set to the value of *LockType*. If the value of *LockType* is not valid, the method returns without performing any locking.

***Note**: You may need to explicitly unlock the record using the* **UnlockRecord** *method, depending upon the value of the* **ReleaseStrategy** *property.*

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_recordlock** function and the BASIC RECORDLOCKL and RECORDLOCKU statements.

Example

```
FileObj.LockRecord SHARED_READ
```

Read Method

Syntax

FileObj.**Read**

Description

This method reads a record identified by the **RecordId** property and returns a **DynamicArray** object in the **Record** property. Record locking is determined by the values of the **BlockingStrategy**, **LockStrategy**, and **ReleaseStrategy** properties.

FileObj is a **File** or **Dictionary** object.

Upon successful completion of this method, the **Record** property contains a **DynamicArray** object for the record read. If an error occurs or the operation is not completed, the **Record** property is set to **Nothing** and the **Error** property contains the error code.

This method corresponds to the InterCall **ic_read** function and the BASIC READ, READL, and READU statements.

Example

```
' read a record
   FileObj.RecordId = "7864"
   FileObj.Read
```

ReadField Method

Syntax

FileObj.**ReadField** *FieldNum*

Description

This method reads the specified field from the record identified by the **RecordId** property. It places the result in the **Record** property as a **DynamicArray** object.

FileObj is a **File** or **Dictionary** object.

FieldNum is the number of the field to be read. If you specify field 0 (the record ID) you can use this method to check if a record exists.

Record locking is defined by the **BlockingStrategy**, **LockStrategy**, and **ReleaseStrategy** properties.

This method corresponds to the InterCall **ic_readv** function and the BASIC READV, READVL, and READVU statements.

Example

```
    FileObj.ReadField 3
```

ReadFields Method

Syntax

FileObj.**ReadFields** *FieldNumArray*

Description

This method reads the specified fields from the record identified by the **RecordId** property. It places the result in the **Record** property as a **DynamicArray** object.

FileObj is a **File** or **Dictionary** object.

FieldNumArray is a long type array containing the numbers of the fields to be read.

Record locking is defined by the **BlockingStrategy**, **LockStrategy**, and **ReleaseStrategy** properties.

This method corresponds to the InterCall **ic_readv** function and the BASIC READV, READVL, and READVU statements.

Example

```
FileObj.ReadFields Fields
```

ReadNamedField Method

Syntax

FileObj.**ReadNamedField** *FieldName*

Description

This method retrieves a value from the specified field and performs any output conversion defined for the field in the file dictionary.

FileObj is a **File** or **Dictionary** object.

FieldName must be a valid field name and must be defined as a D-descriptor or an I-descriptor in the file dictionary. The value of the field is placed in the **Record** property. The value of the **Status** property is undefined.

***Note**: This method needs to read the file dictionary in order to determine the location of the specified field. This can affect the performance of your application. If performance is an issue, use the* **ReadField** *method. For more information about using the* **ReadNamedField** *method, see Data Conversion in Chapter 2, "Using UniObjects."*

The **Error** property is set if an error occurs. If **ReadNamedField** returns the error UVE_RNF (record not found), the missing record may be either the data record you want to read, or the dictionary record that contains *FieldName*.

Example

```
    FileObj.ReadNamedField "CPHONE"
```

UnlockFile Method

Syntax

FileObj.**UnlockFile**

Description

This method unlocks a database file and reverses the action of the **LockFile** method.

FileObj is a **File** or **Dictionary** object.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_fileunlock** function and the BASIC FILEUNLOCK statement. See also “LockFile Method” on page 3-65.

Example

```
FileObj.LockFile
If FileObj.Error <> UVE_NOERROR Then
   MsgBox "Error attempting to lock file" & FileObj.FileName
   End
Else
   ' Process the file
End If
FileObj.UnlockFile
```

UnlockRecord Method

Syntax

FileObj.**UnlockRecord**

Description

This method explicitly unlocks a record identified by the **RecordId** property.

FileObj is a **File** or **Dictionary** object.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_release** function and the BASIC RELEASE statement. See also "LockRecord Method" on page 3-66.

Example

This example assumes that the **LockStrategy** property is set to EXCLUSIVE_UPDATE:

```
FileObj.RecordID = "10001" '  Set record to process
FileObj.Read               ' Set a READU lock
If FileObj.Error = UVE_NOERROR Then
   ProcessRecord           ' Subroutine to process record
   FileObj.UnlockRecord    ' Release the lock
End If
```

Write Method

Syntax

FileObj.**Write**

Description

This method writes the data contained in the **Record** property to the record specified in the **RecordId** property.

FileObj is a **File** or **Dictionary** object.

The value of the **Status** property indicates the state of record locking during the operation as follows:

Valu e	Meaning
0	The record was locked before the operation.
–2	The record was not locked before the operation.

Write Method Status Values

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_write** function and the BASIC WRITE and WRITEU statements.

Example

```
' write record
  FileObj.Write
```

WriteField Method

Syntax

FileObj.**WriteField** *FieldNum*, *String*

Description

This method writes a single field to a record identified by the **RecordId** property.

FileObj is a **File** or **Dictionary** object.

FieldNum is the number of the field to be written.

String is the value to be written to *FieldNum* within the current record.

The value of the **Status** property indicates the state of record locking during the operation as follows:

Value	Meaning
0	The record was locked before the operation.
–2	The record was not locked before the operation.

WriteField Method Status Values

The **Error** property is set if an error occurs.

Example

```
    FileObj.WriteField 5, txtShipper.Text
```

WriteFields Method

Syntax

FileObj.**WriteFields** *FieldNumArray*, *StringArray*

Description

This method writes the specified fields to a record identified by the **RecordId** property.

FileObj is a **File** or **Dictionary** object.

FieldNumArray is a long type array containing the numbers of the fields to be written and its dimensions should match that of *StringArray*.

StringArray is a string array containing the values for the fields identified by *FieldNumArray*.

The value of the **Status** property indicates the state of record locking during the operation as follows:

Value	Meaning
0	The record was locked before the operation.
–2	The record was not locked before the operation.

WriteField Method Status Values

The **Error** property is set if an error occurs.

Example

```
FileObj.WriteFields fields, values
```

WriteNamedField Method

Syntax

FileObj.**WriteNamedField** *FieldName, String*

Description

This method performs the input conversion defined for the field in the file dictionary and writes the field to the record identified by the **RecordId** property. **WriteNamed-Field** does not convert individual values within a multivalued field.

FileObj is a **File** or **Dictionary** object.

FieldName must be a valid field name and must be defined as a D-descriptor in the file dictionary.

String is the value to be written to the file.

The value of the **Status** property is undefined.

Example

```
FileObj.WriteNamedField 'CFAX', txtFax.Text
```

File Object Properties

These are the properties of the **File** object:

- **BlockingStrategy**
- **Error** (read-only)
- **ExceptionOnError**
- **FileName** (read-only)
- **FileType** (read-only)
- **Identifier** (read-only)
- **LockStrategy**
- **Record**
- **RecordId**
- **ReleaseStrategy**
- **Status** (read-only)

BlockingStrategy Property

This property determines the action taken when a database file operation is blocked by a record or file lock. Use this property with the **LockStrategy** and **ReleaseS-trategy** properties. If you do not specify a value, the property inherits its value from the **DefaultBlockingStrategy** property of the **Session** object. The property has one of the following values (all other values are ignored):

Value	Token	Meaning
1	WAIT_ON_LOCKED	If the record is locked, wait until it is released.
2	RETURN_ON_LOCKED	Return a value to the **Status** property to indicate the state of the lock. This is the default value.

BlockingStrategy Property Values

Error Property

This read-only property contains a code for the last error that occurred. If no error has occurred, it contains 0. For a list of error codes and their meanings, see Appendix A, "Error Codes and Replace Tokens." For more information about error conditions, see Error Handling in Chapter 2, "Using UniObjects."

Note: Once this **Error** *property is set to anything other than 0, no other method (except for* **IsOpen***) can be used with the* **File** *object until the error is processed.*

ExceptionOnError Property

Use this property to generate an exception that can be handled by the Visual Basic **On Error** statement. Set this property to **True** to force all errors to generate an exception. The default is **False**, causing only fatal errors to generate an exception.

Note: If you do not specify a value for the **ExceptionOnError** *property, it inherits the value used by the* **Session** *object at the time the* **File** *object was created.*

FileName Property

This read-only property contains the name of the database file supplied in the **OpenFile** method.

FileType Property

When the **OpenFile** method is called, this read-only property contains the file type that is returned in the **Session** object's **Status** property.

Valid file types are:

- 2 through 18 (static hashed files)
- 1 or 19 (directory files)
- 25 (B-tree files)
- 30 (dynamic files)

Identifier Property

This read-only property contains the InterCall file identifier for the **File** object. It is used only in applications that call an InterCall function that requires a file identifier.

LockStrategy Property

This property determines the strategy for setting locks during read operations on the file. If no value is set, the value is inherited from the **DefaultLockStrategy** property of the **Session** object. The **LockStrategy** property has one of the following values:

Value	Token	Meaning
0	NO_LOCKS	No locking.
1	EXCLUSIVE_UPDATE	Sets an exclusive update lock (READU).
2	SHARED_READ	Sets a shared read lock (READL).

LockStrategy Property Values

This property is used with the **BlockingStrategy** and **ReleaseStrategy** properties.

Record Property

This property holds the contents of a record obtained as a **DynamicArray** object. It is updated whenever a call is made to the **Read**, **ReadField**, **ReadFields** or **ReadNamedField** methods.

RecordId Property

This property contains the ID of the record to be processed by **File** object methods such as **Read** or **Write**.

ReleaseStrategy Property

This property determines the strategy for releasing locks set during reads and calls to **LockRecord**. Use this property with the **BlockingStrategy** and **LockStrategy** properties. If no value is set, the value is inherited from the **DefaultReleaseStrategy** property of the **Session** object.

This property can have one of the following values:

Value	Token	Meaning
1	WRITE_RELEASE	Releases the lock when the write finishes. This is the property's initial value.
2	READ_RELEASE	Releases the lock when the read finishes.
4	EXPLICIT_RELEASE	Maintains locks as specified by the **LockStrategy** property. Locks can be released only with the **UnlockRecord** method.
8	CHANGE_RELEASE	Releases the lock whenever a new value is assigned to the **RecordId** property.

ReleaseStrategy Property Values

All the values are additive. If you specify EXPLICIT_RELEASE with WRITE_RELEASE and READ_RELEASE, it takes a lower priority. The initial value of **ReleaseStrategy** is set to 12, that is, release locks when the value of the **RecordId** property changes or when locks are released explicitly.

Status Property

This read-only property contains a status code returned by a method. Refer to each method for a description of any status values that are returned.

Example Using the File Object

```
Sub SampleFile ()
   ' SampleFile
   '
   ' This routine creates a new session and opens the chosen
account's VOC
   ' file. The user is asked for a record id from VOC (e.g.
RELLEVEL), which
   ' is read and displayed in a message box. Finally the session
is closed.
   Dim objSession As object    ' The Session to the database
   Dim objFile As object       ' The file to open (VOC)
   Const UVE_NOERROR = 0       ' From UVOAIF.TXT - no error
   ' The registered name of a database Session - Version 1
   Const UV_SESSION_OBJECT = "UniObjects.unioaifctrl"
   '
   ' Create a Session object to work with
   '  - This is a contrived sample, in a full application the
session object
   '  - would typically be a Global variable that is set once
maybe in
   '  - response to a menu selection (e.g. Connect) on the main
form.
   '
   Set objSession = CreateObject(UV_SESSION_OBJECT)
   If objSession Is Nothing Then
      ' NB. Errors will be reported by VB
      Exit Sub                     ' End the program
   End If
   objSession.UserName = Input.Box ("User Name:","Login")
   objSession.Password = Input.Box ("Password:","Password")
   '
   ' Establish a connection to the database server. By default it
displays
   ' a dialog box to request the HostName and AccountPath property
values.
   '
   objSession.Connect
   If objSession.IsActive Then
      '
      ' Open the VOC file
      '
      Set objFile = objSession.OpenFile("VOC")
      If objFile Is Nothing Then
         MsgBox "Unable to open VOC file (" & objSession.Error &
")"
         Exit Sub                ' End the program
      End If
      '
      ' Read user entered record from the VOC e.g. RELLEVEL
      '
      objFile.RecordId = InputBox("Enter Record Id:", "Record Id")
      objFile.Read
      If objFile.Error = UVE_NOERROR Then
         ' Display the record in a message box and close file
         MsgBox objFile.Record
         objFile.CloseFile   ' Close the file - Good practice
      Else
         MsgBox "Unable to read (" & objFile.RecordId & ") record
from
         ? VOC " & objFile.Error
      End If
      '
      ' Close the session
      '
      objSession.Disconnect
   Else
      '
      ' Check for Session errors - display message box with error
code
      ' No error means the user cancelled the connection dialog
box
      '
      If objSession.Error <> UVE_NOERROR Then
         MsgBox "Unable to open connection:- " & objSession.Error
      End If
   End If
End Sub
```

Dictionary Object

A **Dictionary** object defines and manages a file dictionary. You use the **Dictionary** object through the **OpenDictionary** method of the **Session** object. The **Dictionary** object resembles the **File** object and shares all its properties and methods. In addition, the **Dictionary** object has properties that refer to specific fields in a dictionary record. For more information about file dictionaries and how to use them, see The Database Environment and Using a Dictionary in Chapter 2, “Using UniObjects.” For more information about the fields in a dictionary, see *UniVerse System Description*.

The methods and properties that you can use with the **Dictionary** object are described in the following sections.

Dictionary Object Methods

These are the methods that you can use with the **Dictionary** object:

- **ClearFile**
- **CloseFile**
- **DeleteRecord**
- **GetAkInfo**
- **IsOpen**
- **LockFile**
- **LockRecord**
- **Read**
- **ReadField**
- **ReadNamedField**
- **UnlockFile**
- **UnlockRecord**
- **Write**
- **WriteField**
- **WriteNamedField**

For more information about these methods, see “File Object Methods” on page 3-57.

Dictionary Object Properties

These are the properties of the **Dictionary** object:

- **ASSOC**
- **BlockingStrategy***
- **CONV**
- **Error***
- **ExceptionOnError***
- **FileName***
- **FileType***
- **FORMAT**
- **Identifier***
- **LOC**
- **LockStrategy***
- **NAME**
- **Record***
- **RecordId***
- **ReleaseStrategy***
- **SM**
- **SQLTYPE**
- **Status***
- **TYPE**

***** This property is the same as that of the **File** object. See its description under "File Object Properties" on page 3-77.

ASSOC Property

This property contains the contents of the ASSOC field (field 7) from a dictionary record identified by the **RecordId** property. The **Error** property is set if an error occurs while the property is being set or retrieved. See also "ReadField Method" on page 3-68 and "WriteField Method" on page 3-74.

CONV Property

This property contains the contents of the CONV field (field 3) from a dictionary record identified by the **RecordId** property. The CONV field can contain any of the BASIC conversion codes that are used to format data for output or for internal storage.

The **Error** property is set if an error occurs while the property is being set or retrieved. For more information about conversion codes, see *UniVerse BASIC*. See also "ReadField Method" on page 3-68 and "WriteField Method" on page 3-74.

FORMAT Property

This property contains the contents of the FORMAT field (field 5) from a dictionary record identified by the **RecordId** property. The **Error** property is set if an error occurs while the property is being set or retrieved.

LOC Property

This property contains the contents of the LOC field (field 2) from a dictionary record identified by the **RecordId** property. The **Error** property is set if an error occurs while the property is being set or retrieved.

NAME Property

This property contains the contents of the NAME field (field 4) from a dictionary record identified by the **RecordId** property. The **Error** property is set if an error occurs while the property is being set or retrieved.

SM Property

This property contains the contents of an SM field (field 6) from a dictionary record identified by the **RecordId** property. The **Error** property is set if an error occurs while the property is being set or retrieved.

SQLTYPE Property

This property contains the contents of the SQLTYPE field (field 8) from a UniVerse dictionary record identified by the **RecordId** property. The **Error** property is set if an error occurs while the property is being set or retrieved.

TYPE Property

This property contains the contents of the TYPE field (field 1) from the dictionary record identified by the **RecordId** property. The first characters of the TYPE field indicate the type of field the dictionary record is defining.

Valid types are:

D	D-descriptor
I	I-descriptor
V	(UniData only) V-descriptor
PH	Phrase
X	(UniVerse only) X-descriptor

The **Error** property is set if an error occurs while the property is being set or retrieved.

Example Using the Dictionary Object

```
Sub SampleDict ()
   '
   ' SampleDict
   '
   ' This sample routine will create a new session and ask the
   ' user for a file name. The dictionary of that file is then
   ' opened and the location of each D-type record and the code of
   ' each I-type record in the dictionary will be displayed in a
   ' message box. Finally, the session is closed.

   Dim objSession As object    ' The Session to the database
   Dim objDict As object       ' The dictionary to open
   Dim objSelect As object     ' For getting dictionary records
   Dim strFile As String       ' File name input by user

   Const UVE_NOERROR = 0       ' From UVOAIF.TXT - no error
   ' The registered name of a database Session - Version 1
   Const UV_SESSION_OBJECT = "UniObjects.unioaifctrl"
   Const IDCANCEL = 2             ' Cancel button id
   Const MB_OKCANCEL = 1             ' OK-CANCEL message box style
   '
   ' Create a Session object to work with
   '   - This is a contrived sample, in a full application the
   '   - Session object would usually be a Global variable
   '   - that is set once, perhaps in response to a menu selection
   '   - such as Connect on the main form.
   '
   Set objSession = CreateObject(UV_SESSION_OBJECT)
   If objSession Is Nothing Then
      ' NB. Errors will be reported by VB
      Exit Sub                       ' End the program
   End If
   objSession.UserName = Input.Box ("User Name:","Login")
   objSession.Password = Input.Box ("Password:","Password")

   '
   ' Establish a connection to the database server. By default it
   ' displays a dialog box to request the HostName and AccountPath
   ' property values.
   '
   objSession.Connect
   If objSession.IsActive Then
      '
      strFile = InputBox("Enter file name:", "Dictionary Sample")
      Set objDict = objSession.OpenDictionary(strFile)
      If objDict Is Nothing Then
         MsgBox "Unable to Open Dictionary of: " & strFile & " ("
&
         ?objSession.Error & ")"
         Exit Sub          ' End the subroutine
      End If
      '
      ' Create a select list on the dictionary - use select list 1
      '
      Set objSelect = objSession.SelectList(1)
      If objSelect Is Nothing Then
         MsgBox "Unable to create select list 1
         ?(" & objSession.Error & ()
         Set objDict = Nothing ' Tidy up - not needed, but good
practice
         Exit Sub
      End If
      '
      ' perform the select on the open dictionary - check for
errors
      '
      objSelect.Select objDict                ' Establish a select
list
      objDict.RecordId = objSelect.Next       ' get the first
record id
      Do While Not objSelect.LastRecordRead
         '
         ' Check type of record: for D-type, display location; for
         ' I-type display code (both accessed by LOC property)
         ' (When checking type just look at the first character as
TYPE
         ' returns the whole field)
         '
         Select Case UCase(Left(objDict.TYPE, 1))
         Case "D"
            If MsgBox("Location of " & objDict.RecordId & ": " &
            ?objDict.LOC, MB_OKCANCEL) = IDCANCEL Then
               ' Cancel button hit
               Exit Do
            End If
         Case "I"
            If MsgBox("I-type code of " & objDict.RecordId & ": "
&
            ?objDict.LOC, MB_OKCANCEL) = IDCANCEL Then
               ' Cancel button hit
               Exit Do
            End If
         End Select
         objDict.RecordId = objSelect.Next   ' Get the next record
id
      Loop
      '
      ' Tell the user if the last record was read
      '
      If objSelect.LastRecordRead Then
         MsgBox "Last Record has been Read!"
      End If
      objDict.CloseFile                       ' Close the
dictionary
      objSession.Disconnect                   ' Close session
   Else
```

```
      '
      ' Check for Session errors - display message box with error
code
      ' No error means the user cancelled the connection dialog
box
      '
      If objSession.Error <> UVE_NOERROR Then
      MsgBox "Unable to open connection:- " & objSession.Error
      End If
   End If
End Sub
```

SequentialFile Object

The **SequentialFile** object defines and manages a sequential file. A sequential file is an operating system file on the server containing text or binary data that you want to use in your application. Sequential files are defined *on the server* as UniVerse type 1 or type 19 files. You create a **SequentialFile** object through **OpenSequential** method of the **Session** object. For more information about using the **SequentialFile** object, see Using Binary and Text Files in Chapter 2, "Using UniObjects." For a program example that uses the **SequentialFile** object, see "Example Using the SequentialFile Object" on page 3-104.

The methods and properties that you can use with the **SequentialFile** object are described in the following sections.

SequentialFile Object Methods

These are the methods that you can use with the **SequentialFile** object:

- **CloseSeqFile**
- **FileSeek**
- **IsOpen**
- **ReadBlk**
- **ReadLine**
- **WriteBlk**
- **WriteEOF**
- **WriteLine**

CloseSeqFile Method

Syntax

SeqFileObj.**CloseSeqFile**

Description

This method closes a sequential file.

SeqFileObj is a **SequentialFile** object. If you attempt to use this object following a *CloseSeqFile* call, an error results.

This method corresponds to the InterCall **ic_closeseq** function and the BASIC CLOSESEQ statement.

Example

```
' close the files
  SeqFileObj.CloseSeqFile
```

FileSeek Method

Syntax

SeqFileObj.**FileSeek** *RelPos, Offset*

Description

This method moves the file pointer within a sequential file by an offset specified in bytes, relative to the current position, the beginning of the file, or the end of the file.

SeqFileObj is a valid **SequentialFile** object.

RelPos is the pointer's relative position in a file. Possible values are:

Value	Token	Meaning
0	UVT_START	The start of the file.
1	UVT_CURR	The current position.
2	UVT_END	The end of the file.

RelPos Values

Offset is the number of bytes before or after *RelPos*. A negative offset moves the pointer to a position before *RelPos*.

This method corresponds to the InterCall **ic_seek** function and the BASIC SEEK statement.

Example

```
SeqFileObj.FileSeek UVT_CURR, -1024
```

IsOpen Method

Syntax

Bool = *SeqFileObj*.**IsOpen**

Description

This method checks to see if a file is open. It does not affect the previous state of the error code, and it is not affected by the current error condition.

Bool is 0 if the file is closed; any other value indicates the file is open.

SeqFileObj is a **SequentialFile** object.

Example

```
Sub MyForm_Unload ()
   ' Window is closing
   ' Close the file if it is still open
   '
   If SeqFileObj is Nothing Then
   Else
      'File was previously opened
      If SeqFileObj.IsOpen Then
         'close the file
         FileObj.CloseSeqFile
      End If
   End If
End Sub
```

ReadBlk Method

Syntax

String = *SeqFileObj*.**ReadBlk**

Description

This method reads a block of data from a sequential file. The size of the data block is specified in the **ReadSize** property.

String is the returned block of data. If the method fails for any reason an empty string is returned.

SeqFileObj is a **SequentialFile** object.

When the **ReadBlk** method completes, the **ReadSize** property contains the size of the returned string in bytes, or 0 if the method failed.

These are the values that can be returned to the **Status** property by the **ReadBlk** method:

Value	Meaning
–1	The file is not open for a read.
0	The read was successful.
1	The end of the file was reached.

ReadBlk Status Values

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_readblk** function and the BASIC READBLK statement.

Example

```
DataRecord = SeqFileObj.ReadBlk
```

ReadLine Method

Syntax

String = *SeqFileObj*.**ReadLine**

Description

This method reads successive lines of data from the current position in a sequential file. The lines must be delimited by an end-of-line character, such as a carriage return.

String is the returned line of data.

SeqFileObj is a **SequentialFile** object.

These are the values that may be returned to the **Status** property by the **ReadLine** method:

Value	Meaning
–1	The file is not open for a read.
0	The read was successful.
1	The end of the file was reached, or the value of the **ReadSize** property is 0 or less.

ReadLine Status Values

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_readseq** function and the BASIC READSEQ statement.

Example

```
DataLine = SeqFileObj.ReadLine
```

WriteBlk Method

Syntax

SeqFileObj.**WriteBlk** *String*

Description

This method writes successive blocks of data to a binary file at the current position. This method updates the **Error** property.

SeqFileObj is a **SequentialFile** object.

String is the block of data to be written.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_writeblk** function and the BASIC WRITEBLK statement.

Example

```
    SeqFileObj.WriteBlk Next512
```

WriteEOF Method

Syntax

SeqFileObj.**WriteEOF**

Description

This method writes an end-of-file marker at the current position. This allows an existing file to be truncated at a specified point when used with the **FileSeek** method.

SeqFileObj is a **SequentialFile** object.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_weofseq** function and the BASIC WEOFSEQ statement.

Example

```
Set SeqFileObj = SessObj.OpenSequential("TESTDATA", "TEST1",
False)
SeqFileObj.WriteEOF
```

WriteLine Method

Syntax

SeqFileObj.**WriteLine** *String*

Description

This method writes successive lines of data at the current position.

SeqFileObj is a **SequentialFile** object.

String is a line of data to be written to the file.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_writeseq** function and the BASIC WRITESEQ statement. See also "ReadLine Method" on page 3-97.

Example

```
SeqFileObj.WriteLine ShipAddress(3)
```

SequentialFile Object Properties

These are the properties of the **SequentialFile** object:

- **Error** (read-only)
- **ExceptionOnError**
- **FileName** (read-only)
- **Identifier** (read-only)
- **ReadSize**
- **RecordId** (read-only)
- **Status** (read-only)
- **Timeout**

Error Property

This read-only property contains the last error that occurred. If no error has occurred, it contains 0. For a list of error codes and their meanings, see Appendix A, "Error Codes and Replace Tokens." For more information about error conditions, see Error Handling in Chapter 2, "Using UniObjects."

***Note**: Once this* **Error** *property is set to anything other than 0, no other method can be used with the* **SequentialFile** *object until the error is processed.*

ExceptionOnError Property

Use this property to generate an exception that can be handled by the Visual Basic **On Error** statement. Set this property to **True** to force all errors to generate an exception. The default is **False**, causing only fatal errors to generate an exception.

***Note**: If you do not specify a value for the* **ExceptionOnError** *property, it inherits the value used by the* **Session** *object at the time the* **SequentialFile** *object was created.*

FileName Property

This read-only property contains the name of the type 1 or type 19 file supplied in the **OpenSequential** method.

Identifier Property

This read-only property contains the InterCall file identifier for the **SequentialFile** object. It is used only in applications that call an InterCall function that requires a file identifier.

ReadSize Property

This property specifies the number of bytes to read for each successive call to the **ReadBlk** method. This is initially set to 0, which indicates that all the data should be read in a single block. You should set the value to a suitable number of bytes for the memory available to your application. Values less than 0 are treated as 0.

***Warning**: If the value is set to 0 and there is not enough memory to hold all the data, a run-time exception occurs.*

When the **ReadBlk** method completes, the value of **ReadSize** is reset to the number of bytes that were actually read. A value of 0 indicates an error, or the end of the file.

***Note**: You should reset the* **ReadSize** *property before each use of the* **ReadBlk** *method because the* **ReadSize** *value may have been modified by a previous operation.*

RecordId Property

This read-only property contains the record ID that was supplied in the **OpenSequential** method.

Status Property

This read-only property contains a status code returned by a method. Refer to each method for a description of any status values that are returned.

Timeout Property

This property specifies the timeout for **ReadBlk** operations.

The default value is 0. The timeout is specified in seconds.

This property corresponds to the InterCall **ic_set_comms_timeout** function.

Example Using the SequentialFile Object

```
Sub SampleSeqFil ()
   '
   ' SampleSeqFil
   '
   ' This sample illustrates the use of the SequentialFile object.
   ' This routine creates a session on the server, opens the file
   ' specified by the user, reads the whole file and displays it
in a
   ' message box.

   Dim objSession As Object ' Object variable for session
   Dim objSeq As Object ' Object Variable for Seq File
   Dim strFile As String ' File name
   Dim strItem As String ' Record Id
   Dim strResult As String ' Result of reading the record

   Const UVE_NOERROR = 0                   ' From UVOAIF.TXT - no
error
   ' The registered name of a database Session - Version 1
   Const UV_SESSION_OBJECT = "UniObjects.unioaifctrl"
   ' Create a Session object to work with
   '    - This is a contrived sample, in a full application the
   '    - Session object would normally be a Global variable
   '    - that is set once in response to a menu selection such as
   '    - Connect on the main form.
   Set objSession = CreateObject(UV_SESSION_OBJECT)
   If objSession Is Nothing Then
      ' NB. Errors will be reported by Visual Basic
      Exit Sub                              ' End program
   End If
   objSession.UserName = Input.Box ("User Name:","Login")
   objSession.Password = Input.Box ("Password:","Password")

   '
   ' Establish a connection to the database server. By default it
   ' displays a dialog box to request the HostName and AccountPath
   ' property values.
   '
   objSession.Connect
   If objSession.IsActive Then
      '
      ' Get the user to supply a filename and record ID
      '
      strFile = InputBox("Enter file name:", "Sequential File
Sample")
      strItem = InputBox("Item in " & strFile & ":", "Sequential
File
      ? Sample")
      Set objSeq = objSession.OpenSequential(strFile, strItem,
False)
```

```
      ' don't create
      If objSeq Is Nothing Then
         MsgBox "Unable to open " & strFile & " " & strItem & " ("
&
         ? objSession.Error & ")"
         Exit Sub
      End If
      ' Display the contents of the file. As the default ReadSize
is 0,
      ' the entire record will be returned by ReadBlk
      strResult = objSeq.ReadBlk
      If objSeq.Error = UVE_NOERROR Then
         MsgBox strResult
      End If
      objSeq.CloseSeqFile
      objSession.Disconnect   ' Disconnect session
   Else
      ' Here if session is not active
      ' Check for Session errors - display message box with error
code
      ' No error means the user cancelled the connection dialog
box
      If objSession.Error <> UVE_NOERROR Then
         MsgBox "Unable to open connection:- " & objSession.Error
      End If
   End If
End Sub
```

DynamicArray Object

The **DynamicArray** object allows you to manipulate fields, values, and subvalues in a dynamic array such as a record, or a returned select list. **DynamicArray** objects are used in:

- The **Record** property of the **File** and **Dictionary** objects
- The **ReadList** method of the **SelectList** object

You can also create a **DynamicArray** object independently of a session by specifying:

```
set DynArrayObj = CreateObject(UV_DARRAY_OBJECT)
```

UV_DARRAY_OBJECT is a registered object name representing the **DynamicArray** object for use with **CreateObject**.

For more information about the **DynamicArray** object, see Fields, Values, and Subvalues in Chapter 2, "Using UniObjects." The methods and properties that you can use with the **DynamicArray** object are described in the following sections.

DynamicArray Object Methods

These are the methods that you can use with the **DynamicArray** object:

- **Count**
- **Del**
- **Field**
- **Ins**
- **Length**
- **Replace**
- **SubValue**
- **Value**

Count Method

Syntax

Integer = *DynArrayObj*[.*Context*].**Count**

Description

This method counts the number of fields, values, or subvalues in a dynamic array.

Integer is the returned integer count of fields, values, or subvalues.

DynArrayObj is a **DynamicArray** object.

Context is a **Field**, **Value**, or **SubValue** method that defines what is to be counted.

The **Error** property is set if an error occurs.

This method corresponds to the BASIC DCOUNT function.

Example

```
Dim DynArrayObj As Object
Dim NumFields As Integer
Dim NumValues As Integer

Set DynArrayObj = File.Record' reference the dynamic array
NumFields = DynArrayObj.Count' count all fields in the
            ' record
NumValues = DynArrayObj.Field(2).Count' count values in field 2
```

Del Method

Syntax

Bool = *DynArrayObj*.[*Context*].**Del**

Description

This method deletes the specified fields, values, or subvalues from a dynamic array.

Bool is set to **True** or **False** to indicate whether or not the operation was successful. A **False** value indicates no change.

DynArrayObj is a **DynamicArray** object.

Context is a **Field**, **Value**, or **SubValue** method that defines the part of the dynamic array to be deleted.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_strdel** function and the BASIC DEL statement.

Example

```
Dim DynArrayObj As Object

Set DynArrayObj = File.Record
DynArray.Value(2, 3).Del' delete value 3 of field 2
DynArray.Del' delete the entire dynamic array
```

Field Method

Syntax

DynArrayObj.**Field**(*FieldNum*)[.*operation*]

Description

This method supplies the field context within a dynamic array to be used by another method or property.

DynArrayObj is a **DynamicArray** object.

FieldNum is the number of the field to be used. A value of 0 is treated as 1. A value of –1 or less appends a field to the dynamic array.

operation is a **DynamicArray** object method or property such as **Ins** or **Del**. If *operation* is omitted, as in the example, the value of the **StringValue** property is used.

Example

```
MsgBox DynArrayObj.Field(3)' Field 3 in MsgBox
DynArrayObj.Field(3) = txtCustomer.Text' Assign to field 3
DynArrayObj.Field(3).Replace txtCustomer.Text 'Replace field 3
```

Ins Method

Syntax

DynArrayObj.[*Context*].**Ins** *String*

Description

This method inserts a string into a dynamic array, moving subsequent fields or values down.

DynArrayObj is the **DynamicArray** object.

Context is a **Field**, **Value**, or **SubValue** method that defines where the string is to be inserted.

String is the value to be inserted.

If you do not specify *Context*, the entire dynamic array is replaced.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_insert** function and the BASIC INS statement.

Example

```
DynArrayObj.Ins "Java"' Replaces entire dynamic array
DynArray.Field(2).Ins "Java"' Inserts Java as Field 2, old
                  ' field 2 becomes field 3, and
                  ' so on.
```

Length Method

Syntax

Long = *DynArrayObj*.[*Context*].**Length**

Description

This method obtains the length of the specified field, value, or subvalue.

Long is the returned length in number of characters, including system delimiters.

DynArrayObj is the **DynamicArray** object.

Context is a **Field**, **Value**, or **SubValue** method that specifies which field, value or subvalue to use.

The **Error** property is set if an error occurs.

Example

```
len = DynArrayObj.Value(1,2).Length' length of field 1, value 2
len = DynArrayObj.Length' length of entire array
```

Replace Method

Syntax

DynArrayObj.[*Context*].**Replace** *String*

Description

This method replaces all or part of a dynamic array with a specified string.

DynArrayObj is a **DynamicArray** object.

Context is a **Field**, **Value**, or **SubValue** method that specifies the part of the dynamic array to replace.

String is the replacement value.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_replace** function and the BASIC REPLACE function.

Example

```
' Replace contents of field 10 with the string AssetCategory
  DynArrayObj.Field (10).Replace "AssetCategory"
```

SubValue Method

Syntax

DynArrayObj.**SubValue**(*FieldNum*, *ValueNum*, *SubValueNum*)[.*operation*]

Description

This method supplies the subvalue context within a dynamic array to be used by another method or property.

DynArrayObj is a **DynamicArray** object.

FieldNum is the number of the field to be used. A value of 0 is treated as 1. A value of –1 or less appends a field to the dynamic array.

ValueNum is the number of the value within *FieldNum*. A value of 0 is treated as 1. A value of –1 or less appends a value to *FieldNum*.

SubValueNum is the number of the subvalue within *ValueNum*. A value of 0 is treated as 1. A value of –1 or less appends a subvalue to *ValueNum*.

operation is a **DynamicArray** object method or property such as **Ins** or **Del**. If *operation* is omitted, as in the example, the value of the **StringValue** property is used.

Example

```
DynArrayObj.SubValue (10,1,5) = DocumentationPartNo(5)
```

Value Method

Syntax

DynArrayObj.**Value**(*FieldNum*, *ValueNum*)[.*operation*]

Description

This method supplies the value context within a dynamic array to be used by another method or property.

DynArrayObj is the **DynamicArray** object.

FieldNum is the number of the field to be used. A value of 0 is treated as 1. A value of –1 or less appends a field to the dynamic array.

ValueNum is the number of the value within *FieldNum*. A value of 0 is treated as 1. A value of –1 or less appends a value to *FieldNum*.

operation is a **DynamicArray** object method or property such as **Ins** or **Del**. If *operation* is omitted, the value of the **StringValue** property is used.

Example

```
Bool = DynArrayObj.Value(2,3).Del' Delete value 3 of field 2
```

DynamicArray Object Properties

These are the properties of the **DynamicArray** object:

- **Error** (read-only)
- **ExceptionOnError**
- **StringValue** (default)
- **TextValue**

Error Property

This read-only property contains the last error that occurred. If no error has occurred, it contains 0. For a list of error codes and their meanings, see Appendix A, "Error Codes and Replace Tokens." For more information about error conditions, see Error Handling in Chapter 2, "Using UniObjects."

***Note**: Once this **Error** property is set to anything other than 0, no other method can be used with the **DynamicArray** object until the error is processed.*

ExceptionOnError Property

Use this property to generate an exception that can be handled by the Visual Basic **On Error** statement. Set this property to **True** to force all errors to generate an exception. The default is **False**, causing only fatal errors to generate an exception.

***Note**: You are advised to set the value of the **ExceptionOnError** property to **True**, as errors generated in **DynamicArray** object operations are likely to be fatal. If you do not specify a value, the **ExceptionOnError** property inherits the value used by the **Session** object at the time the **DynamicArray** object was created.*

StringValue Property

This property contains the contents of the **DynamicArray** object as a string. This is the default property of the **DynamicArray** object. Assigning a string value directly to the object has the same effect as assigning the string to this property. The reverse is true for referencing the object. For example:

```
Dim strAString As String

strAString = objDyn.Field(1)' is equivalent to
strAString = objDyn.Field(1).StringValue
```

And:

```
objDyn = strAString' is equivalent to
objDyn.StringValue = strAString
```

TextValue Property

This property contains the contents of the **DynamicArray** object as a string. Assigning a string value directly to the object has the same effect as assigning the string to this property. The reverse is true for referencing the object. For example:

```
Dim strAString As String

strAString = objDyn.Field(1).TextValue
```

And:

```
objDyn.TextValue = strAString
```

This property specifies the content of the **DynamicArray** object, translating delimiters to CRLFs (carriage-return/linefeed pairs).

Syntax	Description
data = *DynArrayObj*.Field(3).**TextValue**	The string data from field 3 of the dynamic array is returned with the value marks translated to CRLFs. No translation is performed on subvalue marks.
data = *DynArrayObj*.**TextValue**	The entire contents of the dynamic array is returned with field marks translated to CRLFs. No translation is performed on value or subvalue marks.
DynArrayObj.Value(2,5).**TextValue** = *data*	The string data writes into value 5 of field 2 of the dynamic array with CRLFs translated to subvalue marks.

TextValue Property Syntax

Example Using the DynamicArray Object

```
Sub SampleDArray ()
   '
   ' SampleDArray
   '
   ' This sample illustrates the use of the DynamicArray object.
This
   ' routine prompts the user for a number of strings. These are
stored
   ' in a dynamic array at a position representing the strings'
length.
   ' When the user has finished entering data, the array is
processed to
   ' count the number of strings entered of each length.
   '
   ' In an application using other objects, a dynamic array object
can be
   ' obtained from any of the File or Dictionary's Record
property, or
   ' returned from the Select object's ReadList method.
   '
   Dim objDArray As Object ' Dynamic array
   Dim ans As String ' User input
   Dim i As Integer ' Loop increment
   Dim iCount As Integer ' Count of fields
   Dim iWordCount As Integer ' Count of strings of a given length
   Dim strMsg As String ' Message text
   Dim btn        ' Button selected

   Const IDYES = 6 ' Yes button
   Const MB_YESNO = 4 ' Yes/No Message box style
   Const MB_ICONQUESTION = 32 ' Question type message box
   Const UV_QUERY = ", Do you want to view them?"
   Const UV_SAMPLE = "Dynamic Array Sample"
   ' The registered name of a Dynamic Array - Version 1
   Const UV_DARRAY_OBJECT = "UniObjects.UniDynArray"

   ' Create a Dynamic Array object - exit if we cannot create it
   Set objDArray = CreateObject(UV_DARRAY_OBJECT)
   If objDArray Is Nothing Then
      ' Visual Basic will report any errors
      Exit Sub                         ' End the program
   End If
   ' Prompt the user for input - continue until user enters "" or
cancel
   Do
      ans = InputBox("Enter String:", UV_SAMPLE)
      If ans <> "" Then
         ' Append to field offset by string length as a new value
         objDArray.Value(Len(ans), -1).StringValue = ans
      End If
   Loop While ans <> ""
   ' Now report what the user entered
   iCount = objDArray.Count
   For i = 1 To iCount Step 1
      ' Process the dynamic array, checking each field for entered
data
      ' Only report where data has been entered
      If objDArray.Field(i).Length > 0 Then
         ' Tell the user how many strings were entered of each
length
         iWordCount = objDArray.Field(i).Count ' count the number
of
         ? strings
         strMsg = "You entered " & Str(iWordCount) & " strings of
" &
         ? Str(i) & " characters"
         strMsg = strMsg & UV_QUERY
         btn = MsgBox(strMsg, MB_YESNO + MB_ICONQUESTION,
UV_SAMPLE)
         ' Display the field if the user hit the YES button
         If btn = IDYES Then
            MsgBox objDArray.Field(i).StringValue, MB_OK,
UV_SAMPLE
         End If
      End If
   Next i

   ' Tidy up object

   Set objDArray = Nothing

End Sub
```

SelectList Object

The **SelectList** object allows you to manipulate a select list on the server. You define this object through the **SelectList** method of the **Session** object, as follows:

```
Set SelectListObj = Sess.Obj.SelectList(0)
```

Select lists are described in The Database Environment and Select Lists in Chapter 2, "Using UniObjects." The methods and properties that you can use with the **SelectList** object are described in the following sections.

SelectList Object Methods

These are the methods that you can use with the **SelectList** object:

- **ClearList**
- **FormList**
- **GetList**
- **Next**
- **ReadList**
- **SaveList**
- **Select**
- **SelectAlternateKey**
- **SelectMatchingAk**

ClearList Method

Syntax

SelectListObj.**ClearList**

Description

This method clears a select list.

SelectListObj is a **SelectList** object.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_clearselect** function and the BASIC CLEARSELECT statement.

Example

```
' clear the contents of the select list
   SelectListObj.ClearList
```

FormList Method

Syntax

SelectListObj.**FormList** *String*

SelectListObj.**FormList** *DynArrayObj*

Description

This method creates a select list from a supplied list of record IDs.

SelectListObj is a **SelectList** object.

String is the location for the record IDs to be made into a select list.

DynArrayObj is a **DynamicArray** object containing record IDs to be made into a select list.

In the first syntax, the record IDs for the new select list are located in *String*. The record IDs in *String* must be separated by field marks (CHAR 254).

In the second syntax, the record IDs for the new select list are stored in a **DynamicArray** object such as the **Record** property.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_formlist** function and the BASIC FORMLIST statement.

Example

```
' Make a select list from the record IDs held in "Storage"
SelectListObj.FormList Storage
.
.
.
' Make a select list from the data in the Record property
' which is held as a dynamic array
SelectListObj.FormList FileObj.Record
```

GetList Method

Syntax

SelectListObj.**GetList**(*ListName*)

Description

This method activates the named list from the &SAVEDLISTS& file.

SelectListObj is a **SelectList** object.

ListName is the list you want to activate.

This method corresponds to the InterCall **ic_getlist** function and the GET.LIST command.

Next Method

Syntax

String = *SelectListObj*.**Next**

Description

This method returns the next record ID in the select list.

String is the value of the next record ID from the select list. *String* is set to an empty string if an error occurs or the list is exhausted.

SelectListObj is a **SelectList** object.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_readnext** function and the BASIC READNEXT statement.

Example

```
UVFile.RecordId = MySelectList.Next' get the first record
               ' from the list
While Not MySelectList.LastRecordRead' check for end of list
   UVFile.Read ' read the next record
   <process the record>
   UVFile.RecordId = MySelectList.Next' get the next record
               ' from the list
End
```

ReadList Method

Syntax

Set *DynArrayObj* = *SelectListObj*.**ReadList**

Description

This method obtains the entire contents of a select list as a dynamic array.

DynArrayObj is an object variable.

SelectListObj is a **SelectList** object.

If an error occurs **Nothing** is returned and the **Error** property is set.

This method corresponds to the InterCall **ic_readlist** function and the BASIC READLIST statement.

Examples

```
Set FileObj.Record = SelectListObj.ReadList

Set DynObj = SelectListObj.ReadList
```

SaveList Method

Syntax

SelectListObj.**SaveList**(*ListName*)

Description

This method saves the named list in the &SAVEDLISTS& file.

SelectListObj is a **SelectList** object.

ListName is the list you want to save.

This method corresponds to the SAVE.LIST command.

Select Method

Syntax

SelectListObj.**Select** *FileObj*

Description

This method creates a select list containing all record IDs from a database file.

SelectListObj is a **SelectList** object.

FileObj identifies the file containing the secondary key index. It must be a **Dictionary** or **File** object returned by the **Session** object's **OpenFile** or **OpenDictionary** method.

The new select list overwrites the select list specified in *SelectListObj* and resets the select list pointer to the first record in the list.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_select** function and the BASIC SELECT statement, and it is equivalent to the database command:

SELECT *filename*

Example

```
' open a select list
   Set SelectListObj = SessObj.SelectList(0)
   SelectListObj.Select FileObj
```

SelectAlternateKey Method

Syntax

SelectListObj.**SelectAlternateKey** *FileObj*, *IndexName*

Description

This method creates a select list from values in the specified secondary key index.

SelectListObj is a **SelectList** object.

FileObj identifies the file containing the secondary key index. It must be a **Dictionary** or **File** object returned by the **Session** object's **OpenFile** or **OpenDictionary** method.

IndexName is the name of a secondary key index. This name must have been specified in a CREATE.INDEX command.

If the named secondary key index does not exist, the select list is empty. The new select list overwrites the select list specified in *SelectListObj* and resets the select list pointer to the first record in the list.

The **Error** property is set if an error occurs.

This method corresponds to a subset of the functionality of the InterCall **ic_selectindex** function and the BASIC SELECTINDEX statement.

Example

```
SelectListObj.SelectAlternateKey FileObj, 'FAX'
```

SelectMatchingAk Method

Syntax

SelectListObj.**SelectMatchingAk** *FileObj*, *IndexName*, *IndexValue*

Description

This method creates a select list from the record IDs whose value matches that in a named secondary key field. The select list contains record IDs.

SelectListObj is a **SelectList** object.

FileObj identifies the file containing the secondary key index. It must be a **Dictionary** or **File** object returned by the **Session** object's **OpenFile** or **OpenDictionary** methods.

IndexName is the name of a secondary key index. This name must have been specified in a CREATE.INDEX command. If the index you specify does not exist, an empty select list is returned and the **LastRecordRead** property is set to **True**.

IndexValue is a value from the secondary key index. Records are selected when *IndexValue* matches the value of the indexed field. It is equivalent to the following SELECT command:

SELECT *FileName* WITH *IndexName* = *IndexValue*

The new select list overwrites the select list specified in *SelectListObj* and resets the select list pointer to the first record in the list.

The **Error** property is set if an error occurs.

This method corresponds to a subset of the functionality of the InterCall **ic_selectindex** function and the BASIC SELECTINDEX statement.

Example

```
SelectListObj.SelectMatchingAk FileObj, "COUNTRY", "DE"
```

SelectList Object Properties

These are the properties of the **SelectList** object:

- **Error** (read-only)
- **ExceptionOnError**
- **Identifier** (read-only)
- **LastRecordRead** (read-only)

Error Property

This read-only property contains the last error that occurred. If no error has occurred, it contains 0. For a list of error codes and their meanings, see Appendix A, "Error Codes and Replace Tokens." For more information about error conditions, see Error Handling in Chapter 2, "Using UniObjects."

***Note**: Once this **Error** property is set to anything other than 0, no other method can be used with the **SelectList** object until the error is processed.*

ExceptionOnError Property

Use this property to generate an exception that can be handled by the Visual Basic **On Error** statement. Set this property to **True** to force all errors to generate an exception. The default is **False**, causing only fatal errors to generate an exception.

***Note**: If you do not specify a value for the **ExceptionOnError** property, it inherits the value used by the **Session** object at the time the **SelectList** object was created.*

Identifier Property

This read-only property contains the InterCall select list identifier for a **SelectList** object. It is used only in applications that call an InterCall function that requires a select list identifier.

LastRecordRead Property

This read-only property contains **True** if:

- You attempt to read beyond the end of a select list
- A method results in an empty select list

Example Using the SelectList Object

```
Sub SampleSelect ()
   ' SampleSelect
   ' This sample illustrates the use of a SelectList object. It
opens a
   ' session, opens the file specified by the user, creates a
select list
   ' and reads the file's records displaying them in query boxes
until either
   ' the end of the file is reached or the user chooses not to
continue.
   Dim objFile As Object           ' Object variable for the file
   Dim objSelect As Object         ' Object Variable for the
select list
   Dim objSession As Object        ' Object Variable for the
session
   Dim strFile As String          ' File name on which to perform
select
   Const UVE_NOERROR = 0           ' From UVOAIF.TXT - no error
   ' The registered name of a database Session - Version 1
   Const UV_SESSION_OBJECT = "UniObjects.unioaifctrl"
   Const IDCANCEL = 2             ' Cancel button id
   Const MB_OKCANCEL = 1             ' OK-CANCEL message box style
   ' Create the session object and check it is OK
   Set objSession = CreateObject(UV_SESSION_OBJECT)
   If objSession Is Nothing Then
      ' NB. Error will be reported by Visual Basic
      Exit sub                        ' End the subroutine
   End If
   objSession.UserName = Input.Box ("User Name:","Login")
   objSession.Password = Input.Box ("Password:","Password")
   ' Connect to the server and check that it is active
   objSession.Connect
   If objSession.IsActive Then
      ' Open the file specified by the user
      strFile = InputBox("Enter file name:", "Select Sample")
      Set objFile = objSession.OpenFile(strFile)
      If objFile Is Nothing Then
         MsgBox "Cannot Open File: " & strFile & " (" &
         ? objSession.Error & ")"
         Exit Sub          ' End the subroutine
      End If
      ' Create a select list on the file - use select list 1
      Set objSelect = objSession.SelectList(1)
      If objSelect Is Nothing Then
         MsgBox "Cannot create select list 1 (" & objSession.Error
& ")"
         Set objFile = Nothing ' Tidy up
         Exit Sub
      End If
      ' perform the select on the open file and check for errors
      objSelect.Select objFile ' Establish a select list
      objFile.RecordId = objSelect.Next ' Get the first record ID
      Do While Not objSelect.LastRecordRead
         ' Read a record and check for errors
         objFile.Read
         If objFile.Error <> UVE_NOERROR Then
            MsgBox "Error reading from file (" & objFile.Error &
")"
            Exit Do
         End If
         ' Display record and check if we should continue
         If MsgBox(objFile.Record, MB_OKCANCEL) = IDCANCEL Then
            ' Cancel button was pressed
            Exit Do        ' Exit loop
         End If
         objFile.RecordId = objSelect.Next' Get the next record ID
      Loop
      ' Tell the user that the last record has been read
      If objSelect.LastRecordRead Then
         MsgBox "Last Record has been Read!"
      End If
      objFile.CloseFile    ' Close the file
      objSession.Disconnect ' Close session
    Else
      ' Check for Session errors - display message box with error
code
      ' No error means the user cancelled the connection dialog
box
      If objSession.Error <> UVE_NOERROR Then
         MsgBox "Unable to open connection:- " & objSession.Error
      End If
   End If
End Sub
```

Command Object

The **Command** object manages the running of a database command on the server. The execution of the command is controlled by the **Session** object. You specify the command you want to run through the **Command** property of the **Session** object. You can run only one command at a time during a session. For more information about using database commands, see Using Database Commands in Chapter 2, "Using UniObjects." The methods and properties that you can use with the **Command** object are described in the following sections.

Command Object Methods

These are the methods that you can use with the **Command** object:

- **Cancel**
- **Exec**
- **NextBlock**
- **Reply**

Cancel Method

Syntax

CmdObj.**Cancel**

Description

This method cancels all outstanding output from the executing command.

CmdObj is a **Command** object.

The **Cancel** method can only be called when the command has a UVS_REPLY or UVS_MORE status. If this method is called successfully, the **CommandStatus** property is set to UVS_COMPLETE.

Exec Method

Syntax

CmdObj.**Exec**

Description

This method executes the command contained in the **Text** property.

CmdObj is a **Command** object.

The **Response** property contains the returned response from executing the command. If an error occurred during the execution, the **Error** property contains the reason and the **Response** property can contain an error message produced by the executed command.

The **CommandStatus** property contains the current status of the command, that is whether it has completed, or is waiting for further input.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_execute** function and the BASIC EXECUTE statement.

Example

```
objSession.Command.Text = "LIST VOC"
objSession.Command.Exec ' Lists the VOC file
```

NextBlock Method

Syntax

CmdObj.**NextBlock**

Description

This method obtains the next block of the command response if the response was too long for the **Response** property. The size of the block defined in the **BlockSize** property.

CmdObj is a **Command** object.

The **Error** property is set if an error occurs.

Example

```
While CommandObj.Error = UVE_NOERROR ~
   & CommandObj.CommandStatus <> UVS_COMPLETE

   Select Case CommandObj.CommandStatus
   Case UVS_MORE:
      ' should not happen unless BlockSize is set to non-zero
      ' assume we have decided to process blocks of return
      ' data
      ProcessBlock CommandObj.Response
      CommandObj.NextBlock
```

Reply Method

Syntax

CmdObj.**Reply** *String*

Description

This method replies to a command execution. Use the **Reply** method to issue the correct response when an executed command requires user input. In this case, the **CommandStatus** property is set to UVS_REPLY.

CmdObj is a **Command** object.

String is the input to supply to the command.

Unless the command failed, the **Response** property contains the returned response from executing the command. If an error occurred, the **Error** property contains the reason. The **Response** property can also contain an error message produced by the executed command.

This method corresponds to the InterCall **ic_inputreply** function.

Example

```
Select Case CmdObj.CommandStatus
Case UVS_REPLY:
   CmdObj.Reply ReplyTxt.Text' send the reply held in the
            ' text box
ResponseTxt.Text = CmdObj.Response' put the prompt in the
            ' text box
```

Command Object Properties

These are the properties of the **Command** object:

- **AtSelected** (read-only)
- **BlockSize**
- **CommandStatus** (read-only)
- **Error** (read-only)
- **ExceptionOnError**
- **Response** (read-only)
- **SystemReturnCode** (read-only)
- **Text** (default)

AtSelected Property

On UniVerse systems this read-only property contains the number of records selected when the command has completed successfully, that is, when the **Command** object's **Status** is set to 0 or UVS_COMPLETE.

On UniData systems this read-only property contains information contained in SYSTEM (11).

BlockSize Property

This property determines the size, in bytes, of the buffer used to hold the contents of the **Response** property. The initial value is 0, which means no limit to the size of the buffer.

If you expect large quantities of output from a command, you can set this property to a manageable value and read the output into the **Response** property in blocks. You read successive blocks with the **NextBlock** method. In this case the **Status** property is set to UVS_MORE when the buffer is full, and when you access the **Response** property, the next block of command output is read from the server. For an example, see "Example Using the Command Object" on page 3-145.

***Note**: In a client/server application, running server commands that produce large quantities of output can decrease performance and increase network traffic. For more information about this, see Client/Server Design Considerations in Chapter 2, "Using UniObjects."*

CommandStatus Property

This read-only property contains a status code returned by a **Command** object method. Possible values are:

Value	Token	Meaning
0	UVS_COMPLETE	The command has finished successfully.
1	UVS_REPLY	The command is waiting for a reply.
2	UVS_MORE	More output to come, the command is waiting for a **NextBlock** method.

CommandStatus Status Code Values

Error Property

This read-only property contains the last error that occurred. If no error has occurred, it contains 0. For a list of error codes and their meanings, see Appendix A, "Error Codes and Replace Tokens." For more information about error conditions, see Error Handling in Chapter 2, "Using UniObjects."

***Note**: Once this* **Error** *property is set to anything other than 0, no other method can be used with the* **Command** *object until the error is processed.*

ExceptionOnError Property

Use this property to generate an exception that can be handled by the Visual Basic **On Error** statement. Set this property to **True** to force all errors to generate an exception. The default is **False**, causing only fatal errors to generate an exception.

***Note**: If you do not specify a value for the* **ExceptionOnError** *property, it inherits the value used by the* **Session** *object at the time the* **Command** *object was created.*

Response Property

This read-only property contains the response from a call to the **Exec** or **Reply** method.

SystemReturnCode Property

This read-only property contains the value of the @SYSTEM.RETURN.CODE code returned by the command.

Text Property

This property specifies the command to be executed. It is the default property of the **Command** object. For example:

```
CmdObj = "SELECT VOC"
```

is equivalent to:

```
CmdObj.Text = "SELECT VOC"
```

Example Using the Command Object

```
Sub SampleCommand ()
   '
   ' SampleCommand
   '
   ' To illustrate the use of the Command object. This sample
creates
   ' a Session object and connects to the server. It then requests
the
   ' name of a server file to create and enters into a user dialog
loop
   ' passing the server prompts onto the user, and the user
responses back
   ' to the server. Throughout this sample we use the Command
property from
   ' the Session object directly. We could equally well have
assigned a
   ' reference to it and used that, for example:
   '
   '   Dim objCmd As Object    ' Object variable for command
   '   Set objCmd = objSession.Command
   '   objCmd.Text = "CREATE.FILE " & "strResponse"
   '
   Dim objSession As object    ' Object variable for session
   Dim strResponse As String   ' Response from user
   '
   ' Token values from UVOAIF.TXT:
   '
   Const UVS_COMPLETE = 0      ' Execution complete
   Const UVS_REPLY = 1         ' Waiting for a reply
   Const UVS_MORE = 2          ' More data to come
   Const UVE_NOERROR = 0       ' No error
   ' The registered name of a database Session - Version 1
   Const UV_SESSION_OBJECT = "UniObjects.unioaifctrl"
   '
   ' Create a Session object to work with
   '  - This is a contrived sample, in a full application the
Session object
   '  - would typically be a Global variable that is set once
maybe in
   '  - response to a menu selection (e.g. Connect) on the main
form.
   '
   Set objSession = CreateObject(UV_SESSION_OBJECT)
   If objSession Is Nothing Then
      ' NB. Error will be reported by Visual Basic
      Exit Sub                     ' End the program
   End If
   objSession.UserName = Input.Box ("User Name:","Login")
   objSession.Password = Input.Box ("Password:","Password")
   '
   ' Establish a connection to the database server. By default it
   ' displays a dialog box to request the HostName and AccountPath
   ' property values.
   '
   objSession.Connect
   If objSession.IsActive Then
      strResponse = InputBox("Name of file to create:", "Command
Sample")
      '
      ' Now issue the command - and continue with user dialog
      ' We should only expect UVS_REPLY state when the server
issues a
      ' prompt and UVS_COMPLETE when the command has completed.
      '
      objSession.Command.Text = "CREATE.FILE " & strResponse
      objSession.Command.Exec ' Execute CREATE.FILE command
      Do While objSession.Command.Error = UVE_NOERROR
         Select Case objSession.Command.CommandStatus
         Case UVS_REPLY:
            '
            ' Command is awaiting a reply - display prompt
            ' and pipe users response back to the database.
            '
            strResponse = InputBox (objSession.Command.Response,
            ? "Command Sample")
            objSession.Command.Reply strResponse
         Case UVS_MORE:
            '
            ' This should not happen!
            ' The default BlockSize of 0 means we should get all
output
            ' UVS_MORE indicates there is more "Response" data to
come!
            ' but clean out any more data - throw it away
            '
            MsgBox "OOPS: Got a MORE status!!!"
            objSession.Command.NextBlock
         Case UVS_COMPLETE:
            '
            ' Normal end condition - when command has finished
            ' Display last response, and exit loop
            '
            MsgBox objSession.Command.Response
            Exit Do ' Normal end of loop
         Case Else:
            MsgBox "Bad status(" &
objSession.Command.CommandStatus & ")"
            Exit Do ' Not a lot we can do but clean up and go home
         End Select
      Loop
      '
      ' Check for command errors - and display them
      '
      If objSession.Command.Error <> UVE_NOERROR Then
```

```
         MsgBox "Error: (" & objSession.Command.Error & ")"
      End If
      '
      ' Disconnect the session
      '
      objSession.Disconnect   ' Disconnect session
   Else
      '
      ' Check for Session errors - display message box with error
code
      ' No error means the user cancelled the connection dialog
box
      '
      If objSession.Error <> UVE_NOERROR Then
         MsgBox "Unable to open connection:- " & objSession.Error
      End If
   End If
End Sub
```

Subroutine Object

The **Subroutine** object allows you to run a cataloged BASIC subroutine on the server. You define the subroutine using the **Subroutine** method of the **Session** object, for example:

```
Set SubrObj = SessObj.Subroutine("sub1",4)
```

For more information about subroutines, see Client/Server Design Considerations in Chapter 2, "Using UniObjects." The methods and properties that you can use with the **Subroutine** object are described in the following sections.

Subroutine Object Methods

These are the methods that you can use with the **Subroutine** object:

- **Call**
- **GetArg**
- **ResetArgs**
- **SetArg**

Call Method

Syntax

SubrObj.**Call**

Description

This method executes the cataloged BASIC subroutine identified by the **NAME** property.

SubrObj is a **Subroutine** object.

You supply any arguments the subroutine requires with the **SetArg** method before you use the **Call** method.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_subcall** function and the BASIC CALL statement.

Example

```
SubrObj.Call ' execute the subroutine
```

GetArg Method

Syntax

String = *SubrObj*.**GetArg**(*ArgNum*)

Description

This method retrieves argument values returned from the subroutine following a successful subroutine call.

String is the returned value.

SubrObj is a **Subroutine** object.

ArgNum is the number of the argument to retrieve. The first argument is 0.

If an argument has no value set, an empty string is returned. The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_getvalue** function.

Example

```
SubrObj.Call' execute the subroutine
Record = SubrObj.GetArg(1)' read the returned arg value
Code = SubrObj.GetArg(2)' read return CODE
' check return code.....
```

ResetArgs Method

Syntax

SubrObj.**ResetArgs**

Description

This method resets all arguments of the **Subroutine** object to their initial value.

SubrObj is a **Subroutine** object.

SetArg Method

Syntax

SubrObj.**SetArg** *ArgNum*, *ArgValue*

Description

This method sets the value of an argument for a subroutine.

SubrObj is a **Subroutine** object.

ArgNum is the number of the argument you are setting. The first argument is 0.

ArgValue is the value of the argument to be passed to the server subroutine. The argument is passed to the server before making the call. Any argument you do not specify with the **SetArg** method is passed as an empty string.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_setvalue** function.

Example

```
SubrObj.SetArg 0, ActionCode
```

Subroutine Object Properties

These are the properties of the **Subroutine** object:

- **Error** (read-only)
- **ExceptionOnError**
- **RoutineName** (read-only)

Error Property

This read-only property contains the last error that occurred. If no error has occurred, it contains 0. For a list of error codes and their meanings, see Appendix A, "Error Codes and Replace Tokens." For more information about error conditions, see Error Handling in Chapter 2, "Using UniObjects."

Note: *Once this* **Error** *property is set to anything other than 0, no other method can be used with the* **Subroutine** *object until the error is processed.*

ExceptionOnError Property

Use this property to generate an exception that can be handled by the Visual Basic **On Error** statement. Set this property to **True** to force all errors to generate an exception. The default is **False**, causing only fatal errors to generate an exception.

Note: *If you do not specify a value for the* **ExceptionOnError** *property, it inherits the value used by the* **Session** *object at the time the* **Subroutine** *object was created.*

RoutineName Property

This read-only property contains the name of the subroutine set by the **Subroutine** method that creates the object.

Example Using the Subroutine Object

```
Sub SampleSubroutine ()
   '
   ' Sample Subroutine
   '
   ' This sample illustrates the use of the Subroutine object.
   ' To run this sample code a subroutine must first be created 
and
   ' cataloged on the server. If you do not already have a 
suitable
   ' subroutine, use ED to enter the following program into a type 
1
   ' file called SAMPLESUBR.
   '
   '   SUBROUTINE SAMPLESUBR(ARG1, ARG2, ARG3)
   '       ARG2 = ARG3
   '       ARG1 = "It Worked"
   '       ARG3 = 0
   '   RETURN
   '   END
   ' Compile and catalog the subroutine as follows:
   '       BASIC <file> SAMPLESUBR
   '       CATALOG <file> SAMPLESUBR
   ' Where <file> is the name of the type-1 file.
   Dim objSession As Object    ' Object variable for session
   Dim objSubroutine As Object ' Object variable for Subroutine 
object
   Dim strArg As String        ' Argument setting
   Dim iArg As Integer         ' Argument index

   Const UVE_NOERROR = 0       ' From UVOAIF.TXT - no error
   ' The registered name of a database Session - Version 1
   Const UV_SESSION_OBJECT = "UniObjects.unioaifctrl"
   ' Create a Session object to work with
   '   - This is a contrived sample, in a full application the
   '   - Session object would normally be a Global variable
   '   - that is set once, for example, in response to a menu 
selection
   '   - such as Connect on the main form.
   Set objSession = CreateObject(UV_SESSION_OBJECT)
   If objSession Is Nothing Then
      Exit Sub                     ' End the program
   End If
   objSession.UserName = Input.Box ("User Name:","Login")
   objSession.Password = Input.Box ("Password:","Password")

   '
   ' Establish a connection to the database server. By default it
   ' displays a dialog box to request the HostName and AccountPath
   ' property values.
   '
   objSession.Connect
   If objSession.IsActive Then
      ' Create object for Subroutine with 3 arguments
      Set objSubroutine = objSession.Subroutine("SAMPLESUBR", 3)
      If objSubroutine Is Nothing Then
         MsgBox "Error creating Subroutine object: (" &
         ? objSession.Error & ")"
      Else
         ' Set each of the arguments - unless user cancels or
enters ""
         For iArg = 0 To 2
            strArg = InputBox ("Enter Arg" & iArg & ":",
"Subroutine
            ? Sample")
            If strArg <> "" Then
               objSubroutine.SetArg iArg, strArg
            End If
         Next iArg
         ' Call the subroutine and display the args
         objSubroutine.Call
         If objSubroutine.Error <> UVE_NOERROR Then
            MsgBox "Error: (" & objSubroutine.Error & ")"
         Else
            MsgBox "Args: " & objSubroutine.GetArg(0) & "," &
            ? objSubroutine.GetArg(1) & "," &
objSubroutine.GetArg(2)
         End If
      End If
      '
      ' Close the session
      '
      objSession.Disconnect
   Else
      ' Check for Session errors - display message box with error
code
      ' No error means the user cancelled the connection dialog
box
      If objSession.Error <> UVE_NOERROR Then
         MsgBox "Unable to open connection:- " & objSession.Error
      End If
   End If
End Sub
```

Transaction Object

The **Transaction** object is available from the **Session** object. The **Transaction** object provides methods to start, commit, and roll back transactions for a session. If a session is closed while transactions are active, they are rolled back by the server. Only one transaction can be active at a time.

***Note**: Transaction processing is different on UniVerse and UniData systems. See your server documentation for detailed information about transaction processing.*

The method and properties you can use with the**Transaction** object are described in the following sections.

Transaction Object Methods

These are the methods that you can use with the **Transaction** object:

- **Commit**
- **IsActive**
- **Rollback**
- **Start**

Commit Method

Syntax

TransactionObj.**Commit**

Description

This method commits the active transaction. If it is a nested transaction, the parent transaction becomes active and the transaction level is decremented.

TransactionObj is a **Transaction** object.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_trans** function and the BASIC COMMIT statement.

IsActive Method

Syntax

Bool = *TransactionObj*.**IsActive**

Description

This method determines if a transaction is active. This method returns **True** if the transaction is active; otherwise it is **False**.

Bool is set to **True** or **False** to indicate whether the transaction is active.

TransactionObj is a **Transaction** object.

The **Error** property is set if an error occurs.

Rollback Method

Syntax

TransactionObj.**Rollback**

Description

This method rolls back the active transaction. If this is a nested transaction, the parent transaction becomes active and the transaction level is decremented.

TransactionObj is a **Transaction** object.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_trans** function and the BASIC ROLLBACK statement.

Start Method

Syntax

TransactionObj.**Start**

Description

This method begins a new transaction. This transaction could be nested. If a transaction is already active, the nested transaction becomes active and the transaction level is decremented.

TransactionObj is a **Transaction** object.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_trans** function and the BASIC TRANSACTION statement.

Transaction Object Properties

These are the properties of the **Transaction** object:

- **Error** (read-only)
- **ExceptionOnError**
- **Level** (read-only)

Error Property

This read-only property contains the last error that occurred. If no error has occurred, it contains 0. For a list of error codes and their meanings, see Appendix A, "Error Codes and Replace Tokens." For more information about error conditions, see Error Handling in Chapter 2, "Using UniObjects."

ExceptionOnError Property

Use this property to generate an exception that can be handled by the Visual Basic **On Error** statement. Set this property to **True** to force all errors to generate an exception. The default is **False**, causing only fatal errors to generate an exception.

Level Property

This read-only property contains the current transaction level.

NLSLocale Object (UniVerse Only)

You can use the **NLSLocale** object only with UniVerse databases.

The **NLSLocale** object defines and manages the locale categories in use. The five locale categories are Time, Numeric, Monetary, Ctype, and Collate. The **NLSLocale** object allows these five names to be supplied as a single **DynamicArray** object, with five fields containing the relevant locale name. Locale names are derived from the client system and a defaultable locale identifier. The **NLSLocale** object is available from the **Session** object. If NLS is disabled on the server, the **NLSLocale** object is not available and returns NULL.

The methods and properties you can use with the **NLSLocale** object are described in the following sections.

NLSLocale Object Method

The only method you can use with the **NLSLocale** object is **SetName**.

SetName Method

Syntax

NLSLocaleObj.**SetName** *DynArrayObj* [, *index*]

Description

The **SetName** method sets the NLS locale category.

NLSLocaleObj is an **NLSLocale** object.

DynArray is a **DynamicArray** object containing either one or all five elements.

- If the dynamic array contains five elements, each value sets the corresponding locale category.
- If the dynamic array contains one element:
    - If no index is specified, all five locale categories are set to the same value.
    - If an index is specified, only the category specified in the index is set.

index defines the correspondence between the array and the locale categories as follows:

Index	Category	Token
1	Time	UVT_NLS_TIME
2	Numeric	UVT_NLS_NUMERIC
3	Monetary	UVT_NLS_MONETARY
4	Ctype	UVT_NLS_CTYPE
5	Collate	UVT_NLS_COLLATE

Index Definitions

If the method succeeds, the values of the **ServerNames** and **ClientNames** properties are updated.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_set_locale** function.

NLSLocale Object Properties

These are the properties of the **NLSLocale** object:

- **ClientNames** (read-only)
- **Error** (read-only)
- **ExceptionOnError**
- **ServerNames** (read-only)

ClientNames Property

This read-only property contains a dynamic array of the locale names requested by the client. This property returns NULL if the **SetName** method has not been called.

Error Property

This read-only property contains the last error that occurred. If no error has occurred, it contains 0. For a list of error codes and their meanings, see Appendix A, "Error Codes and Replace Tokens." For more information about error conditions, see Error Handling in Chapter 2, "Using UniObjects."

***Note**: Once this **Error** property is set to anything other than 0, no other method can be used with the **NLSLocale** object until the error is processed.*

ExceptionOnError Property

Use this property to generate an exception that can be handled by the Visual Basic **On Error** statement. Set this property to **True** to force all errors to generate an exception. The default is **False**, causing only fatal errors to generate an exception.

ServerNames Property

This read-only property contains the locale names reported by the server as a **DynamicArray** object. This property is updated every time a request for its value is received.

This property corresponds to the InterCall **ic_get_locale** function.

NLSMap Object (UniVerse Only)

You can use the **NLSMap** object only with UniVerse databases.

NLS maps are used by the server to determine which map to use for a client’s string data.

The **NLSMap** object is available from the **Session** object. If NLS is disabled on the server, the **NLSMap** object is not available and returns NULL.

Note*: Do not redefine NLS maps while an application has open sessions to a server.*

The method and properties you can use with the **NLSMap** object are described in the following sections.

NLSMap Object Method

The only method you can use with the **NLSMap** object is **SetName**.

SetName Method

Syntax

NLSMapObj.**SetName**(*MapName*)

Description

This method sets the map to be used on the server. When the name has been changed successfully, the **SetName** method updates the values of the **ServerName** and **ClientName** properties of the **NLSMap** object, and all system delimiter properties of the **Session** object.

NLSMapObj is an **NLSMap** object.

MapName is the name of the map requested.

The **Error** property is set if an error occurs.

This method corresponds to the InterCall **ic_set_map** function.

NLSMap Object Properties

These are the properties of the **NLSMap** object:

- **ClientName** (read-only)
- **Error** (read-only)
- **ExceptionOnError**
- **ServerName** (read-only)

ClientName Property

This read-only property contains the name of the map requested by the client. On the server it is mapped through the NLS.CLIENT.MAPS file to the name reported by the **ServerName** property.

Error Property

This read-only property contains the last error that occurred. If no error has occurred, it contains 0. For a list of error codes and their meanings, see Appendix A, "Error Codes and Replace Tokens." For more information about error conditions, see Error Handling in Chapter 2, "Using UniObjects."

***Note**: Once the **Error** property is set to anything other than 0, no other method can be used with the **NLSMap** object until the error is processed.*

ExceptionOnError Property

Use this property to generate an exception that can be handled by the Visual Basic **On Error** statement. Set this property to **True** to force all errors to generate an exception. The default is **False**, causing only fatal errors to generate an exception.

ServerName Property

This read-only property contains the name of the map that the server reports is currently loaded into shared memory. This property is updated after every **SetName** method update.

Chapter

Distributing Your Application

4

When you have completed and tested an application that uses UniObjects, you need to make a setup file that includes the DLLs your application uses. You can do this through the Visual Basic SetupWizard program, described in *Microsoft Visual Basic Programmer's Guide*.

The SetupWizard program automatically includes all the Visual Basic DLLs used in your application, but in addition you must specify:

- Microsoft OLE Automation DLLs
- UniObjects DLLs

You add the full paths for the DLLs to the list of files generated by the SetupWizard. In some cases you also need to specify a destination for the file on your customer's computer. You do this through the Visual Basic Setup Toolkit, as described in *Microsoft Visual Basic Programmer's Guide*. For full details of the files you need to specify, see the online file REDIST/REDIST.TXT in your UniDK installation directory.

Registering the OLE Control

As part of the setup program for your application, you must register the OLE control by running the registration server program. This is a redistributable Microsoft program available in the REDIST directory on the installation CD. Use the appropriate command line for either 16-bit or 32-bit applications, as described in the online file REDIST/REDIST.TXT in your UniDK installation directory.

Appendix

A

Error Codes and Replace Tokens

UniObjects provides replace tokens for error codes and global constants that may be useful in your application. They are contained in the file called UNIDK\INCLUDE\UVOAIF.TXT. You can add this file to a Visual Basic application through the **Add File** option on the File menu.

Note*:UVOAIF.TXT is a generic file used by client programs accessing the database. This appendix describes only those tokens that are relevant to UniObjects.*

Error Codes

These are the error codes that can be returned to a UniObjects application, together with their replace tokens:

Code	Token	Description
0	UVE_NOERROR	No error
14002	UVE_ENOENT	No such file or directory
14005	UVE_EIO	I/O error
14009	UVE_EBADF	Bad file number
14012	UVE_ENOMEM	No memory available
14013	UVE_EACCES	Permission denied
14022	UVE_EINVAL	Invalid argument
14023	UVE_ENFILE	File table overflow
14024	UVE_EMFILE	Too many open files
14028	UVE_ENOSPC	No space left on device
14551	UVE_NETUNREACH	Network is unreachable
22004	UVE_LRR	The last record in the select list has been read
22005	UVE_NFI	Not a file identifier
30001	UVE_RNF	Record not found
30002	UVE_LCK	This file or record is locked by another user
30095	UVE_FIFS	The file ID is incorrect for the current session
30097	UVE_SELFAIL	The select operation failed
30098	UVE_LOCKINVALID	The task lock number specified is invalid

Error Codes

Code	Token	Description
30099	UVE_SEQOPENED	The file was opened for sequential access and you have attempted hashed access
30100	UVE_HASHOPENED	The file was opened for hashed access and you have attempted sequential access
30101	UVE_SEEKFAILED	The operation using FileSeek failed
30103	UVE_INVALIDATKEY	The key used to set or retrieve an @variable is invalid
30105	UVE_UNABLETOLOADSUB	Unable to load the subroutine on server
30106	UVE_BADNUMARGS	Too few or too many arguments supplied to the subroutine
30107	UVE_SUBERROR	The subroutine failed to complete successfully
30108	UVE_ITYPEFTC	The I-type operation failed to complete correctly
30109	UVE_ITYPEFAILEDTOLOAD	The I-type failed to load
30110	UVE_ITYPENOTCOMPILED	The I-type has not been compiled
30111	UVE_BADITYPE	This is not an I-type or the I-type is corrupt
30112	UVE_INVALIDFILENAME	The specified file name is null
30113	UVE_WEOFFAILED	WEOFSEQ failed
30114	UVE_EXECUTEISACTIVE	An EXECUTE is currently active on the server
30115	UVE_EXECUTENOTACTIVE	No EXECUTE is currently active on the server
30125	UVE_CANT_ACCESS_PF	Cannot access part files
30126	UVE_FAIL_TO_CANCEL	Failed to cancel an execute

Error Codes (Continued)

Code	Token	Description
30127	UVE_INVALID_INFO_KEY	Bad key for the HostType property
30128	UVE_CREATE_FAILED	The creation of a sequential file failed
30129	UVE_DUPHANDLE_FAILED	Failed to duplicate a pipe handle
31000	UVE_NVR	No VOC record
31001	UVE_NPN	No path in VOC record
39101	UVE_NODATA	The server is not responding
39119	UVE_AT_INPUT	The server is waiting for input to a command
39120	UVE_SESSION_NOT_OPEN	The session is not open
39121	UVE_UVEXPIRED	The database license has expired
39122	UVE_CSVERSION	The client and the server are not running at the same release level
39123	UVE_COMMSVERSION	The client or server is not running at the same release level as the communications support
39124	UVE_BADSIG	You are trying to communicate with the wrong client or server
39125	UVE_BADDIR	The directory does not exist, or is not a database account
39127	UVE_BAD_UVHOME	Cannot find the UV account directory
39128	UVE_INVALIDPATH	An invalid pathname was found in the UV.ACCOUNT file
39129	UVE_INVALIDACCOUNT	The account name supplied is not an account
39130	UVE_BAD_UVACCOUNT_FILE	The UV.ACCOUNT file could not be found or opened
39131	UVE_FTA_NEW_ACCOUNT	Failed to attach to the specified account

Error Codes (Continued)

Code	Token	Description
39134	UVE_ULR	The user limit has been reached on the server
39135	UVE_NO_NLS	NLS is not available
39136	UVE_MAP_NOT_FOUND	NLS map not found
39137	UVE_NO_LOCALE	NLS locale support not available
39138	UVE_LOCALE_NOT_FOUND	NLS locale not found
39139	UVE_CATEGORY_NOT_FOUND	NLS locale category not found
39201	UVE_SR_SOCK_CON_FAIL	The server failed to connect to the socket
39210	UVE_SR_SELECT_FAIL	The server failed to select on input channel. When you see this error, you must quit and reopen the session.
39211	UVE_SR_SELECT_TIMEOUT	The select has timed out
40001	UVE_INVALIDFIELD	Pointer error in a sequential file operation
40002	UVE_SESSIONEXISTS	The session is already open
40003	UVE_BADPARAM	An invalid parameter was passed to a subroutine
40004	UVE_BADOBJECT	An incorrect object was passed
40005	UVE_NOMORE	The NextBlock method was used but there are no more blocks to pass.
40006	UVE_NOTATINPUT	The Reply method can only be used when the Response property is set to UVS_REPLY
40007	UVE_INVALID_DATAFIELD	The dictionary entry does not have a valid TYPE field
40008	UVE_BAD_DICTIONARY_ ENTRY	The dictionary entry is invalid

Error Codes (Continued)

Code	Token	Description
40009	UVE_BAD_CONVERSION_ DATA	Unable to convert the data in the field
80011	UVE_BAD_LOGINNAME	The login name provided is incorrect
80019	UVE_BAD_PASSWORD	The password has expired
80144	UVE_ACCOUNT_EXPIRED	The account has expired
80147	UVE_RUN_REMOTE_FAILED	Unable to run as the given user
80148	UVE_UPDATE_USER_FAILED	Unable to update user details
81001	UVE_RPC_BAD_CONNECTION	The connection is bad, and may be passing corrupt data.
81002	UVE_RPC_NO_CONNECTION	The connection is broken
81005	UVE_RPC_WRONG_VERSION	The version of the UniRPC on the server is different from the version on the client.
81007	UVE_RPC_NO_MORE_ CONNECTIONS	No more connections available
81009	UVE_RPC_FAILED	The UniRPC failed
81011	UVE_RPC_UNKNOWN_HOST	The host name specified is not valid, or the host is not responding
81014	UVE_RPC_CANT_FIND_ SERVICE	Cannot find the service in the *unirpcservices* file
81015	UVE_RPC_TIMEOUT	The connection has timed out
81016	UVE_RPC_REFUSED	The connection was refused as the UniRPC daemon is not running
81017	UVE_RPC_SOCKET_INIT_ FAILED	Failed to initialize the network interface
81018	UVE_RPC_SERVICE_PAUSED	The UniRPC service has been paused
81019	UVE_RPC_BAD_TRANSPORT	An invalid transport type has been used

Error Codes (Continued)

Code	Token	Description
81020	UVE_RPC_BAD_PIPE	Invalid pipe handle
81021	UVE_RPC_PIPE_WRITE_ERROR	Error writing to pipe
81022	UVE_RPC_PIPE_READ_ERROR	Error reading from pipe

Error Codes (Continued)

@Variables

The following tokens represent BASIC @variables:

Token	Value	BASIC @variable
AT_LOGNAME	1	@LOGNAME
AT_PATH	2	@PATH
AT_USERNO	3	@USERNO
AT_WHO	4	@WHO
AT_TRANSACTION	5	@TRANSACTION
AT_DATA_PENDING	6	@DATA.PENDING
AT_USER_RETURN_CODE	7	@USER.RETURN.CODE
AT_SYSTEM_RETURN_CODE	8	@SYSTEM.RETURN.CODE
AT_NULL_STR	9	@NULL.STR
AT_SCHEMA	10	@SCHEMA

BASIC @variables

BlockingStrategy Property Values

The following tokens can be used to set the **BlockingStrategy** property:

Token	Value	Meaning
WAIT_ON_LOCKED	1	If the record is locked, wait until it is released.
RETURN_ON_LOCKED	2	Return a value to the **Status** property to indicate the state of the lock. This is the default. The values that can be returned are shown in "Lock Status Values" on page A-11.

BlockingStrategy Tokens

CommandStatus Property Values

The following tokens represent the possible values of the **CommandStatus** property:

Token	Value	Meaning
UVS_COMPLETE	0	Execution of the command is complete.
UVS_REPLY	1	The command is waiting for a reply.
UVS_MORE	2	More output to come from the command; the command is waiting for a **NextBlock** method.

CommandStatus Values

Lock Status Values

The following tokens represent the values returned to the **Status** property to indicate the state of a lock:

Token	Value	Meaning
	4	This user holds the shared file lock
LOCK_MY_FILELOCK	3	This user holds the exclusive file lock.
LOCK_MY_READU	2	This user holds the READU lock.
LOCK_MY_READL	1	This user holds the READL lock.
LOCK_NO_LOCK	0	The record is not locked.
LOCK_OTHER_READL	–1	Another user holds the READL lock.
LOCK_OTHER_READU	–2	Another user holds the READU lock.
LOCK_OTHER_FILELOCK	–3	Another user holds the exclusive file lock.
	–4	Another user holds the shared file lock.

Lock Status Values

Locking Strategy Values

The following tokens are used with the **LockStrategy** and **DefaultLockStrategy** properties:

Token	Value	Meaning
EXCLUSIVE_UPDATE	1	Sets a READU lock.
SHARED_READ	2	Sets a READL lock.
NO_LOCKS	0	No locking. This is the default.

LockStrategy and DefaultLockStrategy Tokens

Registered Object Names

The following tokens specify object names to the **CreateObject** method:

Token	Object
UV_SESSION_OBJECT	UniObjects.Unioiafctrl
UV_DARRAY_OBJECT	UniObjects.UniDynArray

CreateObject Tokens

Relative Position Parameter Values

The following tokens indicate the relative position parameter values that are used with the **FileSeek** method of the **SequentialFile** object:

Token	Value	Meaning
UVT_START	0	Start of file
UVT_CURR	1	Current position
UVT_END	2	End of file

FileSeek Tokens

ReleaseStrategy Property Values

The following tokens set the **ReleaseStrategy** property:

Token	Value	Meaning
WRITE_RELEASE	1	Releases the lock when the record is written.
READ_RELEASE	2	Releases the lock when the record is read.
EXPLICIT_RELEASE	4	Maintains locks as specified by the **LockStrategy** property. Releases the locks only with the **Unlock-Record** method.
CHANGE_RELEASE	8	Releases the lock when a new value is assigned to the **RecordId** property. This value is additive and can be combined with any of the other values.

ReleaseStrategy Tokens

Select Lists

The token that defines the highest select list number (10) is as follows:

IC_MAX_SELECT_LIST

System Delimiters

The following tokens represent database system delimiters:

Token	Character Value	Meaning
I_IM	255	Item mark
I_FM	254	Field mark
I_VM	253	Value mark
I_SM	252	Subvalue mark
I_TM	251	Text mark

System Delimiter Tokens

Appendix

The Demo Application

B

This appendix describes the demonstration application that is supplied with UniObjects on your installation CD. The demonstration application is part of a simple order entry application. The appendix starts by describing where to find the demo and what it does, and then explains the programming techniques that were used to build it.

Installing the Demo

All the files for the demonstration application can be found in the SAMPLES\ORDERS subdirectory of your UniDK installation directory. This directory has two subdirectories:

- HOSTSAVE contains files that are used to set up the server account needed by the demonstration application.
- CLIENT contains Visual Basic source files and an executable file for the demonstration application. The source files in this directory are described in "Code Structure" on page B-7.

***Note**: The database should be running on the server before you start.*

To install the demo, follow these steps:

1. *On the server*: Create a directory to hold the database account used in the application.
2. Copy the contents of the SAMPLES\ORDERS\HOSTSAVE subdirectory from the UniDK installation directory to the newly created directory. The files in the HOSTSAVE directory are in Windows format. For UNIX servers, transfer the files using FTP in text mode to preserve the formatting correctly.
3. Make the directory into a database account. How you do this depends on the type of server you are using.
   - **On a Windows server:** Access the server through Telnet. At the prompt, enter the name of the directory you have just created.
   - **On a UNIX server:** Change to the directory you just created and enter **`uv`** to invoke UniVerse.
   - You are asked if you want to set up the database account. Enter **`y`** at this prompt.
   - Enter `0` to specify IDEAL flavor.
4. At the database prompt, enter the following:

```
>COPY FROM &UFD& TO VOC LOAD.PAR
>LOAD.PAR
```

5. *On the client*, use Notepad or another editor to edit the ORDERS.INI file in the SAMPLES\ORDERS\CLIENT subdirectory of the UniDK installation directory as follows. Change the `HostName=` and `AccountPath=` lines to specify your server and the new database account that you have created. *For UNIX servers only*, change the `UserName=` line to specify the user name that you want the demo to run as.

***Note**: If the program cannot retrieve the server name and account path from the ORDERS.INI file, it prompts you to enter the information every time you run the program.*

To run the demo, go into File Manager, select SAMPLES\ORDERS\CLIENT, and double-click the ORDERS.EXE file.

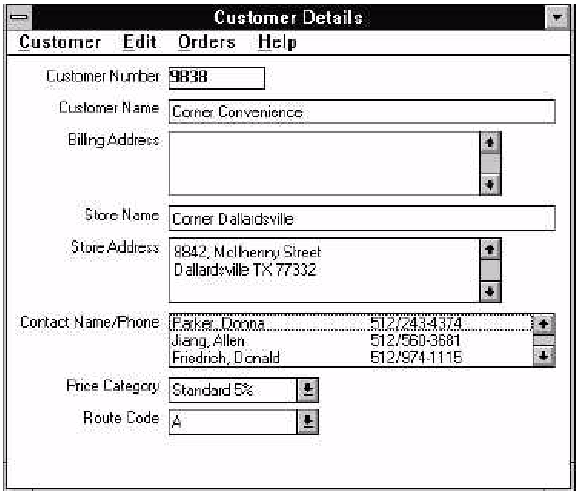
What It Does

This application would be used by a wholesale supplier to take orders from customers over the phone. This process can be divided into three activities:

- Selecting a customer
- Entering a new order
- Viewing details of existing orders

Selecting a Customer

The program displays a splash screen to show that something is happening while it is getting started. Once this has finished, you see an empty Customer form. At this point the only thing you can do is to choose **Open** from the Customer menu; all the other options are either dimmed or display a `not implemented` message. **Open** displays a list of customers, from which you can select one.



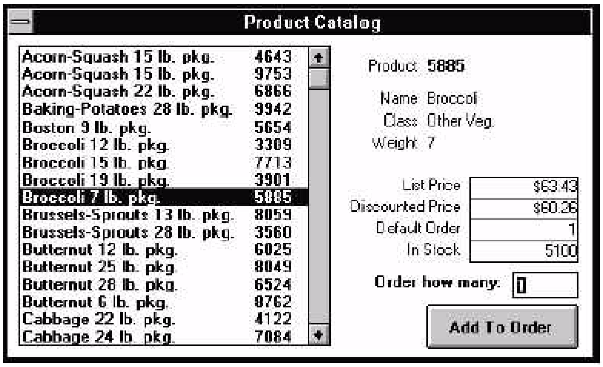
Customer Number	9838
Customer Name	Corner Convenience
Billing Address	
Store Name	Corner Dallardsville
Store Address	8842, McIlhenny Street Dallardsville TX 77332
Contact Name/Phone	Parker, Donna 512/243-4374 Jiang, Allen 512/560-3681 Friedrich, Donald 512/974-1115
Price Category	Standard 5%
Route Code	A

You can edit the contents of this form, but this will not affect the file, since the **Save** option is not implemented. Try selecting Customer 9838, Corner Dallardsville, as there are already several orders for this customer in the database.

When a customer record is displayed, the Orders menu becomes available.

Entering a New Order

You enter a new order by choosing **New** from the Orders menu. This displays a blank Customer Order form and also the Product Catalog window.



Acorn-Squash 15 lb. pkg.	4643
Acorn-Squash 15 lb. pkg.	9753
Acorn-Squash 22 lb. pkg.	6866
Baking-Potatoes 28 lb. pkg.	9942
Boston 9 lb. pkg.	5654
Broccoli 12 lb. pkg.	3309
Broccoli 15 lb. pkg.	7713
Broccoli 19 lb. pkg.	3901
Broccoli 7 lb. pkg.	5885
Brussels-Sprouts 13 lb. pkg.	8059
Brussels-Sprouts 28 lb. pkg.	3560
Butternut 12 lb. pkg.	6025
Butternut 25 lb. pkg.	8049
Butternut 28 lb. pkg.	6524
Butternut 6 lb. pkg.	0762
Cabbage 22 lb. pkg.	4122
Cabbage 24 lb. pkg.	7084

Product 5885

Name Broccoli
Class Other Veg.
Weight 7

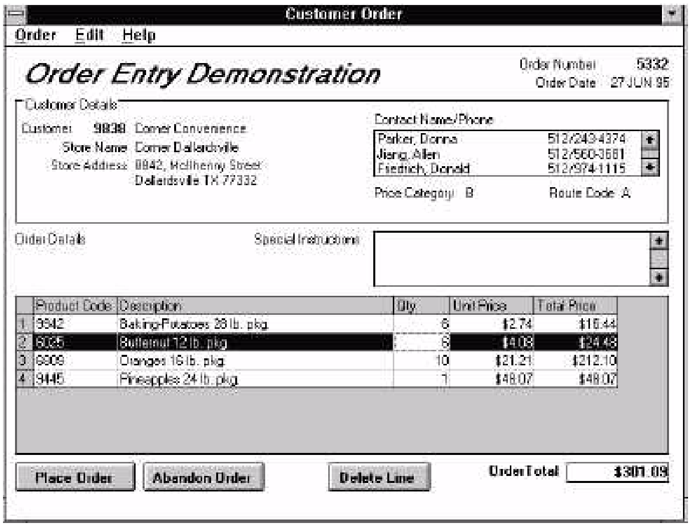
List Price	$63.43
Discounted Price	$60.26
Default Order	1
In Stock	5100

Order how many: 0

To add items to the order, go to the Product Catalog, select a product, change the quantity, if required, and click **Add To Order**. Keep doing this until you have several items on the order. To complete the order, click in the Customer Order form. You can enter special instructions in the appropriate box, or you can edit the items ordered. The only fields you can change are the quantity and the unit price (you may want to discount the price for certain customers). To edit a field, you can double-click the cell, or move to the cell with the arrow keys, and either press **F2** to edit the entry or just start typing to overwrite it.

Try entering letters in the quantity field. The program rejects your entry, forcing you to either retry the edit or cancel it. If you have ordered the wrong product, you must delete the line with the **Delete Line** button and go back to the Product Catalog to select the right one.

Once you are happy with the order, click **Place Order** to file it. It is now in the database. The next example shows a completed Customer Order form.



Order Entry Demonstration

Order Number 5332
Order Date 27 JUN 95

Customer Details

Customer 9838 Corner Convenience
Store Name Corner Dallardsville
Store Address 8842, McIlhenny Street
Dallardsville TX 77332

Contact Name/Phone

Parker, Donna	512/243-4374
Jiang, Allen	512/560-3681
Friedrich, Donald	512/974-1115

Price Category B   Route Code A

Order Details   Special Instructions

	Product Code	Description	Qty	Unit Price	Total Price
1	9942	Baking-Potatoes 28 lb. pkg.	6	$2.74	$16.44
2	6035	Butternut 12 lb. pkg.	6	$4.08	$24.48
3	6909	Oranges 16 lb. pkg.	10	$21.21	$212.10
4	9445	Pineapples 24 lb. pkg.	1	$48.07	$48.07

Order Total $301.09

Viewing Details of Existing Orders

To see the order that you have just entered, go back to the Customer Details form, and choose either **List Outstanding** or **List All** from the Orders menu. (In this demonstration program these two options give the same results.) This displays a dialog box with a list of orders. When you select one, it appears in read-only format on an Order form.

Code Structure

The code for the demo application is held in 12 modules, as shown in the following table. All these files are in the SAMPLES\ORDERS\CLIENT directory. They are text files, so that you can print them or inspect them using Notepad or another editor. You can also browse the application with Visual Basic.

Category	Module Name	Description
Data entry forms:	CUSTOMER.FRM	Manages the Customer Details form
	ORDER.FRM	Manages the Order Details form
Dialog boxes:	GETCUST.FRM	Selects a customer
	ORDRLIST.FRM	Selects an existing order
	PASSWORD.FRM	Prompts user for password
Other forms:	PRODUCTS.FRM	Manages the Product Catalog
	SPLASH.FRM	Splash screen for the start of the application
Subroutines:	EDITGRID.BAS	Subroutines to manage editing of data displayed in a Microsoft Grid control
	ORDERSUB.BAS	Application-specific subroutines to help manage the Order Details form
	UNI_UTILS.BAS	General data-handling subroutine
Other files:	ORDERS.VBP	Visual Basic Project file
	ORDERDEF.BAS	Constant definitions
	ORDERGBL.BAS	Global variable declarations

Modules Used in the Demo Application

Program Initialization

While the program is starting, it displays a splash screen, that is, a form that shows the stages of progress. The code that does this is in the **Form_Load** procedure of the **frmCustomer** form contained in the file CUSTOMER.FRM. The stages shown are:

- Connecting to the server
- Opening files
- Loading customer list and loading product catalog

Connecting to the Server

The program finds the server name and account path from an initialization file, and then uses the **Connect** method. Reading the initialization file is discussed in Using an Initialization File on page 11. If the program cannot find the server name and account path, it prompts for them.

```
' Establish the session parameters
   Rslt = SetSessionDetails()
If Not (Rslt) Then
   frmConnect.Show 1
   If ReturnStatus = IDCANCEL Then
      End
   End If
End If
' Connect to the server
frmSplash.Label1.Caption = "Connecting to server..."
frmSplash.Refresh
Screen.MousePointer = HOURGLASS
Rslt = UVSession.Connect()
Screen.MousePointer = DEFAULT
If Not (Rslt) Then
   If UVSession.Error <> 0 Then
      DisplayError "Error in Connect!", UVSession.Error,
MB_ICONSTOP
      End
   End If
End If

If Not (UVSession.IsActive) Then
   DisplayError "Session is not active!", 0, MB_ICONSTOP
   End
End If
```

Opening Files

The program opens four files: the CUSTOMERS, PRODUCTS, and ORDERS data files, and the dictionary of the ORDERS file.

```
' Open the application files
frmSplash.Label1.Caption = "Opening files..."
frmSplash.Refresh
Set CustomerFile = UVSession.OpenFile(CUSTOMER_FILE_NAME)
If UVSession.Error <> 0 Then
   DisplayError "Could not open " & CUSTOMER_FILE_NAME,
   ?UVSession.Error, MB_ICONSTOP
   End
End If

Set ProductFile = UVSession.OpenFile(PRODUCT_FILE_NAME)
If UVSession.Error <> 0 Then
   DisplayError "Could not open " & PRODUCT_FILE_NAME,
   ?UVSession.Error,MB_ICONSTOP
    End
End If

Set OrderFile = UVSession.OpenFile(ORDER_FILE_NAME)
If UVSession.Error <> 0 Then
   DisplayError "Could not open " & ORDER_FILE_NAME,
UVSession.Error,
   ?MB_ICONSTOP
   End
End If
Set OrderDict = UVSession.OpenDictionary(ORDER_FILE_NAME)
If UVSession.Error <> 0 Then
   DisplayError "Could not open " & ORDER_FILE_NAME & "
dictionary",
   ?UVSession.Error, MB_ICONSTOP
    End
End If
```

Loading the Customer List and Product Catalog

The Select Customer dialog box and the Product Catalog both need to display a list of the records in a file. Rather than wait for this data to be loaded each time the form is shown, the program preloads the data at the beginning.

```
' Load frmGetCustomer, to pre-load its list box with customer
names
frmSplash.Label1.Caption = "Loading customer list..."
frmSplash.Refresh
Load frmGetCustomer

' Same thing for frmProducts
frmSplash.Label1.Caption = "Loading product catalog..."
frmSplash.Refresh
Load frmProducts
```

Using an Initialization File

Setting up the server name and account path from an initialization file is done by the **SetSessionDetails** subroutine, also found in the file CUSTOMER.FRM. It uses the **GetPrivateProfileString** Windows API function.

```
'*****************************************************************
*******
' SetSessionDetails: set the host name and account path of the
session
' according to the configuration file contents. If there is an
error,
' attributes are not set, and the user will be prompted.
'*****************************************************************
*******
Function SetSessionDetails () As Integer
Const IN_BUFFER_SIZE = 1024
   Dim Rslt            As Integer
   Dim InBuffer        As String * IN_BUFFER_SIZE
   Dim ConfigName      As String
   Dim MsgText         As String
   ConfigName = Command$
   If ConfigName = "" Then
      ConfigName = ".\\orders.ini"
   End If
   Rslt = GetPrivateProfileString("Orders Demo", "HostName", "",
   ?InBuffer, Len(InBuffer), ConfigName)
   If Rslt <= 0 Then
      SetSessionDetails = False
      Exit Function
   End If
   UVSession.HostName = Left$(InBuffer, Rslt)
   Rslt = GetPrivateProfileString("Orders Demo", "AccountPath",
"",
   ?InBuffer, Len(InBuffer), ConfigName)
   If Rslt <= 0 Then
      SetSessionDetails = False
      Exit Function
   End If
   UVSession.AccountPath = Left$(InBuffer, Rslt)
   Rslt = GetPrivateProfileString("Orders Demo", "UserName", "",
   ?InBuffer, Len(InBuffer), ConfigName)
   If Rslt < 0 Then
      SetSessionDetails = False
      Exit Function
   End If
   If Rslt = 0 Then
      ' Username is empty, so assume we don't need one
   Else
      ' We have a genuine username...
      UVSession.UserName = Left$(InBuffer, Rslt)
      ' ...so presumably we need a password.
      frmPassword.Show 1
      If ReturnStatus = IDCANCEL Then
         SetSessionDetails = False
         Exit Function
      End If
   End If
   SetSessionDetails = True
End Function
```

Preloading List Boxes

Both the Select Customer dialog box and the Product Catalog present list boxes showing lists of records. Each list box entry consists of a record ID and a description, separated by a tab character. "Tab Stops in List Boxes" on page B-13 describes how to place the tab stops in more detail. Each of the two forms includes code to load the list box with the necessary information. For the Select Customer dialog box, the **Form_Load** procedure in GETCUST.FRM, shown below, includes all the code. For the Product Catalog, the code in PRODUCTS.FRM is divided between the **Form_Load** and **Form_Activate** procedures.

```
'*****************************************************************
*****
' Form_Load: set tab stops in the list box, and set up the list of
' customers. There is no need to worry about sorting the list,
' since lstCustomers.Sorted is set to True at design time.
'*****************************************************************
*****
Sub Form_Load ()
   Dim Rslt    As Integer
   Dim MySelectList  As Object
   Dim NextCustomerNo As String
   TabStops(1) = 100   ' 4 * numbers of character positions
   Rslt = SendMessage(lstCustomers.hWnd, LB_SETTABSTOPS, 1,
TabStops(1))

   '  The list box control is called "lstCustomers". This
statement
   '  removes any existing contents.
   lstCustomers.Clear
   Set MySelectList = UVSession.SelectList(0)

   ' Create the select list. There is no need to sort it; the list
box
   ' will do that.
   MySelectList.Select CustomerFile
   NextCustomerNo = MySelectList.Next
   Do While Not (MySelectList.LastRecordRead)
      ' Retrieve the descriptive field using ReadField.
      CustomerFile.RecordID = NextCustomerNo
```

```
         CustomerFile.ReadField CSTORENAME
         If CustomerFile.Error <> 0 Then
            DisplayError "Error reading customer " & NextCustomerNo,
            ?CustomerFile.Error, MB_ICONEXCLAMATION
               Exit Do
           Else
            ' The Record property now holds the field value read.
            lstCustomers.AddItem CustomerFile.Record & Chr(9) &
            ?NextCustomerNo
           End If
           NextCustomerNo = MySelectList.Next
      Loop
      Set MySelectList = Nothing
   End Sub
```

Note the use of the symbolic constant CSTORENAME to refer to the field position for the Store Name field. The use of symbolic constants is discussed in "Defining Record Layouts" on page B-15.

Tab Stops in List Boxes

Sometimes the program needs a coded value or record ID that does not give sufficient information to a user. The solution is to use tab stops which allow you to display explanatory text beside each option. You can use a Windows API message to set tab stops. When you have done this, you can add items tab characters, that is, CHAR(9) to separate text into columns. For more information about this, see the article called "How to Set Tab Stops in a Visual Basic List Box" in the Microsoft Knowledge Base.

The following procedure sets a single tab stop to separate the customer store name from the customer number. When the user has selected something from the list box, the **Text** property of the list box holds the complete text of the selected item, from which the program can easily extract the necessary code. In GETCUST.FRM, the **cmdOpen_Click** procedure does just this:

```
'*****************************************************************
*****
'  cmdOpen_Click: the user wants to open the selected customer.
'*****************************************************************
*****
Sub cmdOpen_Click ()
   Dim ListBoxText As String, TabPosition As Integer

   ListBoxText = lstCustomers.Text
   TabPosition = InStr(ListBoxText, TAB_CHAR)
   If TabPosition > 0 Then
      NewCustomerNo = Mid$(ListBoxText, TabPosition + 1)
      ReturnStatus = IDOK
    Else
```

```
            NewCustomerNo = ""
            ReturnStatus = IDCANCEL
         End If
         Me.Hide

      End Sub
```

In this case, the customer number is left visible, but you can hide the code or record ID from the user by setting the tab stop to a value greater than the width of the list box. Any text following the tab character is not displayed, although it is still stored by the list box and is returned as part of the **Text** property.

Selecting a Customer

The code in GETCUST.FRM stores the record ID of the selected customer. Control returns to the code in CUSTOMER.FRM, which must read the data for that customer and display it on the screen. Here is the **mnuCustOpen_Click** procedure, which performs the **Open** action from the Customer menu:

```
'*****************************************************************
*******
'  Open an existing customer record
'*****************************************************************
*******
Sub mnuCustOpen_Click ()

   CheckCustomerChanges
   If ReturnStatus = IDCANCEL Then
      Exit Sub
   End If

   frmGetCustomer.Show 1
   If ReturnStatus = IDCANCEL Then
      Exit Sub
   End If

   OpenCustomer NewCustomerNo
End Sub
```

The first action is to check for any unsaved changes to a customer record that is already displayed. Once that is dealt with, it shows the **frmGetCustomer** form as a modal dialog, so that control does not return here until the dialog is completed. After the dialog, it must check whether the user chose **Cancel**, and if so, exit from the operation. Finally, it calls the **OpenCustomer** procedure to display the data.

Defining Record Layouts

The **OpenCustomer** procedure makes extensive use of symbolic constants to identify the fields of the CUSTOMERS record. These constants, along with similar constants for the other files involved, are held in the ORDERDEF.BAS file. Here is an extract from that file:

```
'*****************************************************************
*****
' ORDERDEF.BAS
'
' Copyright (c) 1995 VMARK Software Inc.
'
```

```
' File of constant definitions for the Order Entry sample
application
'
'*****************************************************************
*****
Option Explicit

' PRODUCTS file:
Global Const PRODUCT_FILE_NAME = "PRODUCTS"

' Structure of the PRODUCTS record
Global Const PNAME = 1      ' Name
Global Const PCLASS = 2     ' Class
Global Const PDESC = 3      ' Description
Global Const PPRICE = 4     ' Price
Global Const PDEFAULT = 5   ' Default Order Qty
Global Const PQOH = 6       ' Quantity On Hand
Global Const PWARN = 7      ' Re-order level
Global Const PWEIGHT = 8    ' Weight
' CUSTOMERS file:
Global Const CUSTOMER_FILE_NAME = "CUSTOMERS"

' Structure of the CUSTOMERS record
Global Const CNAME = 1          ' Customer Name
Global Const CADDRESS = 2       ' Billing Address
Global Const CSTORENAME = 3     ' Store Name
Global Const CSTOREADDR = 4     ' Store Address
Global Const CCONTACT = 5       ' Contact
Global Const CCTPHONE = 6       ' Phone
Global Const CPRICE = 7         ' Price Category
Global Const CROUTE = 8         ' Route Code
Global Const CSTOREPHONE = 9    ' Store Phone
Global Const CORDER = 11        ' Orders
' ORDERS file:
Global Const ORDER_FILE_NAME = "ORDERS"

' Structure of the ORDERS record
Global Const ODATE = 1          ' Order Date
Global Const OCUSTOMER = 2      ' Customer Number
Global Const OINSTRUCTIONS = 3  ' Special Instructions
Global Const OPRODUCT = 4       ' Product Code
Global Const OQTY = 5           ' Quantity Ordered
Global Const OUNITPRICE = 6     ' Unit Price
'  Symbolic constants used with the Order Lines Grid
Global Const TWIPS_PER_INCH = 1440

'  Column positions
Global Const GC_PRODUCT = 1        '  Product Code
Global Const GC_PDESC = 2          '  Product Description
Global Const GC_QTY = 3           '  Quantity Ordered
Global Const GC_UNITPRICE = 4     '  Unit Price
Global Const GC_PPRICE = 5        '  Total Price
Global Const GC_MAXCOL = 5        '  Highest column number

' End of constants specific to this application
```

Displaying Record Data

The **OpenCustomer** procedure manages the display of data from the record in the CUSTOMERS file:

```
'*****************************************************************
*******
'  OpenCustomer opens a customer record, using the customer number
'  passed to it.
'*****************************************************************
*******
Sub OpenCustomer (ByVal RecID As String)
   Dim NumContacts     As Integer
   Dim ContactNo       As Integer

   Screen.MousePointer = HOURGLASS
   CustomerFile.RecordID = RecID
   CustomerFile.Read
   Screen.MousePointer = DEFAULT
   Select Case CustomerFile.Error
   Case 0
      '  No error - process text
      Set CustomerRec = CustomerFile.Record

      CurrentCustomer = RecID
      lblCID.Caption = RecID
      txtCName.Text = CustomerRec.Field(CNAME)
      txtCAddress.Text = MVToText(CustomerRec.Field(CADDRESS), VM)
      txtCStoreName.Text = CustomerRec.Field(CSTORENAME)
      txtCStoreAddr.Text =
MVToText(CustomerRec.Field(CSTOREADDR),VM)
      NumContacts = CustomerRec.Field(CCONTACT).Count
      For ContactNo = 1 To NumContacts
         lstCContact.AddItem CustomerRec.value(CCONTACT,ContactNo)
&
         ? TAB_CHAR & CustomerRec.value(CCTPHONE, ContactNo)
         Next ContactNo
      SetListIndex cmbCPrice, PriceCatList,
CustomerRec.Field(CPRICE)
      CurrentDiscount = DiscountFactors(cmbCPrice.ListIndex)
      SetListIndex cmbCRoute, RouteCodeList,
CustomerRec.Field(CROUTE)
   Case UVE_RNF
      '  No record, but the ID is still valid (and locked)
      DisplayError "Customer " & RecID & " not found.", 0,
      ?MB_ICONEXCLAMATION
      ClearDataFields

   Case UVE_LCK
      ' Record not available, so the ID is not locked.
      DisplayError "Customer " & RecID & " is locked by another
user.",
      ?0, MB_ICONEXCLAMATION
      ClearDataFields

   Case Else
      DisplayError "Read error!", CustomerFile.Error,
MB_ICONEXCLAMATION
      ClearDataFields
    End Select
    '  Clear the flag that indicates changes to be saved.
    '  This must be done as the very last thing, because
    '  any changes to control contents will cause it
    '  to be set again.
    CDetailsChanged = False

End Sub
```

Entering a New Order

The process of entering a new order starts in CUSTOMER.FRM, with the New action on the Orders menu. This invokes the **mnuOrderNew_Click** procedure:

```
Sub mnuOrderNew_Click ()

   If CurrentCustomer = "" Then
      Exit Sub
   End If

   frmCustomer.Enabled = False

   NewOrderFlag = True
   frmOrder.Show 0
End Sub
```

Note that the program shows the **frmOrder** form as a modeless dialog, rather than a modal dialog. In a modal dialog, the user would not be able to switch between the Product Catalog and the Customer Order form, as intended. However, you do not want the user to return to the Customer Details form, and so the program disables it before showing the Customer Order form.

The **Form_Load** procedure of ORDER.FRM performs these tasks:

- Sets tab stops in the list box for the Customer Contacts field
- Initializes the Grid control
- Displays the customer details in controls provided on the Customer Order form

For a new order, the program allocates an order number and enables the form's buttons to allow the user to edit and save the order. For an existing order, it reads the order record from the file and disables the buttons. Finally, the procedure calls **LoadOrderData** to display details from the order record, whether it is a new or existing record.

Here is the procedure in full:

```
Sub Form_Load ()
   Dim iRslt           As Integer
   Dim iNumContacts    As Integer
   Dim iContactNo      As Integer

   ' Set a tab stop in the list box for Contacts.
   TabStops(1) = 100   ' 4 * numbers of character positions
   iRslt = SendMessage(lstCContact.hWnd, LB_SETTABSTOPS, 1,
TabStops(1))
   ' Set up the fixed attributes of the Order Line grid
   ' EG_Setup gdOrderLine, txtLineEdit, GC_MAXCOL
   ' Product Code
   EG_SetCol GC_PRODUCT, "OPRODUCT", "K", "Product Code",
   ?"", .75 * TWIPS_PER_INCH, "L"
   ' Product Description
   EG_SetCol GC_PDESC, "OPDESC", "I", "Description",
   ?"", 2.5 * TWIPS_PER_INCH, "L"
   '  Quantity
   EG_SetCol GC_QTY, "OQTY", "D", "Qty","MD0", .5 *
TWIPS_PER_INCH, "R"
   '  Unit Price
   EG_SetCol GC_UNITPRICE, "OUNITPRICE", "D", "Unit Price",
   ?"MD2,$", .75 * TWIPS_PER_INCH, "R"
   '  6th column - total price
   EG_SetCol GC_PPRICE, "OPPRICE", "I", "Total Price",
   ?"MD2,$", .75 * TWIPS_PER_INCH, "R"

   ' Clear the Order Line Grid
   EG_Clear
   ' Load up the Customer details
   lblCID.Caption = CurrentCustomer
   lblCName.Caption = CustomerRec.Field(CNAME)
   lblCStoreName.Caption = CustomerRec.Field(CSTORENAME)
   lblCStoreAddr.Caption = MVToText(CustomerRec.Field(CSTOREADDR),
VM)
   iNumContacts = CustomerRec.Field(CCONTACT).Count
   For iContactNo = 1 To iNumContacts
      lstCContact.AddItem CustomerRec.value(CCONTACT, iContactNo)
      ?& TAB_CHAR & CustomerRec.value(CCTPHONE, iContactNo)
   Next iContactNo
   lblCPrice.Caption = CustomerRec.Field(CPRICE)
   lblCRoute.Caption = CustomerRec.Field(CROUTE)
   If NewOrderFlag Then
      ' It is a new order - initialize it, and prepare the
controls
      ' accordingly.
      CreateOrder
      If OrderRec Is Nothing Then
         ' Could not create an order for some reason
         Unload frmOrder
         Exit Sub
      End If
      cmdAbandon.Enabled = True
      cmdDeleteLine.Enabled = True
      cmdPlace.Enabled = True
      mnuOrderPlace.Enabled = True
      txtOInstructions.Enabled = True
      iRslt = EG_AllowEdits(True)

      ' Prepare to add products
      frmProducts.Show 0
   Else
      ' Not a new order - displaying an existing one
      OpenOrder
      If OrderRec Is Nothing Then
         ' Could not read the order for some reason
         Unload frmOrder
         Exit Sub
      End If
      cmdAbandon.Enabled = False
      cmdDeleteLine.Enabled = False
      cmdPlace.Enabled = False
      mnuOrderPlace.Enabled = False
      txtOInstructions.Enabled = False
      iRslt = EG_AllowEdits(False)
   End If

   LoadOrderData

End Sub
```

Using the Grid Control

The Orders program uses only the custom controls supplied by Microsoft with Visual Basic Version 3.0 so that any Visual Basic user can browse and modify the Orders program. But the Microsoft Grid control GRID.VBX does not allow data to be edited, nor does it recognize BASIC conversion codes. To overcome this, the Orders program includes a code module, EDITGRID.BAS, which holds a set of procedures designed to handle the grid. The names of EDITGRID.BAS procedures all start with **EG_**.

Note: *The EDITGRID.BAS module is too long to include in this appendix, but you can browse or print the online file if required.*

The Customer Orders form includes a Grid control (**gdOrderLine**) and a special text box control (**txtLineEdit**) which is normally invisible but becomes visible when grid data is edited. The text box control's **Borderstyle** property is set to 0. The **Form_Load** procedure shown in "Entering a New Order" on page B-5 calls the **EG_Setup** procedure with the Grid control, the text box, and the number of data columns as arguments. Then it calls **EG_SetCol** once for each data column, passing it details of the column, including the field type, column heading, column width, and conversion code specified in the dictionary. A number of event procedures in the Customers Orders form call the corresponding **EDITGRID.BAS** procedure, as shown in the following table.

Event Procedure	EDITGRID.BAS Procedure	Function
gdOrderLine_Click	EG_Grid_Click	Selects the entire row of the selected cell.
gdOrderLine_DblClick	EG_Grid_DblClick	Starts editing the selected cell.
gdOrderLine_GotFocus	EG_Grid_GotFocus	Selects the entire current row.
gdOrderLine_KeyPress	EG_Grid_KeyPress	Starts editing the current cell by entering any printable character.

Event Procedures for Grid Handling

Event Procedure	EDITGRID.BAS Procedure	Function
gdOrderLine_KeyDown	EG_Grid_KeyDown	Allows movement of selected row with **Up** or **Down Arrow** keys.
txtLineEdit_KeyDown	EG_Text_KeyDown	Allows the arrow keys to change the selected cell.
txtLineEdit_LostFocus	EG_Text_LostFocus	Validates the changed entry and enters it into the grid.

Event Procedures for Grid Handling (Continued)

EDITGRID.BAS also includes procedures to manipulate the grid and its data, which are available to be called when necessary, as shown in the following table.

Procedure	Function
EG_AddItem	Appends a new row. The procedure returns the row number of the new row.
EG_AllowEdits	Enables or disables editing.
EG_Clear	Empties the grid of all data and deletes data rows.
EG_GetColumnData	Returns the data for one column. The returned data is in internal format, with row values separated by value marks.
EG_RemoveItem	Removes a row from the grid. The following rows are moved up.
EG_SetData	Sets a single data value in one cell. Data should be supplied in internal format; the conversion code specified for that column will be applied.
EG_StopEdit	Terminates any edit currently in progress. This procedure returns **False** if the user has chosen to retry the edit, **True** if it is now complete.

Other Grid Handling Procedures

Allocating an Order Number

The order record for the new order is created by the **CreateOrder** procedure, found in the ORDERSUB.BAS module:

```
Sub CreateOrder ()

   ' Obtain a number for the new order.
   CurrentOrder = GetNewOrderNumber()
   If CurrentOrder = "" Then
      Set OrderRec = Nothing
      Exit Sub
   End If

   Set OrderRec = CreateObject("DataBase.DynamicArray.1")
   '  initial values for a new order ....
   OrderRec.Field(ODATE) = DateNow()
   OrderRec.Field(OCUSTOMER) = CurrentCustomer
   OrderRec.Field(OINSTRUCTIONS) = ""

End Sub
```

This procedure tries to get a new order number. If successful, it creates a **Dynami-cArray** object to hold the new record, and fills in the Order Date, Customer Number, and Special Instructions fields. The order date comes from the **DateNow** procedure, which returns today's date in internal format, and which you can find in the UV_UTILS.BAS module.

The new order number is allocated by **GetNewOrderNumber**, also found in ORDERSUB.BAS, which reads and updates a record called &NEXT.AVAILABLE& held in the dictionary of the ORDERS file. Here is the procedure:

```
Function GetNewOrderNumber () As String
   Dim strOrderNo      As String
   Dim lOrderNo        As Long

   ' &NEXT.AVAILABLE& is an X-type dictionary record with the next
   ' available order number in field 2.  The default LockStrategy
   ' and ReleaseStrategy ensure that this record is locked.
   OrderDict.RecordID = "&NEXT.AVAILABLE&"
   OrderDict.Read
```

```
    If OrderDict.Error Then
       DisplayError "Error reading next available order number",
          OrderDict.Error, MB_ICONEXCLAMATION
       GetNewOrderNumber = ""
       Exit Function
    End If
    strOrderNo = OrderDict.Record.Field(2)
    If Not (IsNumeric(strOrderNo)) Then
       DisplayError "Next available order number is not numeric",
       ? 0, MB_ICONEXCLAMATION
       GetNewOrderNumber = ""
       Exit Function
     End If
    OrderDict.Record.Field(2) = Format$(CLng(strOrderNo) + 1)
    OrderDict.Write
    If OrderDict.Error Then
       DisplayError "Error writing next available order number",
       ? OrderDict.Error, MB_ICONEXCLAMATION
       GetNewOrderNumber = ""
       Exit Function
    End If

    GetNewOrderNumber = LTrim$(strOrderNo)

End Function
```

Displaying the Product Catalog

The Product Catalog form and its associated procedures are found in the file PRODUCT.FRM. These procedures are not listed here, apart from the **cmdAdd_Click** procedure, since they mostly use programming techniques that are described elsewhere.

The **cmdAdd_Click** procedure uses some of the grid-handling procedures from EDITGRID.BAS to add a new item to the order. Here is the procedure:

```
Sub cmdAdd_Click ()
   Dim iRow        As Integer

   iRow = EG_AddItem()
   ' Product Code
   EG_SetData iRow, GC_PRODUCT, lblPID.Caption
   ' Description
   EG_SetData iRow, GC_PDESC, ProductRec.Field(PDESC)
   ' Quantity Ordered
   EG_SetData iRow, GC_QTY, txtOrderQty.Text
```

```
      ' Unit Price
      EG_SetData iRow, GC_UNITPRICE, GetDiscountedPrice()
      ComputeOrderTotals
      ODetailsChanged = True

   End Sub
```

After adding the row, it calls the **ComputeOrderTotals** procedure, which is described in the next section.

Computing Order Totals

Every time the order details change, the program calls the **ComputeOrderTotals** procedure found in ORDERSUB.BAS to recompute the line total for each order line and the overall order total. It uses two **DynamicArray** objects as working variables, and it also calls grid-handling procedures from EDITGRID.BAS. Here is the procedure:

```
   Dim OIQty       As Object
   Dim OIPrice     As Object

   Sub ComputeOrderTotals ()
      Dim iNumRows        As Integer
      Dim iRow            As Integer
      Dim lRowTotal       As Long
      Dim lOrderTotal     As Long

      If OIQty Is Nothing Then
         Set OIQty = CreateObject("dataBase.DynamicArray.1")
      End If
      If OIPrice Is Nothing Then
         Set OIPrice = CreateObject("dataBase.DynamicArray.1")
      End If
      OIQty.StringValue = EG_GetColumnData(GC_QTY)
      OIPrice.StringValue = EG_GetColumnData(GC_UNITPRICE)
      iNumRows = OIQty.Field(1).Count
      lOrderTotal = 0

      For iRow = 1 To iNumRows
         lRowTotal = Val(OIQty.value(1, iRow)) * Val(OIPrice.value(1,
   iRow))
         lOrderTotal = lOrderTotal + lRowTotal
         EG_SetData iRow, 5, CStr(lRowTotal)
      Next iRow
         frmOrder.lblOTotal.Caption =
   UVSession.OConv(CStr(lOrderTotal),
         ? "MD2,$")

   End Sub
```

Saving the Order

When you have finished entering an order, and you want to save it in the file, you can either click the **Place Order** button or you can choose **Place** from the Order menu. Either action calls the **mnuOrderPlace_Click** procedure. This checks for completion of any editing of data in the grid, allowing for the action to be cancelled if the user chooses to retry the edit. Then it builds up the order record from the data held in the form controls, and writes it to the ORDERS file. Finally, it unloads the **frmOrders** form. This invokes the **Form_Unload** procedure, which reenables the customer form, so that the user can go back to working with customer data.

Here is the **mnuOrderPlace_Click** procedure:

```
Sub mnuOrderPlace_Click ()

   ' Complete any Grid edit that may have been underway...
   If Not (EG_StopEdit(True)) Then
      ' User is re-trying the edit
      Exit Sub
   End If

   ' Map the data from the screen controls into the record.
   OrderRec.Field(OINSTRUCTIONS) = TextToMV(txtOInstructions.Text,
VM)
   OrderRec.Field(OPRODUCT) = EG_GetColumnData(GC_PRODUCT)
   OrderRec.Field(OQTY) = EG_GetColumnData(GC_QTY)
   OrderRec.Field(OUNITPRICE) = EG_GetColumnData(GC_UNITPRICE)
   ' Write the record to the ORDERS file.
   OrderFile.RecordID = CurrentOrder
   OrderFile.Record = OrderRec.StringValue
   Screen.MousePointer = HOURGLASS
   OrderFile.Write
   Screen.MousePointer = DEFAULT
   If OrderFile.Error = 0 Then
      ODetailsChanged = False
   Else
      DisplayError "Error writing order file!", OrderFile.Error,
      ?MB_ICONEXCLAMATION
    End If

   ' No need to clean up the form, because Form_Unload will do
that
   Unload frmOrder
End Sub
```

Here is the **Form_Unload** procedure:

```
Sub Form_Unload (Cancel As Integer)

   CheckOrderChanges
   If ReturnStatus = IDCANCEL Then
      Cancel = True
      Exit Sub
   End If

   ' Clear current order details
   NewOrderFlag = False
   ODetailsChanged = False
   CurrentOrder = ""
   Set OrderRec = Nothing

   ' Wake up the Customer form
   frmCustomer.Enabled = True

End Sub
```

Viewing Existing Orders

From the Customer Details form, the other action you can take is to look at existing orders for the selected customer. To do this, choose **List All** or **List Outstanding** from the Orders menu. This invokes either the **mnuOrderList_Click** or the **mnuOrderListAll_Click** procedure. Since the demonstration application does not differentiate outstanding orders, the two procedures have the same effect, and in fact are almost identical. The **mnuOrderList_Click** procedure will be discussed.

The **mnuOrderList_Click** procedure first checks that there is a customer displayed, and if not, it exits. Secondly, it calls **SelectOrders** to select the orders for the current customer. The program does this before loading the **frmOrderList** form, so that if there are no orders, it does not confuse the user by displaying an empty form and then removing it again. Here is the **mnuOrderList_Click** procedure:

```
Sub mnuOrderList_Click ()
   Dim iOrderCount     As Long

   If CurrentCustomer = "" Then
      Exit Sub
   End If
   iOrderCount = SelectOrders(False)
   If iOrderCount > 0 Then
      frmOrderList.Show 1
   End If
End Sub
```

Selecting the Order Records

The program creates a select list of orders for a particular customer by executing an SSELECT command. This happens in the **SelectOrders** procedure, found in the ORDERSUB.BAS file. It returns the number of records selected, or –1 if there was an error. The argument is intended to indicate whether to select all orders (if **True**), or only outstanding ones (if **False**). This is not implemented in the demo.

```
Function SelectOrders (ByVal iDisplayAll As Integer) As Long
   Dim MsgText         As String
   Dim UVCommand       As Object
   Dim NumSelected     As Long
   '  Execute the SELECT command
   Set UVCommand = UVSession.Command
   UVCommand.Text = "SSELECT ORDERS BY.DSND ODATE BY.DSND @ID
   ?WITH OCUSTOMER = '" & CurrentCustomer & "'"
   Screen.MousePointer = HOURGLASS
   UVCommand.Exec
   Screen.MousePointer = DEFAULT
   ' Check that we were able to execute a command
   If UVCommand.CommandStatus <> UVS_COMPLETE Then
      ' id not manage to execute it
      MsgText = "Error executing SELECT:" & NL
      MsgText = MsgText & "Command status " &
UVCommand.CommandStatus
      DisplayError MsgText, 0, MB_ICONSTOP
      ' The connection is probably unusable at this stage, so quit
      End
   End If
   ' Check whether the SELECT was successful
   NumSelected = UVCommand.AtSelected
   If NumSelected < 0 Then
      ' Executed the SELECT, but it reported an error
      MsgText = "Error in SELECT:" & NL & UVCommand.Text & NL & NL
      MsgText = MsgText & MVToText(UVCommand.Response, FM)
      DisplayError MsgText, 0, MB_ICONSTOP
      SelectOrders = -1
      Exit Function
   End If
   '  Check whether there are any IDs in the select list.
   If NumSelected = 0 Then
      '  No records were selected.
      MsgText = "Customer " & CurrentCustomer & " has no orders on
file."
      DisplayError MsgText, 0, MB_ICONINFORMATION
      SelectOrders = 0
      Exit Function
   End If
   SelectOrders = NumSelected

End Function
```

Displaying the Order List

The **frmOrderList** form, found in the ORDRLIST.FRM file, uses a list box with tab stops to display the list of orders. The **Form_Load** procedure sets up the display. It sets the tab stops in the list box using the method described earlier, displays some information about the customer, and then calls the **LoadOrderList** procedure to load the list box. Here it is:

```
Sub Form_Load ()
   Dim Rslt            As Integer

   ' Each tab position is calculated as 4 * number of character
   ' positions needed for that field.
   TabStops(1) = 32
   TabStops(2) = 88
   TabStops(3) = 144
   Rslt = SendMessage(lstOrders.hWnd, LB_SETTABSTOPS, 3,
TabStops(1))
   ' Load up the form controls.
   lblCID.Caption = CurrentCustomer
   lblCName.Caption = CustomerRec.Field(CNAME)
   lblCStoreName.Caption = CustomerRec.Field(CSTORENAME)
   lblCStoreAddr.Caption = MVToText(CustomerRec.Field(CSTOREADDR),
VM)
   Rslt = LoadOrderList()
   If Not (Rslt) Then
      lstOrders.Clear
   End If

End Sub
```

The **LoadOrderList** procedure is unlike the other list box procedures in that it uses a select list that has already been created, and it uses **ReadNamedField** to retrieve the data. **ReadNamedField** does two things that **ReadField** does not: it applies the conversion code from the dictionary record, and it can evaluate an I-descriptor. In **LoadOrderList**, the OTOTAL field is an I-descriptor, and both fields have conversion specifiers. Here is the **LoadOrderList** procedure:

```
Function LoadOrderList () As Integer
   Dim UVSelectList    As Object
   Dim NextOrderNo     As String
   Dim NextOrderDate   As String
   Dim NextOrderValue  As String

   lstOrders.Clear
   lstOrders.AddItem "Order" & Chr(9) & "Order Date" & Chr(9) &
   ?"Total Value"
   Set UVSelectList = UVSession.SelectList(0)
   NextOrderNo = UVSelectList.Next
   Do While Not (UVSelectList.LastRecordRead)
      OrderFile.RecordID = NextOrderNo
      OrderFile.ReadNamedField "ODATE"
      NextOrderDate = OrderFile.Record
      OrderFile.ReadNamedField "OTOTAL"
      NextOrderValue = OrderFile.Record
      If OrderFile.Error <> 0 Then
         DisplayError "Error reading order " & NextOrderNo,
         ?OrderFile.Error, MB_ICONEXCLAMATION
         LoadOrderList = False
         Exit Function
         End If
         lstOrders.AddItem NextOrderNo & Chr(9) & NextOrderDate &
         ?Chr(9) & NextOrderValue
         NextOrderNo = UVSelectList.Next
   Loop
   Set UVSelectList = Nothing
   LoadOrderList = True
End Function
```

Displaying the Order Details

To view details of one of the orders in the order list, you can either click the **View** button or double-click in the list box. Either action calls the **cmdView_Click** procedure. The first step is to check that you have selected an order, and not the line of column headings that is also shown in the list box. If you have selected an order, the procedure extracts the order number from the text of the selected item, and shows the **frmOrder** form. This time, it can be shown as a modal dialog, since there is no need to access the Product Catalog at this point.

Here is the **cmdView_Click** procedure from ORDRLIST.FRM:

```
Sub cmdView_Click ()
   Dim ListBoxText As String, TabPosition As Integer

   If lstOrders.ListIndex < 1 Then
      ' The first item is a heading
      MsgBox "Please select a valid order.", MB_OK +
MB_ICONEXCLAMATION
      Exit Sub
   End If

   ListBoxText = lstOrders.Text
   TabPosition = InStr(ListBoxText, TAB_CHAR)
   If TabPosition > 0 Then
      CurrentOrder = Left$(ListBoxText, TabPosition - 1)
   Else
      CurrentOrder = ""
   End If
   frmOrder.Show 1

End Sub
```

Showing the **frmOrder** form invokes its **Form_Load** procedure, which is listed in Entering a New Order. This time, the **NewOrderFlag** variable is **False**, so that the **OpenOrder** and **LoadOrderData** procedures are called. **OpenOrder**, found in ORDERSUB.BAS, reads an existing order record into the **OrderRec** variable:

```
Sub OpenOrder ()
   Screen.MousePointer = HOURGLASS
   OrderFile.RecordID = CurrentOrder
   OrderFile.Read
   Screen.MousePointer = DEFAULT
   Select Case OrderFile.Error
   Case 0
      ' No error - process text
      Set OrderRec = OrderFile.Record
   Case UVE_RNF
      ' No record, but the ID is still valid (and locked)
      DisplayError "Order " & CurrentOrder & " not found.", 0,
      ?MB_ICONEXCLAMATION
      Set OrderRec = Nothing

   Case UVE_LCK
      ' Record not available, so the ID is not locked.
      DisplayError "Order " & CurrentOrder &
      ?" is locked by another user.", 0, MB_ICONEXCLAMATION
      Set OrderRec = Nothing

   Case Else
      DisplayError "Read error!", OrderFile.Error,
MB_ICONEXCLAMATION
      Set OrderRec = Nothing
   End Select
End Sub
```

LoadOrderData, found in ORDER.FRM, transfers the order details from **OrderRec** to the screen controls:

```
Sub LoadOrderData ()
   Dim iRow            As Integer
   Dim iNumItems       As Integer
   Dim iItemNo         As Integer
   Dim sProductCode    As String

   ' The simple stuff is loaded first.
   lblOID.Caption = CurrentOrder
   lblODate.Caption = UVSession.OConv(OrderRec.Field(ODATE), "D2")
   txtOInstructions.Text = MVToText(OrderRec.Field(OINSTRUCTIONS),
VM)

   ' Now for the line items grid...
   iNumItems = OrderRec.Field(OPRODUCT).Count
   For iItemNo = 1 To iNumItems

      iRow = EG_AddItem()
      ' Product Code
      sProductCode = OrderRec.value(OPRODUCT, iItemNo)
      EG_SetData iRow, GC_PRODUCT, sProductCode
      ' Description
      ProductFile.RecordID = sProductCode
      ProductFile.ReadField PDESC
      If ProductFile.Error Then
         ' Don't really care
         EG_SetData iRow, GC_PDESC, ""
      Else
         EG_SetData iRow, GC_PDESC, ProductFile.Record
      End If
      ' Quantity Ordered
      EG_SetData iRow, GC_QTY, OrderRec.value(OQTY, iItemNo)
      ' Unit Price
      EG_SetData iRow, GC_UNITPRICE, OrderRec.value(OUNITPRICE,
iItemNo)

   Next iItemNo

   ComputeOrderTotals
   ODetailsChanged = False

End Sub
```

Data Conversion Functions

The demo program described in Appendix B, “ The Demo Application,” contains some data conversion functions that you can use in your Visual Basic applications. They can be found in the file SAMPLES\ORDERS\CLIENT\UV_UTILS.BAS in your UniDK installation directory. The functions are described below.

TextToMV

This function converts each newline sequence (Return followed by Line Feed) in a string into a database system delimiter character. Use **TextToMV** to convert text from a multiline text control into a multivalued field to be written to a database file. For example:

```
OrderRec.Field(OINSTRUCTIONS) =
TextToMV(txtOInstructions.Text, Chr(I_VM))
```

MVToText

This function converts the specified delimiters in a string into the text newline sequence (Return followed by Line Feed). Use **MVToText** to display a group of values from a multivalued field as lines in a multiline text control. For example:

```
txtOInstructions.Text =
MVToText(OrderRec.Field(OINSTRUCTIONS), Chr(I_VM))
```

GetFieldType

Use this function to retrieve a field's type from field 1 of a dictionary record. For example:

```
f_type = GetFieldType(FileDictPart.Type)
```

DateNow

Use this function to return today's date as a long integer in internal format as returned by the BASIC DATE function. After using the **DateNow** function, convert the returned value to a readable string using the **Oconv** method. For example:

```
OrderRec.Field(ODATE) = DateNow()
```