OpenOffice.org 3.1 BASIC Guide

Copyright

This listing is Chapters 1 to 3 only. If you want more detail, see the unedited full PDF of the 3.2 version of this document.

For ENGG1811 students, the most important sections are on pages 15 to 27 and 33 to 47.

Contents

Preface

This guide provides an introduction to programming with OpenOffice.org Basic. To get the most out of this book, you should be familiar with other programming languages. Extensive examples are provided to help you quickly develop your own OpenOffice.org Basic programs.

Note – Throughout this document, the OpenOffice.org installation directory is represented in syntax as *`install-dir`*.

◆ ◆ ◆ CHAPTER 1

OpenOffice.org BASIC Programming Guide

This guide divides information about OpenOffice.org administration into several chapters. The first three chapters introduce you to OpenOffice.org Basic:

- The Language of OpenOffice.org Basic
- Runtime Library
- Introduction to the API

These chapters provide an overview of OpenOffice.org Basic and should be read by anyone who intends to write OpenOffice.org Basic programs. The remaining chapters describe the individual components of the OpenOffice.org API in more detail and can be read selectively as required:

- Working with Documents
- Text Documents
- Spreadsheet Documents
- Drawings and Presentations
- Charts (Diagrams)
- Databases
- Dialogs
- Forms

About OpenOffice.org Basic

The OpenOffice.org Basic programming language has been developed especially for OpenOffice.org and is firmly integrated in the Office package.

As the name suggests, OpenOffice.org Basic is a programming language from the Basic family. Anyone who has previously worked with other Basic languages — in particular with Visual Basic or Visual Basic for Applications (VBA) from Microsoft — will quickly become accustomed to OpenOffice.org Basic. Large sections of the basic constructs of OpenOffice.org Basic are compatible with Visual Basic.

The OpenOffice.org Basic programming language can be divided into four components:

- The language of OpenOffice.org Basic: Defines the elementary linguistic constructs, for example, for variable declarations, loops, and functions.
- The runtime library: Provides standard functions which have no direct reference to OpenOffice.org, for example, functions for editing numbers, strings, date values, and files.
- The OpenOffice.org API (Application Programming Interface): Permits access to OpenOffice.org documents and allows these to be created, saved, modified, and printed.
- The Dialog Editor: Creates personal dialog windows and provides scope for the adding of control elements and event handlers.

Note – Compatibility between OpenOffice.org Basic and VBA relates to the OpenOffice.org Basic language as well as the runtime library. The OpenOffice.org API and the Dialog Editor are not compatible with VBA (standardizing these interfaces would have made many of the concepts provided in OpenOffice.org impossible).

Intended Users of OpenOffice.org Basic

The scope of application for OpenOffice.org Basic begins where the standard functions of OpenOffice.org end. Routine tasks can therefore be automated in OpenOffice.org Basic, links can be made to other programs — for example to a database server — and complex activities can be performed at the press of a button by using predefined scripts.

OpenOffice.org Basic offers complete access to all OpenOffice.org functions, supports all

functions, modifies document types, and provides options for creating personal dialog windows.

Use of OpenOffice.org Basic

OpenOffice.org Basic can be used by any OpenOffice.org user without any additional programs or aids. Even in the standard installation, OpenOffice.org Basic has all the components needed to create its own Basic macros, including:

- The integrated development environment (IDE) which provides an editor for creating and testing macros.
- The interpreter, which is needed to run OpenOffice.org Basic macros.
- The interfaces to various OpenOffice.org applications, which allow for direct access to Office documents.

More Information

The components of the OpenOffice.org API that are discussed in this guide were selected based on their practical benefits for the OpenOffice.org Basic programmer. In general, only parts of the interfaces are discussed. For a more detailed picture, see the API reference.

The Developer's Guide describes the OpenOffice.org API in more detail than this guide, but is primarily intended for Java and C++ programmers. Anyone who is already familiar with OpenOffice.org Basic programming can find additional information in the Developer's Guide on OpenOffice.org Basic and OpenOffice.org programming.

Programmers who want to work directly with Java or C++ rather than OpenOffice.org Basic should consult the OpenOffice.org Developer's Guide instead of this guide. OpenOffice.org programming with Java or C++ is a considerably more complex process than programming with OpenOffice.org Basic.

◆ ◆ ◆ CHAPTER 2

The Language of OpenOffice.org BASIC

OpenOffice.org Basic belongs to the family of Basic languages. Many parts of OpenOffice.org Basic are identical to Microsoft Visual Basic for Applications and Microsoft Visual Basic. Anyone who has already worked with these languages can quickly become accustomed to OpenOffice.org Basic.

Programmers of other languages —such as Java, C++, or Delphi —should also find it easy to familiarize themselves with OpenOffice.org Basic. OpenOffice.org Basic is a fully-developed procedural programming language and no longer uses rudimentary control structures, such as `GoTo` and `GoSub`.

You can also benefit from the advantages of object-oriented programming since an interface in OpenOffice.org Basic enables you to use external object libraries. The entire OpenOffice.org API is based on these interfaces, which are described in more detail in the following chapters of this document.

This chapter provides an overview of the key elements and constructs of the OpenOffice.org Basic language, as well as the framework in which applications and libraries are oriented to OpenOffice.org Basic.

Overview of a Basic Program

OpenOffice.org Basic is an interpreter language. Unlike C++ or Delphi, the OpenOffice.org Basic compiler does not create executable or self-extracting files that are capable of running automatically. Instead, you execute an OpenOffice.org Basic program inside OpenOffice.org. The code is first checked for obvious errors and then executed line by line.

Program Lines

The Basic interpreter's line-oriented execution produces one of the key differences between Basic and other programming languages. Whereas the position of hard line breaks in the source code of Java, C++, or Delphi programs is irrelevant, each line in a Basic program forms a self-contained unit. Function calls, mathematical expressions, and other linguistic elements, such as function and loop headers, must be completed on the same line that they begin on.

If there is not enough space, or if this results in long lines, then several lines can be linked together by adding underscores _. The following example shows how four lines of a mathematical expression can be linked:

```
LongExpression = (Expression1 * Expression2) + _
(Expression3 * Expression4) + _
(Expression5 * Expression6) + _
(Expression7 * Expression8)
```

Note – The underscore must always be the last character in a linked line and cannot be followed by a space or a tab, otherwise the code generates an error.

In addition to linking individual lines, OpenOffice.org Basic, you can use colons to divide one line into several sections so that there is enough space for several expressions. The assignments

```
a = 1
a = a + 1
a = a + 1
```

can be written as follows:

```
a = 1  :  a = a + 1  :  a = a + 1
```

Comments

In addition to the program code to be executed, an OpenOffice.org Basic program can also contain comments that explain the individual parts of the program and provide important information that can be helpful at a later point.

OpenOffice.org Basic provides two methods for inserting comments in the program code:

- All characters that follow an apostrophe are treated as comments:

```
Dim A    ' This is a comment for variable A
```

- The keyword `Rem`, followed by the comment:

```
Rem This comment is introduced by the keyword Rem.
```

A comment usually includes all characters up to the end of the line. OpenOffice.org Basic then interprets the following line as a regular instruction again. If comments cover several lines, each line must be identified as a comment:

```
Dim B     ' This comment for variable B is relatively long
          ' and stretches over several lines. The
          ' comment character must therefore be repeated
          ' in each line.
```

Markers (Identifiers)

A OpenOffice.org Basic program can contain dozens, hundreds, or even thousands of **markers**, which are names for variables, constants, functions, and so on. When you select a name for a marker, the following rules apply:

- Markers can only contain Latin letters, numbers, and underscores (_).
- The first character of a marker must be a letter or an underscore.
- Markers cannot contain special characters, such as ä â î ß.
- The maximum length of a marker is 255 characters.
- No distinction is made between uppercase and lowercase characters. The `OneTestVariable` marker, for example, defines the same variable as `onetestVariable` and `ONETESTVARIABLE`.

  There is, however, one exception to this rule: a distinction is made between uppercase and lowercase characters for UNO-API constants. More information about UNO is presented in Introduction to the OpenOffice.org API.

Note – The rules for constructing markers are different in OpenOffice.org Basic than in VBA. For example, OpenOffice.org Basic only allows special characters in markers when using Option Compatible, since they can cause problems in international projects.

Here are a few examples of correct and incorrect markers:

```
Surname      ' Correct
Surname5     ' Correct (number 5 is not the first digit)
First Name   ' Incorrect (spaces are not permitted)
DéjàVu       ' Incorrect (letters such as é, à are not permitted)
5Surnames    ' Incorrect (the first character must not be a number)
First,Name   ' Incorrect (commas and full stops are not permitted)
```

Enclosing a variable name in square brackets allows names that might otherwise be disallowed; for example, spaces.

```
Dim [First Name] As String  'Space accepted in square brackets
Dim [DéjàVu] As Integer     'Special characters in square brackets
[First Name] = "Andrew"
[DéjàVu] = 2
```

Working With Variables

Implicit Variable Declaration

Basic languages are designed to be easy to use. As a result, OpenOffice.org Basic enables the creation of a variable through simple usage and without an explicit declaration. In other words, a variable exists from the moment that you include it in your code. Depending on the variables that are already present, the following example declares up to three new variables:

```
a = b + c
```

Declaring variables implicitly is not good programming practice because it can result in the inadvertent introduction of a new variable through, for example, a typing error. Instead of producing an error message, the interpreter initializes the typing error as a new variable with a value of 0. It can be very difficult to locate errors of this kind in your code.

Explicit Variable Declaration

To prevent errors caused by an implicit declaration of variables, OpenOffice.org Basic provides a switch called:

```
Option Explicit
```

This must be listed in the first program line of each module and ensures that an error message is issued if one of the variables used is not declared. The `Option Explicit` switch should be included in all Basic modules.

In its simplest form, the command for an explicit declaration of a variable is as follows:

```
Dim MyVar
```

This example declares a variable with the name `MyVar` and the type `variant`. A variant is a universal variable that can record all conceivable values, including strings, whole numbers, floating point figures, and Boolean values. Here are a few examples of Variant variables:

```
MyVar = "Hello World"      ' Assignment of a string
MyVar = 1                  ' Assignment of a whole number
MyVar = 1.0                ' Assignment of a floating point number
MyVar = True               ' Assignment of a Boolean value
```

The variables declared in the previous example can even be used for different variable types in the same program. Although this provides considerable flexibility, it is best to restrict a variable to one variable type. When OpenOffice.org Basic encounters an incorrectly defined variable type in a particular context, an error message is generated.

Use the following style when you make a type-bound variable declaration:

```
Dim MyVar As Integer   ' Declaration of a variable of the integer type
```

The variable is declared as an integer type and can record whole number values. You can also use the following style to declare an integer type variable:

```
Dim MyVar%          ' Declaration of a variable of the integer type
```

The Dim instruction can record several variable declarations:

```
Dim MyVar1, MyVar2
```

If you want to assign the variables to a permanent type, you must make separate assignments for each variable:

```
Dim MyVar1 As Integer, MyVar2 As Integer
```

If you do not declare the type for a variable, OpenOffice.org Basic assigns the variable a variant

type. For example, in the following variable declaration, `MyVar1` becomes a variant and `MyVar2` becomes an integer:

```
Dim MyVar1, MyVar2 As Integer
```

The following sections list the variable types that are available in OpenOffice.org Basic and describe how they can be used and declared.

Strings

Strings, together with numbers, form the most important basic types of OpenOffice.org Basic. A string consists of a sequence of consecutive individual characters. The computer saves the strings internally as a sequence of numbers where each number represents one specific character.

From a Set of ASCII Characters to Unicode

Character sets match characters in a string with a corresponding code (numbers and characters) in a table that describes how the computer is to display the string.

The ASCII Character Set

The ASCII character set is a set of codes that represent numbers, characters, and special symbols by one byte. The 0 to 127 ASCII codes correspond to the alphabet and to common symbols (such as periods, parentheses, and commas), as well as some special screen and printer control codes. The ASCII character set is commonly used as a standard format for transferring text data between computers.

However, this character set does not include a range of special characters used in Europe, such as â, ä, and î, as well as other character formats, such as the Cyrillic alphabet.

The ANSI Character Set

Microsoft based its Windows product on the American National Standards Institute (ANSI) character set, which was gradually extended to include characters that are missing from the ASCII character set.

Code Pages

The ISO 8859 character sets provide an international standard. The first 128 characters of the ISO character set correspond to the ASCII character set. The ISO standard introduces new character sets (**code pages**) so that more languages can be correctly displayed. However, as a result, the same character value can represent different characters in different languages.

Unicode

Unicode increases the length of a character to four bytes and combines different character sets to create a standard to depict as many of the world's languages as possible. Version 2.0 of Unicode is now supported by many programs — including OpenOffice.org and OpenOffice.org Basic.

String Variables

OpenOffice.org Basic saves strings as string variables in Unicode. A string variable can store up to 65535 characters. Internally, OpenOffice.org Basic saves the associated Unicode value for every character. The working memory needed for a string variable depends on the length of the string.

Example declaration of a string variable:

```
Dim Variable As String
```

You can also write this declaration as:

```
Dim Variable$
```

Note – When porting VBA applications, ensure that the maximum allowed string length in OpenOffice.org Basic is observed (65535 characters).

Specification of Explicit Strings

To assign an explicit string to a string variable, enclose the string in quotation marks (").

```
Dim MyString As String
MyString = " This is a test"
```

To split a string across two lines, add an ampersand sign at the end of the first line:

```
Dim MyString As String
MyString =   "This string is so long that it " & _
             "has been split over two lines."
```

To include a quotation mark (") in a string, enter it twice at the relevant point:

```
Dim MyString As String
MyString = "a ""-quotation mark."    ' produces a "-quotation mark
```

Numbers

OpenOffice.org Basic supports five basic types for processing numbers:

- Integer
- Long Integer
- Single
- Double
- Currency

Integer Variables

Integer variables can store any whole number between **-32768** and **32767**. An integer variable can take up to two bytes of memory. The type declaration symbol for an integer variable is `%`.

Calculations that use integer variables are very fast and are particularly useful for loop counters. If you assign a floating point number to an integer variable, the number is rounded up or down to the next whole number.

Example declarations for integer variables:

```
Dim Variable As Integer
Dim Variable%
```

Long Integer Variables

Long integer variables can store any whole number between **–2147483648** and **2147483647**. A long integer variable can takes up to four bytes of memory. The type declaration symbol for a long integer is `&`. Calculations with long integer variables are very fast and are particularly useful for loop counters. If you assign a floating point number to a long integer variable, the number is rounded up or down to the next whole number.

Example declarations for long integer variables:

```
Dim Variable as Long
Dim Variable&
```

Single Variables

Single variables can store any positive or negative floating point number between **3.402823 x 10^{38}** and **1.401298 x 10^{-45}**. A single variable can take up to four bytes of memory. The type declaration symbol for a single variable is `!`.

Originally, single variables were used to reduce the computing time required for the more precise double variables. However, these speed considerations no longer apply, reducing the need for single variables.

Example declarations for single variables:

```
Dim Variable as Single
Dim Variable!
```

Double Variables

Double variables can store any positive or negative floating point numbers between **1.79769313486232 x 10**^{308} and **4.94065645841247 x 10**$^{-324}$. A double variable can take up to eight bytes of memory. Double variables are suitable for precise calculations. The type declaration symbol is `#`.

Example declarations of double variables:

```
Dim Variable As Double
Dim Variable#
```

Currency Variables

Currency variables differ from the other variable types by the way they handle values. The decimal point is fixed and is followed by four decimal places. The variable can contain up to 15 numbers before the decimal point. A currency variable can store any value between **-922337203685477.5808** and **+922337203685477.5807** and takes up to eight bytes of memory. The type declaration symbol for a currency variable is `@`.

Currency variables are mostly intended for business calculations that yield unforeseeable rounding errors due to the use of floating point numbers.

Example declarations of currency variables:

```
Dim Variable As Currency
Dim Variable@
```

Floats

The types single, double and currency are often collectively referred to as floats, or floating-point number types. They can contain numerical values with decimal fractions of various length, hence the name: The decimal point seems to be able to 'float' through the number.

You can declare variables of the type float. The actual variable type (single, long, currency) is determined the moment a value is assigned to the variable:

```
Dim A As Float
A = 1210.126
```

Specification of Explicit Numbers

Numbers can be presented in several ways, for example, in decimal format or in scientific notation, or even with a different base than the decimal system. The following rules apply to numerical characters in OpenOffice.org Basic:

Whole Numbers

The simplest method is to work with whole numbers. They are listed in the source text without a comma separating the thousand figure:

```
Dim A As Integer
Dim B As Float

A = 1210
B = 2438
```

The numbers can be preceded by both a plus (+) or minus (-) sign (with or without a space in between):

```
Dim A As Integer
Dim B As Float

A = + 121
B = - 243
```

Decimal Numbers

When you type a decimal number, use a period (.) as the decimal point. This rule ensures that source texts can be transferred from one country to another without conversion.

```
Dim A As Integer
Dim B As Integer
Dim C As Float

A = 1223.53      ' is rounded
B = - 23446.46   ' is rounded
C = + 3532.76323
```

You can also use plus (+) or minus (-) signs as prefixes for decimal numbers (again with or without spaces).

If a decimal number is assigned to an integer variable, OpenOffice.org Basic rounds the figure up or down.

Exponential Writing Style

OpenOffice.org Basic allows numbers to be specified in the exponential writing style, for example, you can write `1.5e-10` for the number 1.5 x 10^{-10} (0.00000000015). The letter "e" can be lowercase or uppercase with or without a plus sign (+) as a prefix.

Here are a few correct and incorrect examples of numbers in exponential format:

```
Dim A As Double

A = 1.43E2    ' Correct
A = + 1.43E2  ' Correct (space between plus and basic number)
A = - 1.43E2  ' Correct (space between minus and basic number)
A = 1.43E-2   ' Correct (negative exponent)
A = 1.43E -2  ' Incorrect (spaces not permitted within the number)
A = 1,43E-2   ' Incorrect (commas not permitted as decimal points)
A = 1.43E2.2  ' Incorrect (exponent must be a whole number)
```

Note, that in the first and third incorrect examples that no error message is generated even though the variables return incorrect values. The expression

```
A = 1.43E -2
```

is interpreted as 1.43 minus 2, which corresponds to the value -0.57. However, the value 1.43 x 10^{-2} (corresponding to 0.0143) was the intended value. With the value

```
A = 1.43E2.2
```

OpenOffice.org Basic ignores the part of the exponent after the decimal point and interprets the expression as

```
A = 1.43E2
```

Hexadecimal Values

In the hexadecimal system (base 16 system), a 2-digit number corresponds to precisely one byte. This allows numbers to be handled in a manner which more closely reflects machine architecture. In the hexadecimal system, the numbers 0 to 9 and the letters A to F are used as

numbers. An A stands for the decimal number 10, while the letter F represents the decimal number 15. OpenOffice.org Basic lets you use whole numbered hexadecimal values, so long as they are preceded by `&H`.

```
Dim A As Long
A = &HFF ' Hexadecimal value FF, corresponds to the decimal value 255
A = &H10 ' Hexadecimal value 10, corresponds to the decimal value 16
```

Octal Values

OpenOffice.org Basic also understands the octal system (base 8 system), which uses the numbers 0 to 7. You must use whole numbers that are preceded by `&O`.

```
Dim A As Long
A = &O77 ' Octal value 77, corresponds to the decimal value 63
A = &O10 ' Octal value 10, corresponds to the decimal value 8
```

Boolean Values

Boolean variables can only contain one of two values: `True` or `False`. They are suitable for binary specifications that can only adopt one of two statuses. A Boolean value is saved internally as a two-byte integer value, where 0 corresponds to the False and any other value to True. There is no type declaration symbol for Boolean variables. The declaration can only be made using the supplement `As Boolean`.

Example declaration of a Boolean variable:

```
Dim Variable As Boolean
```

Date and Time

Date variables can contain date and time values. When saving date values, OpenOffice.org Basic uses an internal format that permits comparisons and mathematical operations on date and time values. There is no type declaration symbol for date variables. The declaration can only be made using the supplement `As Date`.

Example declaration of a date variable:

```
Dim Variable As Date
```

Arrays [not used in ENGG1811]

In addition to simple variables (scalars), OpenOffice.org Basic also supports arrays (data fields). A data field contains several variables, which are addressed through an index.

Defining Arrays

Arrays can be defined as follows:

Simple Arrays

An array declaration is similar to that of a simple variable declaration. However, unlike the variable declaration, the array name is followed by parentheses which contain the specifications for the number of elements. The expression

```
Dim MyArray(3)
```

declares an array that has four variables of the variant data type, namely `MyArray(0)`, `MyArray(1)`, `MyArray(2)`, and `MyArray(3)`.

You can also declare type-specific variables in an array. For example, the following line declares an array with four integer variables:

```
Dim MyInteger(3) As Integer
```

In the previous examples, the index for the array always begins with the standard start value of zero. As an alternative, a validity range with start and end values can be specified for the data field declaration. The following example declares a data field that has six integer values and which can be addressed using the indexes 5 to 10:

```
Dim MyInteger(5 To 10) As Integer
```

The indexes do not need to be positive values. The following example also shows a correct declaration, but with negative data field limits:

```
Dim MyInteger(-10 To -5) As Integer
```

It declares an integer data field with 6 values that can be addressed using the indexes -10 to -5.

There are three limits that you must observe when you define data field indexes:

- The smallest possible index is -32768.
- The largest possible index is 32767.
- The maximum number of elements (within a data field dimension) is 16368.

Note – Other limit values sometimes apply for data field indexes in VBA. The same also applies to the maximum number of elements possible per dimension. The values valid there can be found in the relevant VBA documentation.

Specified Value for Start Index

The start index of a data field usually begins with the value 0. Alternatively, you can change the start index for all data field declarations to the value 1 by using the call:

```
Option Base 1
```

The call must be included in the header of a module if you want it to apply to all array declarations in the module. However, this call does not affect the UNO sequences that are defined through the OpenOffice.org API whose index always begins with 0. To improve clarity, you should avoid using Option Base 1.

The number of elements in an array is not affected if you use `Option Base 1`, only the start index changes. The declaration

```
Option Base 1
' ...
Dim MyInteger(3)
```

creates 4 integer variables which can be described with the expressions `MyInteger(1)`, `MyInteger(2)`, `MyInteger(3)`, and `MyInteger(4)`.

Note – In OpenOffice.org Basic, the expression `Option Base 1` does not affect the number of elements in an array as it does in VBA. It is, rather, the start index which moves in OpenOffice.org Basic. While the declaration `MyInteger(3)` creates three integer values in VBA with the indexes 1 to 3, the same declaration in OpenOffice.org Basic creates four integer values with the indexes 1 to 4. By using `Option Compatible`, OpenOffice.org Basic behaves like VBA.

Multi-Dimensional Data Fields

In addition to single dimensional data fields, OpenOffice.org Basic also supports work with multi-dimensional data fields. The corresponding dimensions are separated from one another by commas. The example

```
Dim MyIntArray(5, 5) As Integer
```

defines an integer array with two dimensions, each with 6 indexes (can be addressed through the indexes 0 to 5). The entire array can record a total of 6 x 6 = 36 integer values.

You can define hundreds of dimensions in OpenOffice.org Basic Arrays; however, the amount of available memory limits the number of dimensions you can have.

Dynamic Changes in the Dimensions of Data Fields

The previous examples are based on data fields of a specified dimension. You can also define arrays in which the dimension of the data fields dynamically changes. For example, you can define an array to contain all of the words in a text that begin with the letter A. As the number of these words is initially unknown, you need to be able to subsequently change the field limits. To do this in OpenOffice.org Basic, use the following call:

```
ReDim MyArray(10)
```

Note – Unlike VBA, where you can only dimension dynamic arrays by using `Dim MyArray()`, OpenOffice.org Basic lets you change both static and dynamic arrays using `ReDim`.

The following example changes the dimension of the initial array so that it can record 11 or 21 values:

```
Dim MyArray(4) As Integer ' Declaration with five elements
' ...
ReDim MyArray(10) As Integer ' Increase to 11 elements
' ...
ReDim MyArray(20) As Integer ' Increase to 21 elements
```

When you reset the dimensions of an array, you can use any of the options outlined in the previous sections. This includes declaring multi-dimensional data fields and specifying explicit start and end values. When the dimensions of the data field are changed, all contents are lost. If you want to keep the original values, use the `Preserve` command:

```
Dim MyArray(10) As Integer ' Defining the initial
' dimensions
```

```
' ...
ReDim Preserve MyArray(20) As Integer ' Increase in
' data field, while
' retaining content
```

When you use `Preserve`, ensure that the number of dimensions and the type of variables remain the same.

Note – Unlike VBA, where only the upper limit of the last dimension of a data field can be changed through `Preserve`, OpenOffice.org Basic lets you change other dimensions as well.

If you use `ReDim` with `Preserve`, you must use the same data type as specified in the original data field declaration.

Defining values for arrays

Values for the Array fields can be stored like this:

```
MyArray(0) = "somevalue"
```

Accessing Arrays

Accessing values in an array works like this:

```
MsgBox("Value:" & MyArray(0))
```

Array Creation, value assignment and access example

And example containing all steps that show real array usage:

```
Sub TestArrayAxess
        Dim MyArray(3)
        MyArray(0) = "lala"
        MsgBox("Value:" & MyArray(0))
End Sub
```

Scope and Life Span of Variables

A variable in OpenOffice.org Basic has a limited life span and a limited scope from which it can be read and used in other program fragments. The amount of time that a variable is retained, as well as where it can be accessed from, depends on its specified location and type.

Local Variables

Variables that are declared in a function or a subprogram are called local variables:

```
Sub Test
  Dim MyInteger As Integer
  ' ...
End Sub
```

Local variables only remain valid as long as the function or the subprogram is executing, and then are reset to zero. Each time the function is called, the values generated previously are not available.

To keep the previous values, you must define the variable as `Static`:

```
Sub Test
  Static MyInteger As Integer
  ' ...
End Sub
```

Note – Unlike VBA, OpenOffice.org Basic ensures that the name of a local variable is not used simultaneously as a global and a private variable in the module header. When you port a VBA application to OpenOffice.org Basic, you must change any duplicate variable names.

Public Domain Variables

Public domain variables are defined in the header section of a module by the keyword `Dim`. These variables are available to all of the modules in their library:

Module A:

```
Dim A As Integer
Sub Test
  Flip
  Flop
End Sub

Sub Flip
 A = A + 1
End Sub
```

Module B:

```
Sub Flop
  A = A - 1
End Sub
```

The value of variable `A` is not changed by the `Test` function, but is increased by one in the `Flip` function and decreased by one in the `Flop` function. Both of these changes to the variable are global.

You can also use the keyword `Public` instead of `Dim` to declare a public domain variable:

```
Public A As Integer
```

A public domain variable is only available so long as the associated macro is executing and then the variable is reset.

Global Variables

In terms of their function, global variables are similar to public domain variables, except that their values are retained even after the associated macro has executed. Global variables are declared in the header section of a module using the keyword `Global`:

```
Global A As Integer
```

Private Variables

`Private` variables are only available in the module in which they are defined. Use the keyword `Private` to define the variable:

```
Private MyInteger As Integer
```

If several modules contain a `Private` variable with the same name, OpenOffice.org Basic creates a different variable for each occurrence of the name. In the following example, both module `A` and `B` have a `Private` variable called `C`. The `Test` function first sets the `Private` variable in module `A` and then the `Private` variable in module `B`.

Module A:

```
Private C As Integer

Sub Test
  SetModuleA      ' Sets the variable C from module A
  SetModuleB      ' Sets the variable C from module B
  ShowVarA        ' Shows the variable C from module A (= 10)
  ShowVarB        ' Shows the variable C from module B (= 20)
End Sub

Sub SetmoduleeA
  C = 10
End Sub

Sub ShowVarA
  MsgBox C        ' Shows the variable C from module A.
End Sub
```

Module B:

```
Private C As Integer

Sub SetModuleB
  C = 20
End Sub

Sub ShowVarB
  MsgBox C        ' Shows the variable C from module B.
End Sub
```

Constants

Constants are values which may be used but not changed by the program.

Defining Constants

In OpenOffice.org Basic, use the keyword `Const` to declare a constant.

```
Const A = 10
```

You can also specify the constant type in the declaration:

```
Const B As Double = 10
```

Scope of Constants

Constants have the same scope as variables (see Scope and Life Span of Variables), but the syntax is slightly different. A `Const` definition in the module header is available to the code in that module. To make the definition available to other modules, add the `Public` keyword.

```
Public Const one As Integer = 1
```

Predefined Constants

OpenOffice.org Basic predefines several constants. Among the most useful are:

- `True` and `False`, for Boolean assignment statements
- `PI` as a type `Double` numeric value

```
Dim bHit as Boolean
bHit = True

Dim dArea as Double, dRadius as Double
' ... (assign a value to dRadius)
dArea = PI * dRadius * dRadius
```

Operators

OpenOffice.org Basic understands common mathematical, logical, and comparison operators.

Mathematical Operators

Mathematical operators can be applied to all numbers types, whereas the + operator can also be used to link strings.

+	Addition of numbers and date values, linking of strings
&	Link strings
–	Subtraction of numbers and date values
*	Multiplication of numbers
/	Division of numbers
\	Division of numbers with a whole number result (rounded)
^	Raising the power of numbers
MOD	modulo operation (calculation of the remainder of a division)

Although you can use the + operator to link strings, the + operator can become confused when linking a number to a string. The & operator is safer when dealing with strings because it assumes that all arguments should be strings, and converts the arguments to strings if they are not strings.

Logical Operators

Logical operators allow you to link elements according to the rules of Boolean algebra. If the operators are applied to Boolean values, the link provides the result required directly. If used in conjunction with integer and long integer values, the linking is done at the bit level.

AND	And linking
OR	Or linking
XOR	Exclusive or linking
NOT	Negation
EQV	Equivalent test (both parts `True` or `False`)
IMP	Implication (if the first expression is true, then the second must also be true)

Comparison Operators

Comparison operators can be applied to all elementary variable types (numbers, date details, strings, and Boolean values).

=	Equality of numbers, date values and strings
<>	Inequality of numbers, date values and strings
>	Greater than check for numbers, date values and strings
>=	Greater than or equal to check for numbers, date values and strings
<	Less than check for numbers, date values and strings
<=	Less than or equal to check for numbers, date values and strings

Note – OpenOffice.org Basic does ~~not~~ **now** support the VBA `Like` comparison operator.

Branching

Use branching statements to restrict the execution of a code block until a particular condition is satisfied.

If...Then...Else

The most common branching statement is the `If` statement as shown in the following example:

```
If A > 3 Then
  B = 2
End If
```

The `B = 2` assignment only occurs when value of variable `A` is greater than three. A variation of the `If` statement is the `If/Else` clause:

```
If A > 3 Then
  B = 2
Else
  B = 0
End If
```

In this example, the variable `B` is assigned the value of 2 when `A` is greater than 3, otherwise `B` is assigned the value of 0.

For more complex statements, you can cascade the `If` statement, for example:

```
If A = 0 Then
  B = 0
ElseIf A < 3 Then
  B = 1
Else
  B = 2
End If
```

If the value of variable `A` equals zero, `B` is assigned the value 0. If `A` is less than 3 (but not equal to zero), then `B` is assigned the value 1. In all other instances (that is, if `A` is greater than or equal to 3), `B` is assigned the value 2.

Select...Case

The `Select...Case` instruction is an alternative to the cascaded `If` statement and is used when you need to check a value against various conditions:

```
Select Case DayOfWeek
  Case 1:
    NameOfWeekday = "Sunday"
  Case 2:
    NameOfWeekday = "Monday"
  Case 3:
    NameOfWeekday = "Tuesday"
  Case 4:
    NameOfWeekday = "Wednesday"
  Case 5:
    NameOfWeekday = "Thursday"
  Case 6:
    NameOfWeekday = "Friday"
  Case 7:
    NameOfWeekday = "Saturday"
End Select
```

In this example, the name of a weekday corresponds to a number, so that the `DayOfWeek` variable is assigned the value of 1 for `Sunday`, 2 for `Monday` value, and so on.

The `Select` command is not restricted to simple 1:1 assignments — you can also specify comparison operators or lists of expressions in a `Case` branch. The following example lists the most important syntax variants:

```
Select Case Var
  Case 1 To 5
    ' ... Var is between the numbers 1 and 5 (including the values 1 and 5).
  Case > 100
    ' ... Var is greater than 100
  Case 6, 7, 8
    ' ... Var is 6, 7 or 8
  Case 6, 7, 8, > 15, < 0
    ' ... Var is 6, 7, 8, greater than 15, or less than 0
  Case Else
    ' ... all other instances
End Select
```

Now consider a misleading (advanced) example, and a common error:

```
Select Case Var
  Case Var = 8
    ' ... Var is 0
  Case Else
    ' ... all other instances
End Select
```

The statement (Var = 8) evaluates to TRUE if Var is 8, and FALSE otherwise. TRUE is -1 and FALSE is 0. The Select Case statement evaluates the expression, which is TRUE or FALSE, and then compares that value to Var. When Var is 0, there is a match. If you understand the last example, then you also know why this example does not do what it appears

```
Select Case Var
  Case Var > 8 And Var < 11
    ' ... Var is 0
  Case Else
    ' ... all other instances
End Select
```

Loops

A loop executes a code block for the number of passes that are specified. You can also have loops with an undefined number of passes.

For...Next

The `For...Next` loop has a fixed number of passes. The loop counter defines the number of times that the loop is to be executed. In the following example, variable `I` is the loop counter, with an initial value of 1. The counter is incremented by 1 at the end of each pass. When variable I equals 10, the loop stops.

```
Dim I
For I = 1 To 10
  ' ...  Inner part of loop
Next I
```

If you want to increment the loop counter by a value other than 1 at the end of each pass, use the `Step` function:

```
Dim I
For I = 1 To 10 Step 0.5
  ' ... Inner part of loop
Next I
```

In the preceding example, the counter is increased by 0.5 at the end of each pass and the loop is executed 19 times.

You can also use negative step values:

```
Dim I
For I = 10 To 1 Step -1
  ' ... Inner part of loop
Next I
```

In this example, the counter begins at 10 and is reduced by 1 at the end of each pass until the counter is 1.

The `Exit For` instruction allows you to exit a `For` loop prematurely. In the following example, the loop is terminated during the fifth pass:

```
Dim I
For I = 1 To 10
  If I = 5 Then
    Exit For
  End If
  ' ... Inner part of loop
Next I
```

For Each

The `For Each...Next` loop variation in VBA is supported in OpenOffice.org Basic. `For Each` loops do not use an explicit counter like a `For...Next` loop does. A `For Each` loop says "do this to everything in this set", rather than "do this n times". For example:

```
Const d1 = 2
Const d2 = 3
Const d3 = 2
Dim i
Dim a(d1, d2, d3)
For Each i In a()
    ' ... Inner part of loop
Next i
```

The loop will execute 36 times.

Do...Loop

The `Do...Loop` is not linked to a fixed number of passes. Instead, the `Do...Loop` is executed until a certain condition is met. There are four versions of the `Do...Loop`. In the first two examples, the code within the loop may not be executed at all ("do 0 times" logic). In the latter examples, the code will be executed at least once. (In the following examples, `A > 10` represents any condition):

1 **The `Do While...Loop` version**

```
Do While A > 10
   ' ... loop body
Loop
```

checks whether the condition after the `While` is **true** before every pass and only then executes the loop.

2 **The `Do Until...Loop` version**

```
Do Until A > 10
  ' ... loop body
Loop
```

executes the loop as long as the condition after the `Until` evaluates to **false**.

3 **The `Do...Loop While` version**

```
Do
```

```
  ' ... loop body
Loop While A > 10
```

only checks the condition after the first loop pass and terminates if the condition after the `While` evaluates to **false**.

4 The `Do...Loop Until` version

```
Do
  ' ... loop body
Loop Until A > 10
```

also checks its condition after the first pass, but terminates if the condition after the `Until` evaluates to **true**.

As in the `For...Next` loop, the `Do...Loop` also provides a terminate command. The `Exit Do` command can exit at loop at any point within the loop.

```
Do
  If A = 4 Then
    Exit Do
  End If
  ' ... loop body
Loop While A > 10
```

While...Wend

The `While...Wend` loop construct works exactly the same as the `Do While...Loop`, but with the disadvantage that there is no `Exit` command available. The following two loops produce identical results:

```
Do While A > 10
   ' ... loop body
Loop

While A > 10
   ' ... loop body
Wend
```

Programming Example: Sorting With Embedded Loops

There are many ways to use loops, for example, to search lists, return values, or execute complex mathematical tasks. The following example is an algorithm that uses two loops to sort a list by names.

```
Sub Sort
  Dim Entry(1 To 10) As String
  Dim Count As Integer
  Dim Count2 As Integer
  Dim Temp As String

  Entry(1) = "Patty"
  Entry(2) = "Kurt"
  Entry(3) = "Thomas"
  Entry(4) = "Michael"
  Entry(5) = "David"
  Entry(6) = "Cathy"
  Entry(7) = "Susie"
  Entry(8) = "Edward"
  Entry(9) = "Christine"
  Entry(10) = "Jerry"

  For Count = 1 To 9
    For Count2 = Count + 1 To 10
      If Entry(Count) > Entry(Count2) Then
        Temp = Entry(Count)
        Entry(Count) = Entry(Count2)
        Entry(Count2) = Temp
      End If
    Next Count2
  Next Count

  For Count = 1 To 10
    Print Entry(Count)
  Next Count

End Sub
```

The values are interchanged as pairs several times until they are finally sorted in ascending order. Like bubbles, the variables gradually migrate to the right position. For this reason, this algorithm is also known as a Bubble Sort.

Procedures

[Edited GW. The original document used the term *procedure* to refer to subprograms only, rather than a generic name for **subprograms** and **functions**.]

Procedures form pivotal points in the structure of a program. They provide the framework for dividing a complex problem into various sub-tasks.

Subprograms

A **subprogram** executes an action without providing an explicit value. Its syntax is

```
Sub Test
  ' ... here is the actual code of the subprogram
End Sub
```

The example defines a subprogram called `Test` that contains code that can be accessed from any point in the program. The call is made by entering the subprogram name at the relevant point of the program.

Functions

A **function**, just like a subprogram, combines a block of programs to be executed into one logical unit. However, unlike a subprogram, a function provides a return value.

```
Function Test
  ' ... here is the actual code of the function
  Test = 123
End Function
```

The return value is assigned using simple assignment. The assignment does not need to be placed at the end of the function, but can be made anywhere in the function.

The preceding function can be called within a program as follows:

```
Dim A
A = Test
```

The code defines a variable `A` and assigns the result of the `Test` function to it.

The return value can be overwritten several times within the function. As with classic variable assignment, the function in this example returns the value that was last assigned to it.

```
Function Test
  Test = 12
  ' ...
  Test = 123
End Function
```

In this example, the return value of the function is 123.

If nothing is assigned, the function returns a `zero` value (number 0 for numerical values and a blank for strings).

The return value of a function can be any type. The type is declared in the same way as a variable declaration:

```
Function Test As Integer
  ' ... here is the actual code of the function
End Function
```

Terminating subprograms and Functions Prematurely

In OpenOffice.org Basic, you can use the `Exit Sub` and `Exit Function` commands to terminate a subprogram or function prematurely, for example, for error handling. These commands stop the subprogram or function and return the program to the point at which the subprogram or function was called up.

The following example shows a subprogram which terminates implementation when the `ErrorOccured` variable has the value `True`.

```
Sub Test
  Dim ErrorOccured As Boolean
  ' ...
  If ErrorOccured Then
    Exit Sub
  End If
  ' ...
End Sub
```

Passing Parameters

Functions and subprograms can receive one or more parameters. Essential parameters must be enclosed in parentheses after the function or subprogram names. The following example defines a subprogram that expects an integer value `A` and a string `B` as parameters.

```
Sub Test (A As Integer, B As String)
  ' ...
End Sub
```

Parameters are normally passed by **Reference** in OpenOffice.org Basic. Changes made to the variables are retained when the subprogram or function is exited:

```
Sub Test
  Dim A As Integer
  A = 10
  ChangeValue(A)
  ' The parameter A now has the value 20
End Sub

Sub ChangeValue(TheValue As Integer)
  TheValue = 20
End Sub
```

In this example, the value `A` that is defined in the `Test` function is passed as a parameter to the `ChangeValue` function. The value is then changed to 20 and passed to `TheValue`, which is retained when the function is exited.

You can also pass a parameter as a **value** if you do not want subsequent changes to the parameter to affect the value that is originally passed. To specify that a parameter is to be passed as a value, ensure that the `ByVal` keyword precedes the variable declaration in the function header.

In the preceding example, if we replace the `ChangeValue` function then the superordinate variable A remains unaffected by this change. After the call for the `ChangeValue` function, variable `A` retains the value 10.

```
Sub ChangeValue(ByVal TheValue As Integer)
  TheValue = 20
End Sub
```

Note – The method for passing parameters to subprograms and functions in OpenOffice.org Basic is virtually identical to that in VBA. By default, the parameters are passed by reference. To pass parameters as values, use the `ByVal` keyword. In VBA, you can also use the keyword `ByRef` to force a parameter to be passed by reference. OpenOffice.org Basic recognizes but ignores this keyword, because this is already the default behaviour in OpenOffice.org Basic.

Optional Parameters

Functions and subprograms can only be called up if all the necessary parameters are passed during the call.

OpenOffice.org Basic lets you define parameters as **optional**, that is, if the corresponding values are not included in a call, OpenOffice.org Basic passes an empty parameter. In the following example the `A` parameter is obligatory, whereas the `B` parameter is optional.

```
Sub Test(A As Integer, Optional B As Integer)
  ' ...
End Sub
```

The `IsMissing` function checks whether a parameter has been passed or is left out.

```
Sub Test(A As Integer, Optional B As Integer)
  Dim B_Local As Integer
  ' Check whether B parameter is actually present
  If Not IsMissing (B) Then
    B_Local = B      ' B parameter present
  Else
    B_Local = 0      ' B parameter missing -> default value 0
  End If
  ' ... Start the actual function
End Sub
```

The example first tests whether the `B` parameter has been passed and, if necessary, passes the same parameter to the internal `B_Local` variable. If the corresponding parameter is not present, then a default value (in this instance, the value 0) is passed to `B_Local` rather than the passed parameter.

Note – The `ParamArray` keyword present in VBA is not supported in OpenOffice.org Basic.

Recursion

A recursive subprogram or function is one that has the ability to call itself until it detects that some base condition has been satisfied. When the function is called with the base condition, a result is returned.

The following example uses a recursive function to calculate the factorial of the numbers `42`, `-42`, and `3.14`:

```
Sub Main
  Msgbox CalculateFactorial(  42 )    ' Displays 1,40500611775288E+51
  Msgbox CalculateFactorial( -42 )    ' Displays "Invalid number for factorial!"
  Msgbox CalculateFactorial( 3.14 )   ' Displays "Invalid number for factorial!"
End Sub

Function CalculateFactorial( Number )
  If Number < 0 Or Number <> Int( Number ) Then
    CalculateFactorial = "Invalid number for factorial!"
  ElseIf Number = 0 Then
    CalculateFactorial = 1
  Else
    ' This is the recursive call:
    CalculateFactorial = Number * CalculateFactorial( Number - 1 )
  Endif
End Function
```

The example returns the factorial of the number `42` by recursively calling the `CalculateFactorial` function until it reaches the base condition of `0! = 1`.

Note – The recursion levels are set at different levels based on the software platform. For Windows the recursion level is 5800. For Solaris and Linux, an evaluation of the stacksize is performed and the recursion level is calculated.

Error Handling

Correct handling of error situations is one of the most time-consuming tasks of programming. OpenOffice.org Basic provides a range of tools for simplifying error handling.

The On Error Instruction

The `On Error` instruction is the key to any error handling:

```
Sub Test
  On Error Goto ErrorHandler
  ' ... undertake task during which an error may occur
  Exit Sub
  ErrorHandler:
    ' ... individual code for error handling
End Sub
```

The `On Error Goto ErrorHandler` line defines how OpenOffice.org Basic proceeds in the event of an error. The `Goto ErrorHandler` ensures that OpenOffice.org Basic exits the current program line and then executes the `ErrorHandler:` code.

The Resume Command

The `Resume Next` command continues the program from the line that follows where the error occurred in the program after the code in the error handler has been executed:

```
ErrorHandler:
  ' ... individual code for error handling
  Resume Next
```

Use the `Resume Proceed` command to specify a jump point for continuing the program after error handling:

```
ErrorHandler:
  ' ... individual code for error handling
  Resume Proceed

Proceed:
  ' ... the program continues here after the error
```

To continue a program without an error message when an error occurs, use the following format:

```
Sub Test
  On Error Resume Next
  ' ... perform task during which an error may occur
End Sub
```

Use the `On Error Resume Next` command with caution as its effect is global.

Queries Regarding Error Information

In error handling, it is useful to have a description of the error and to know where and why the error occurred:

- The `Err` variable contains the number of errors that has occurred.
- The `Error$` variable contains a description of the error.

- The `Erl` variable contains the line number where the error occurred.

The call

```
MsgBox "Error " & Err & ": " & Error$ & " (line : " & Erl & ")"
```

shows how the error information can be displayed in a message window.

Note – Whereas VBA summarizes the error messages in a statistical object called `Err`, OpenOffice.org Basic provides the `Err`, `Error$`, and `Erl` variables.

The status information remains valid until the program encounters a `Resume` or `On Error` command, whereupon the information is reset.

Note – In VBA, the `Err.Clear` method of the `Err` object resets the error status after an error occurs. In OpenOffice.org Basic, this is accomplished with the `On Error` or `Resume` commands.

Tips for Structured Error Handling

Both the definition command, `On Error`, and the return command, `Resume`, are variants of the `Goto` construct.

If you want to cleanly structure your code to prevent generating errors when you use this construct, you should not use jump commands without monitoring them.

Care should be taken when you use the `On Error Resume Next` command as this dismisses all open error messages.

The best solution is to use only one approach for error handling within a program - keep error handling separate from the actual program code and do not jump back to the original code after the error occurs.

The following code is an example of an error handling subprogram:

```
Sub Example
  ' Define error handler at the start of the function
  On Error Goto ErrorHandler
    ' ... Here is the actual program code
  On Error Goto 0           ' Deactivate error handling
```

```
    ' End of regular program implementation
  Exit Sub

 ' Start point of error handling
 ErrorHandler:
   ' Check whether error was expected
    If Err = ExpectedErrorNo Then
      ' ... Process error
    Else
      ' ... Warning of unexpected error
    End If
 On Error Goto 0            ' Deactivate error handling
End Sub
```

This subprogram begins with the definition of an error handler, followed by the actual program code. At the end of the program code, the error handling is deactivated by the `On Error Goto 0` call and the subprogram implementation is ended by the `Exit Sub` command (not to be confused with `End Sub`).

The example first checks if the error number corresponds to the expected number (as stored in the imaginary `ExpectedErrorNo` constant) and then handles the error accordingly. If another error occurs, the system outputs a warning. It is important to check the error number so that unanticipated errors can be detected.

The `On Error Goto 0` call at the end of the code resets the status information of the error (the error code in the `Err` system variables) so that an error occurring at a later date can be clearly recognized.

Other Instructions

Type...End Type

A *struct* is a collection of data fields, that can be manipulated as a single item. In older terms, you may think of a struct as a record, or part of a record.

The API often uses pre-defined structs, but these are *UNO structs*, a highly-specialized kind of struct.

Definition

With the `Type...End Type` statements, you can define your own (non-UNO) structs:

```
Type aMenuItem                          'assign the name of the type
    'Define the data fields within the struct. Each
    ' definition looks like a Dim statement, without the "Dim".
    aCommand as String
    aText as String
End Type                          'close the definition
```

Instance

The `Type` definition is only a pattern or template, not a set of actual variables. To make an *instance* of the type, actual variables that can be read and stored, use the `Dim as New` statement:

```
Dim maItem as New aMenuItem
```

Scope

As shown in the example below, the `Type` definition may be written at the start of a module (before the first `Sub` or `Function`). The definition will then be available to all routines in the module.

As of OpenOffice.org Version 3.0, unlike variables, there is no way to make the definition accessible outside of the module.

An instance of the new type *is* a variable, and follows the usual rules for variable scope (see Scope and Life Span of Variables).

An example of how to use the definition, and how to reference the fields within an instance, appears in the section on `With...End With`.

With...End With

Qualifiers

In general, Basic does not look inside a container, such as an `Object`, to see what names might be defined there. If you want to use such a name, you must tell Basic where to look. You do that by using the name of the object as a *qualifier*. Write it before the inner name, and separate it by a period:

```
MyObject.SomeName
```

Since containers may hold other containers, you may need more than one qualifier. Write the qualifiers in order, from outer to inner:

```
OuterObject.InnerObject.FarInsideObject.SomeName
```

These names may also be described as, "concatenated with the dot-operator ('.')".

The With Alternative

The `With...End With` bracketing statements provide an alternative to writing out all the qualifiers, every time - and some of the qualifiers in the API can be quite long. You specify the qualifiers in the `With` statement. Until Basic encounters the `End With` statement, it looks for *partly-qualified* names: names that begin with a period (unary dot-operator). The compiler uses the qualifiers from the `With` as though they were written in front of the partly-qualified name.

Example 1: A User-defined Struct

This example shows how you may define and use a struct, and how to reference the items within it, both with and without `With`. Either way, the names of the data fields (from the `Type` definition) must be qualified by the name of the instance (from the `Dim` statement).

```
Type aMenuItem
    aCommand as String
    aText as String
End Type

Sub Main
```

```
    'Create an instance of the user-defined struct.
    ' Note the keyword, "New".
    Dim maItem as New aMenuItem
    With maItem
        .aCommand = ".uno:Copy"
        .aText = "~Copy"
    End With

    MsgBox     "Command: " & maItem.aCommand & Chr(13) _
            & "Text: " & maItem.aText
End Sub
```

Example 2: Case statement

In Cells and Ranges, the following example has the qualifiers in the `Case` statements written out completely, for clarity. You can write it more easily, this way:

```
Dim Doc As Object
Dim Sheet As Object
Dim Cell As Object

Doc = StarDesktop.CurrentComponent
Sheet = Doc.Sheets(0)
Cell = Sheet.getCellByPosition(1,1)        'Cell "B2" (0-based!)

Cell.Value = 1000

With com.sun.star.table.CellContentType
  Select Case Cell.Type
    Case .EMPTY
      MsgBox "Content: Empty"
    Case .VALUE
      MsgBox "Content: Value"
    Case .TEXT
      MsgBox "Content: Text"
    Case .FORMULA
     MsgBox "Content: Formula"
  End Select
End With
```

Notice that the `With` construct must be entirely outside of the `Select` construct.

◆ ◆ ◆ CHAPTER 3

The Runtime Library of OpenOffice.org Basic

The following sections present the central functions of the runtime library:

- Conversion Functions
- Strings
- Date and Time
- Files and Directories
- Message and Input Boxes
- Other Functions

Conversion Functions

In many situations, circumstances arise in which a variable of one type has to be changed into a variable of another type.

Implicit and Explicit Type Conversions

The easiest way to change a variable from one type to another is to use an assignment.

```
Dim A As String
Dim B As Integer

B = 101
A = B
```

In this example, variable A is a string, and variable B is an integer. OpenOffice.org Basic ensures that variable B is converted to a string during assignment to variable A. This conversion is much more elaborate than it appears: the integer B remains in the working memory in the form of a two-byte long number. A, on the other hand, is a string, and the computer saves a one- or two-byte long value for each character (each number). Therefore, before copying the content from B to A, B has to be converted into A's internal format.

Unlike most other programming languages, Basic performs type conversion automatically. However, this may have fatal consequences. Upon closer inspection, the following code sequence

```
Dim A As String
Dim B As Integer
Dim C As Integer

B = 1
C = 1
A = B + C
```

which at first glance seems straightforward, ultimately proves to be something of a trap. The Basic interpreter first calculates the result of the addition process and then converts this into a string, which, as its result, produces the string 2.

If, on the other hand, the Basic interpreter first converts the start values B and C into a string and applies the plus operator to the result, it produces the string 11.

The same applies when using variant variables:

```
Dim A
Dim B
Dim C

B = 1
C = "1"
A = B + C
```

Since variant variables may contain both numbers and strings, it is unclear whether variable A

is assigned the number 2 or the string 11.

The error sources noted for implicit type conversions can only be avoided by careful programming; for example, by not using the variant data type.

To avoid other errors resulting from implicit type conversions, OpenOffice.org Basic offers a range of conversion functions, which you can use to define when the data type of an operation should be converted:

CStr(Var)
converts any data type into a string.

CInt(Var)
converts any data types into an integer value.

CLng(Var)
converts any data types into a long value.

CSng(Var)
converts any data types into a single value.

CDbl(Var)
converts any data types into a double value.

CBool(Var)
converts any data types into a Boolean value.

CDate(Var)
converts any data types into a date value.

You can use these conversion functions to define how OpenOffice.org Basic should perform these type conversion operations:

```
Dim A As String
Dim B As Integer
Dim C As Integer

B = 1
C = 1
A = CStr(B + C)         ' B and C are added together first, then
                        ' converted to the string "2"
A = CStr(B) + CStr(C)   ' B and C are converted into a string,then
                        ' combined to produce the string "11"
```

During the first addition in the example, OpenOffice.org Basic first adds the integer variables and then converts the result into a chain of characters. `A` is assigned the string `2`. In the second instance, the integer variables are first converted into two strings and then linked with one

another by means of the assignment. `A` is therefore assigned the string `11`.

The numerical `CSng` and `CDbl` conversion functions also accept decimal numbers. The symbol defined in the corresponding country-specific settings must be used as the decimal point symbol. Conversely, the `CStr` methods use the currently selected country-specific settings when formatting numbers, dates and time details.

The `Val` function is different from the `Csng, Cdbl` and `Cstr` methods. It converts a string into a number; however it always expects a period to be used as the decimal point symbol.

```
Dim A As String
Dim B As Double

A = "2.22"
B = Val(A)      ' Is converted correctly regardless of the
                ' country-specific settings
```

Checking the Content of Variables

In some instances, the date cannot be converted:

```
Dim A As String
Dim B As Date

A = "test"
B = A           ' Creates error message
```

In the example shown, the assignment of the `test` string to a date variable makes no sense, so the Basic interpreter reports an error. The same applies when attempting to assign a string to a Boolean variable:

```
Dim A As String
Dim B As Boolean

A = "test"
B = A           ' Creates error message
```

Again, the basic interpreter reports an error.

These error messages can be avoided by checking the program before an assignment, in order to establish whether the content of the variable to be assigned matches the type of the target variable. OpenOffice.org Basic provides the following test functions for this purpose:

`IsNumeric(Value)`
: checks whether a value is a number.

IsDate(Value)
checks whether a value is a date.

IsArray(Value)
checks whether a value is an array.

These functions are especially useful when querying user input. For example, you can check whether a user has typed a valid number or date.

```
If IsNumeric(UserInput) Then
  ValidInput = UserInput
Else
  ValidInput = 0
  MsgBox "Error message."
End If
```

In the previous example, if the `UserInput` variable contains a valid numerical value, then this is assigned to the `ValidInput` variable. If `UserInput` does not contain a valid number, `ValidInput` is assigned the value `0` and an error message is returned.

While test functions exist for checking numbers, date details and arrays in OpenOffice.org Basic, a corresponding function for checking Boolean values does not exist. The functionality can, however, be imitated by using the `IsBoolean` function:

```
Function IsBoolean(Value As Variant) As Boolean
  On Error Goto ErrorIsBoolean:
  Dim Dummy As Boolean
  Dummy = Value
  IsBoolean = True
  On Error Goto 0
  Exit Sub

  ErrorIsBoolean:
  IsBoolean = False
  On Error Goto 0
End Function
```

The `IsBoolean` function defines an internal `Dummy` help variable of the Boolean type and tries to assign this to the transferred value. If assignment is successful, the function returns `True`. If it fails, a runtime error is produced, the error handler intercepts the error, and the function returns `False`.

Note – If a string in OpenOffice.org Basic contains a non-numerical value and if this is assigned to a number, OpenOffice.org Basic does not produce an error message, but stops converting the string at the first invalid character. This procedure differs from VBA. There, an error is triggered and program implementation terminated if a corresponding assignment is executed.

Strings

Working with Sets of Characters

When administering strings, OpenOffice.org Basic uses the set of Unicode characters. The `Asc` and `Chr` functions allow the Unicode value belonging to a character to be established and/or the corresponding character to be found for a Unicode value. The following expressions assign the various Unicode values to the code variable:

```
Code = Asc("A")         ' Latin letter A (Unicode-value 65)
Code = Asc("€")         ' Euro character (Unicode-value 8364)
Code = Asc("Л")         ' Cyrillic letter Л (Unicode-value 1083)
```

Conversely, the expression

```
MyString = Chr(13)
```

ensures that the `MyString` string is initialized with the value of the number `13`, which stands for a hard line break.

The `Chr` command is often used in Basic languages to insert control characters in a string. The assignment

```
MyString = Chr(9) + "This is a test" + Chr(13)
```

therefore ensures that the text is preceded by a tab character (Unicode-value 9) and that a hard line break (Unicode-value 13) is added after the text.

Accessing Parts of a String

OpenOffice.org Basic provides three functions that return partial strings, plus a length function:

`Left(MyString, Length)`
: returns the first Length characters of `MyString`.

`Right(MyString, Length)`
: returns the last Length characters of `MyString`.

`Mid(MyString, Start, Length)`
: returns first Length characters of `MyString` as of the `Start` position.

`Len(MyString)`
: returns the number of characters in `MyString`.

Here are a few example calls for the named functions:

```
Dim MyString As String
Dim MyResult As String
Dim MyLen As Integer

MyString = "This is a small test"
MyResult = Left(MyString,5)      ' Provides the string "This "
MyResult = Right(MyString, 5)    ' Provides the string " test"
MyResult = Mid(MyString, 8, 5)   ' Provides the string " a sm"
MyLen = Len(MyString)            ' Provides the value 20
```

Search and Replace

OpenOffice.org Basic provides the `InStr` function for searching for a partial string within another string:

```
ResultString = InStr (MyString, SearchString)
```

The `SearchString` parameter specifies the string to be searched for within `MyString`. The function returns a number that contains the position at which the `SearchString` first appears within `MyString`. If you want to find other matches for the string, the function also provides the opportunity to specify an optional start position from which OpenOffice.org Basic begins the search. In this case, the syntax of the function is:

```
ResultString = InStr(StartPosition, MyString, SearchString)
```

In the previous examples, `InStr` ignores uppercase and lowercase characters. To change the search so that `InStr` is case sensitive, add the parameter `0`, as shown in the following example:

```
ResultString = InStr(MyString, SearchString, 0)
```

Using the previous functions for editing strings, programmers can search for and replace one string in another string:

```
Function Replace(Source As String, Search As String, NewPart As String)
  Dim Result As String
  Dim StartPos As Long
  Dim CurrentPos As Long
```

```
  Result = ""
  StartPos = 1
  CurrentPos = 1

  If Search = "" Then
    Result = Source
  Else
    Do While CurrentPos <> 0
      CurrentPos = InStr(StartPos, Source, Search)
      If CurrentPos <> 0 Then
        Result = Result + Mid(Source, StartPos, _
        CurrentPos - StartPos)
        Result = Result + NewPart
        StartPos = CurrentPos + Len(Search)
      Else
        Result = Result + Mid(Source, StartPos, Len(Source))
      End If                ' Position <> 0
    Loop
  End If

  Replace = Result
End Function
```

The function searches through the transferred `Search` string in a loop by means of `InStr` in the original term `Source`. If it finds the search term, it takes the part before the expression and writes it to the `Result` return buffer. It adds the `NewPart` section at the point of the search term `Search`. If no more matches are found for the search term, the function establishes the part of the string still remaining and adds this to the return buffer. It returns the string produced in this way as the result of the replacement process.

Since replacing parts of character sequences is one of the most frequently used functions, the `Mid` function in OpenOffice.org Basic has been extended so that this task is performed automatically. The following example replaces three characters with the string `is` from the sixth position of the `MyString` string.

```
Dim MyString As String

MyString = "This was my text"
Mid(MyString, 6, 3, "is")
```

Formatting Strings

The `Format` function formats numbers as a string. To do this, the function expects a `Format` expression to be specified, which is then used as the template for formatting the numbers. Each place holder within the template ensures that this item is formatted correspondingly in the output value. The five most important place holders within a template are the zero (`0`), pound sign (`#`), period (`.`), comma (`,`) and dollar sign (`$`) characters.

The `0` character within the template ensures that a number is always placed at the corresponding point. If a number is not provided, 0 is displayed in its place.

A `.` stands for the decimal point symbol defined by the operating system in the country-specific settings.

The example below shows how the `0` and `.` characters can define the digits after the decimal point in an expression:

```
MyFormat = "0.00"
MyString = Format(-1579.8, MyFormat)     ' Provides "-1579,80"
MyString = Format(1579.8, MyFormat)      ' Provides "1579,80"
MyString = Format(0.4, MyFormat)         ' Provides "0,40"
MyString = Format(0.434, MyFormat)       ' Provides "0,43"
```

In the same way, zeros can be added in front of a number to achieve the desired length:

```
MyFormat = "0000.00"
MyString = Format(-1579.8, MyFormat)     ' Provides "-1579,80"
MyString = Format(1579.8, MyFormat)      ' Provides "1579,80"
MyString = Format(0.4, MyFormat)         ' Provides "0000,40"
MyString = Format(0.434, MyFormat)       ' Provides "0000,43"
```

A `,` represents the character that the operating system uses for a thousands separator, and the `#` stands for a digit or place that is only displayed if it is required by the input string.

```
MyFormat = "#,##0.00"
MyString = Format(-1579.8, MyFormat)     ' Provides "-1.579,80"
MyString = Format(1579.8, MyFormat)      ' Provides "1.579,80"
MyString = Format(0.4, MyFormat)         ' Provides "0,40"
MyString = Format(0.434, MyFormat)       ' Provides "0,43"
```

In place of the `$` place holder, the `Format` function displays the relevant currency symbol defined by the system (this example assumes a European locale has been defined):

```
MyFormat = "#,##0.00 $"
MyString = Format(-1579.8, MyFormat)     ' Provides "-1.579,80 €"
MyString = Format(1579.8, MyFormat)      ' Provides "1.579,80 €"
MyString = Format(0.4, MyFormat)         ' Provides "0,40 €"
MyString = Format(0.434, MyFormat)       ' Provides "0,43 €"
```

The format instructions used in VBA for formatting date and time details can also be used:

```
sub main
    dim myDate as date
    myDate = "01/06/98"
    TestStr = Format(myDate, "mm-dd-yyyy") ' 01-06-1998
    MsgBox TestStr
end sub
```

Date and Time

OpenOffice.org Basic provides the `Date` data type, which saves the date and time details in binary format.

Specification of Date and Time Details within the Program Code

You can assign a date to a date variable through the assignment of a simple string:

```
Dim MyDate As Date
MyDate = "24.1.2002"
```

This assignment can function properly because OpenOffice.org Basic automatically converts the date value defined as a string into a date variable. This type of assignment, however, can cause errors, date and time values are defined and displayed differently in different countries.

Since OpenOffice.org Basic uses the country-specific settings of the operating system when converting a string into a date value, the expression shown previously only functions correctly if the country-specific settings match the string expression.

To avoid this problem, the `DateSerial` function should be used to assign a fixed value to a date variable:

```
Dim MyVar As Date
MyDate = DateSerial (2001, 1, 24)
```

The function parameter must be in the sequence: year, month, day. The function ensures that the variable is actually assigned the correct value regardless of the country-specific settings

The `TimeSerial` function formats time details in the same way that the `DateSerial` function formats dates:

```
Dim MyVar As Date
MyDate =  TimeSerial(11, 23, 45)
```

Their parameters should be specified in the sequence: hours, minutes, seconds.

Extracting Date and Time Details

The following functions form the counterpart to the `DateSerial` and `TimeSerial` functions:

`Day(MyDate)`
: returns the day of the month from `MyDate`.

`Month(MyDate)`
: returns the month from `MyDate`.

`Year(MyDate)`
: returns the year from `MyDate`.

`Weekday(MyDate)`
: returns the number of the weekday from `MyDate`.

`Hour(MyTime)`
: returns the hours from `MyTime`.

`Minute(MyTime)`
: returns the minutes from `MyTime`.

`Second(MyTime)`
: returns the seconds from `MyTime`.

These functions extract the date or time sections from a specified `Date` variable. The following example checks whether the date saved in `MyDate` is in the year 2003.

```
Dim MyDate As Date
' ... Initialization of MyDate

If Year(MyDate) = 2003 Then
  ' ... Specified date is in the year 2003
End If
```

In the same way, the following example checks whether `MyTime` is between 12 and 14 hours.

```
Dim MyTime As Date
' ... Initialization of MyTime

If Hour(MyTime) >= 12 And Hour(MyTime) < 14 Then
  ' ... Specified time is between 12 and 14 hours
End If
```

The `Weekday` function returns the number of the weekday for the transferred date:

```
Dim MyDate As Date
Dim MyWeekday As String
' ... initialize MyDate

Select Case WeekDay(MyDate)
  case 1
    MyWeekday = "Sunday"
  case 2
    MyWeekday = "Monday"
  case 3
    MyWeekday = "Tuesday"
  case 4
    MyWeekday = "Wednesday"
  case 5
    MyWeekday = "Thursday"
  case 6
    MyWeekday = "Friday"
  case 7
    MyWeekday = "Saturday"
End Select
```

Note – Sunday is considered the first day of the week.

Retrieving System Date and Time

The following functions are available in OpenOffice.org Basic to retrieve the system time and system date:

`Date`
: returns the present date as a string. The format depends on localization settings.

`Time`
: returns the present time as a string.

`Now`
: returns the present point in time (date and time) as a combined value of type `Date`.

Files and Directories

Working with files is one of the basic tasks of an application. The OpenOffice.org API provides you with a whole range of objects with which you can create, open and modify Office documents. These are presented in detail in the Introduction to the OpenOffice.org API. Regardless of this, in some instances you will have to directly access the file system, search through directories or edit text files. The runtime library from OpenOffice.org Basic provides several fundamental functions for these tasks.

Note – Some DOS-specific file and directory functions are no longer provided in OpenOffice.org, or their function is only limited. For example, support for the `ChDir`, `ChDrive` and `CurDir` functions is not provided. Some DOS-specific properties are no longer used in functions that expect file properties as parameters (for example, to differentiate from concealed files and system files). This change became necessary to ensure the greatest possible level of platform independence for OpenOffice.org.

Administering Files

Compatibility Mode

The `CompatibilityMode` statement and function provide greater compatibility with VBA, by changing the operation of certain functions. The effect on any particular function is described with that function, below.

As a statement, `CompatibilityMode( value )` takes a Boolean value to set or clear the mode. As a function, `CompatibilityMode()` returns the Boolean value of the mode.

```
CompatibilityMode( True ) 'set mode
CompatibilityMode( False) 'clear mode

Dim bMode as Boolean
bMode = CompatibilityMode()
```

Searching Through Directories

The `Dir` function in OpenOffice.org Basic is responsible for searching through directories for files and sub-directories. When first requested, a string containing the path of the directories to be searched must be assigned to `Dir` as its first parameter. The second parameter of `Dir` specifies the file or directory to be searched for. OpenOffice.org Basic returns the name of the first directory entry found. To retrieve the next entry, the `Dir` function should be requested without parameters. If the `Dir` function finds no more entries, it returns an empty string.

The following example shows how the `Dir` function can be used to request all files located in one directory. The subprogram saves the individual file names in the `AllFiles` variable and then displays this in a message box.

```
Sub ShowFiles
  Dim NextFile As String
  Dim AllFiles As String

  AllFiles = ""
  NextFile = Dir("C:\", 0)

  While NextFile  <> ""
    AllFiles = AllFiles & Chr(13) &  NextFile
    NextFile = Dir
  Wend

  MsgBox AllFiles
End Sub
```

The `0` (zero) used as the second parameter in the `Dir` function ensures that `Dir` only returns the names of files and directories are ignored. The following parameters can be specified here:

- `0` : returns normal files
- `16` : sub-directories

The following example is virtually the same as the preceding example, but the `Dir` function transfers the value 16 as a parameter, which returns the sub-directories of a folder rather than the file names.

```
Sub ShowDirs
  Dim NextDir As String
  Dim AllDirs As String

  AllDirs = ""
  NextDir = Dir("C:\", 16)

  While NextDir <> ""
    AllDirs = AllDirs & Chr(13) &  NextDir
    NextDir = Dir
  Wend
```

```
  MsgBox AllDirs
End Sub
```

Note – When requested in OpenOffice.org Basic, the `Dir` function, using the parameter 16, only returns the sub-directories of a folder. In VBA, the function also returns the names of the standard files so that further checking is needed to retrieve the directories only. When using the `CompatibilityMode ( true )` function, OpenOffice.org Basic behaves like VBA and the Dir function, using parameter 16, returns sub-directories and standard files.

Note – The options provided in VBA for searching through directories specifically for files with the **concealed**, **system file**, **archived**, and **volume name** properties does not exist in OpenOffice.org Basic because the corresponding file system functions are not available on all operating systems.

Note – The path specifications listed in `Dir` may use the * and ? place holders in both VBA and OpenOffice.org Basic. In OpenOffice.org Basic, the * place holder may however only be the last character of a file name and/or file extension, which is not the case in VBA.

Creating and Deleting Directories

OpenOffice.org Basic provides the `MkDir` function for creating directories.

```
MkDir ("C:\SubDir1")
```

This function creates directories and sub-directories. All directories needed within a hierarchy are also created, if required. For example, if only the `C:\SubDir1` directory exists, then a call

```
MkDir ("C:\SubDir1\SubDir2\SubDir3\")
```

creates both the `C:\SubDir1\SubDir2` directory and the `C:\SubDir1\SubDir2\SubDir3` directory.

The `RmDir` function deletes directories.

```
RmDir ("C:\SubDir1\SubDir2\SubDir3\")
```

If the directory contains sub-directories or files, these are **also deleted.** You should therefore be careful when using `RmDir`.

Note – In VBA, the `MkDir` and `RmDir` functions only relate to the current directory. In OpenOffice.org Basic on the other hand, `MkDir` and `RmDir` can be used to create or delete levels of directories.

Note – In VBA, `RmDir` produces an error message if a directory contains a file. In OpenOffice.org Basic, the directory **and all its files** are deleted. If you use the `CompatibilityMode ( true )` function, OpenOffice.org Basic will behave like VBA.

Copying, Renaming, Deleting and Checking the Existence of Files

The following call creates a copy of the `Source` file under the name of `Destination`:

```
FileCopy(Source, Destination)
```

With the help of the following function you can rename the `OldName` file with `NewName`. The `As` keyword syntax, and the fact that a comma is not used, goes back to the roots of the Basic language.

```
Name OldName As NewName
```

The following call deletes the `Filename` file. If you want to delete directory (including its files) use the `RmDir` function.

```
Kill(Filename)
```

The `FileExists` function can be used to check whether a file exists:

```
If FileExists(Filename) Then
  MsgBox "file exists."
End If
```

Reading and Changing File Properties

When working with files, it is sometimes important to be able to establish the file properties, the time the file was last changed and the length of the file.

The following call returns some properties about a file.

```
Dim Attr As Integer
Attr = GetAttr(Filename)
```

The return value is provided as a bit mask in which the following values are possible:

- 1 : read-only file
- 16 : name of a directory

The following example determines the bit mask of the `test.txt` file and checks whether this is read-only whether it is a directory. If neither of these apply, `FileDescription` is assigned the "normal" string.

```
Dim FileMask As Integer
Dim FileDescription As String

FileMask = GetAttr("test.txt")

If (FileMask AND 1) > 0 Then
  FileDescription = FileDescription & " read-only "
End IF

If (FileMask AND 16) > 0 Then
  FileDescription = FileDescription & " directory "
End IF

If FileDescription = "" Then
  FileDescription = " normal "
End IF

MsgBox FileDescription
```

Note – The flags used in VBA for querying the **concealed**, **system file,archived** and **volume name** file properties are not supported in OpenOffice.org Basic because these are Windows-specific and are not or are only partially available on other operating systems.

The `SetAttr` function permits the properties of a file to be changed. The following call can therefore be used to provide a file with read-only status:

```
SetAttr("test.txt", 1)
```

An existing read-only status can be deleted with the following call:

```
SetAttr("test.txt", 0)
```

The date and time of the last amendment to a file are provided by the `FileDateTime` function. The date is formatted here in accordance with the country-specific settings used on the system.

```
FileDateTime("test.txt")   ' Provides date and time of the last file amendment.
```

The `FileLen` function determines the length of a file in bytes (as long integer value).

```
FileLen("test.txt")      ' Provides the length of the file in bytes
```

Writing and Reading Text Files

OpenOffice.org Basic provides a whole range of methods for reading and writing files. The following explanations relate to working with text files (**not** text documents).

Writing Text Files

Before a text file is accessed, it must first be opened. To do this, a free **file handle** is needed, which clearly identifies the file for subsequent file access.

The `FreeFile` function is used to create a free file handle:

```
FileNo = FreeFile
```

`FileNo` is an integer variable that receives the file handle. The handle is then used as a parameter for the `Open` instruction, which opens the file.

To open a file so that it can be written as a text file, the `Open` call is:

```
Open Filename For Output As #FileNo
```

`Filename` is a string containing the name of the file. `FileNo` is the handle created by the `FreeFile` function.

Once the file is opened, the `Print` instruction can create the file contents, line by line:

```
Print #FileNo, "This is a test line."
```

`FileNo` also stands for the file handle here. The second parameter specifies the text that is to be saved as a line of the text file.

Once the writing process has been completed, the file must be closed using a `Close` call:

```
Close #FileNo
```

Again here, the file handle should be specified.

The following example shows how a text file is opened, written, and closed:

```
Dim FileNo As Integer
Dim CurrentLine As String
Dim Filename As String

Filename = "c:\data.txt"            ' Define file name
FileNo = FreeFile               ' Establish free file handle
```

```
Open Filename For Output As #FileNo         ' Open file (writing mode)
Print #FileNo, "This is a line of text"      ' Save line
Print #FileNo, "This is another line of text"   ' Save line
Close #FileNo                  ' Close file
```

Reading Text Files

Text files are read in the same way that they are written. The `Open` instruction used to open the file contains the `For Input` expression in place of the `For Output` expression and, rather than the Print command for writing data, the Line Input instruction should be used to read the data.

Finally, when calling up a text file, the `eof` instruction is used to check whether the end of the file has been reached:

```
eof(FileNo)
```

The following example shows how a text file can be read:

```
Dim FileNo As Integer
Dim CurrentLine As String
Dim File As String
Dim Msg as String

' Define filename
Filename = "c:\data.txt"

' Establish free file handle
FileNo = Freefile

' Open file (reading mode)
Open Filename For Input As FileNo

' Check whether file end has been reached
Do While not eof(FileNo)
  ' Read line
  Line Input #FileNo, CurrentLine
  If CurrentLine <>"" then
    Msg = Msg & CurrentLine & Chr(13)
  end if
Loop


' Close file

Close #FileNo
Msgbox Msg
```

The individual lines are retrieved in a `Do While` loop, saved in the `Msg` variable, and displayed at the end in a message box.

Message and Input Boxes

OpenOffice.org Basic provides the `MsgBox` and `InputBox` functions for basic user communication.

Displaying Messages

`MsgBox` displays a basic information box, which can have one or more buttons. In its simplest variant the `MsgBox` only contains text and an OK button:

```
MsgBox "This is a piece of information!"
```

The appearance of the information box can be changed using a parameter. The parameter provides the option of adding additional buttons, defining the pre-assigned button, and adding an information symbol.

Note – By convention, the symbolic names given below are written in UPPERCASE, to mark them as predefined, rather than user-defined. However, the names are not case-sensitive.

The values for selecting the buttons are:

- `0, MB_OK` - OK button
- `1, MB_OKCANCEL` - OK and Cancel button
- `2, MB_ABORTRETRYIGNORE` - Abort, Retry, and Ignore buttons
- `3, MB_YESNOCANCEL` - Yes, No, and Cancel buttons
- `4, MB_YESNO` - Yes and No buttons
- `5, MB_RETRYCANCEL` - Retry and Cancel buttons

To set a button as the default button, add one of the following values to the parameter value from the list of button selections. For example, to create Yes, No and Cancel buttons (value 3) where Cancel is the default (value 512), the parameter value is 3 + 512 = 515. The expression `MB_YESNOCANCEL + MB_DEFBUTTON3` is harder to write, but easier to understand.

- `0, MB_DEFBUTTON1` - First button is default value

- `256, MB_DEFBUTTON2` - Second button is default value
- `512, MB_DEFBUTTON3` - Third button is default value

Finally, the following information symbols are available and can also be displayed by adding the relevant parameter values:

- `16, MB_ICONSTOP` - Stop sign
- `32, MB_ICONQUESTION` - Question mark
- `48, MB_ICONEXCLAMATION` - Exclamation point
- `64, MB_ICONINFORMATION` - Tip icon

The following call displays an information box with the Yes and No buttons (value 4), of which the second button (No) is set as the default value (value 256) and which also receives a question mark (value 32), 4+256+32=292.

```
MsgBox "Do you want to continue?",  292
' or,
MsgBox "Do you want to continue?", MB_YESNO + MB_DEFBUTTON2 + MB_ICONQUESTION
```

If an information box contains several buttons, then a return value should be queried to determine which button has been pressed. The following return values are available in this instance:

- `1, IDOK` - Ok
- `2, IDCANCEL` - Cancel
- `3, IDABORT` - Abort
- `4, IDRETRY` - Retry
- `5` - Ignore
- `6, IDYES` - Yes
- `7, IDNO` - No

In the previous example, checking the return values could be as follows:

```
Dim iBox as Integer
iBox = MB_YESNO + MB_DEFBUTTON2 + MB_ICONQUESTION
If MsgBox ("Do you want to continue?", iBox) = IDYES Then
' or,
If MsgBox ("Do you want to continue?",  292) = 6 Then
  ' Yes button pressed
Else
  ' No button pressed
End IF
```

In addition to the information text and the parameter for arranging the information box, `MsgBox` also permits a third parameter, which defines the text for the box title:

```
MsgBox "Do you want to continue?",  292, "Box Title"
```

If no box title is specified, the default is “soffice”.

Input Box For Querying Simple Strings

The `InputBox` function queries simple strings from the user. It is therefore a simple alternative to configuring dialogs. `InputBox` receives three standard parameters:

- An information text.
- A box title.
- A default value which can be added within the input area.

```
InputVal = InputBox("Please enter value:", "Test", "default value")
```

As a return value, the `InputBox` provides the string typed by the user.

Other Functions

Beep

The `Beep` function causes the system to play a sound that can be used to warn the user of an incorrect action. `Beep` does not have any parameters:

```
Beep     ' creates an informative tone
```

Shell

External programs can be started using the Shell function.

```
Shell(Pathname, Windowstyle, Param)
```

`Pathname` defines the path of the program to be executed.

`Windowstyle` defines the window in which the program is started.

The following values are possible:

- `0` - The program receives the focus and is started in a concealed window.
- `1` - The program receives the focus and is started in a normal-sized window.
- `2` - The program receives the focus and is started in a minimized window.
- `3` - The program receives the focus and is started in a maximized window.
- `4` - The program is started in a normal-sized window, without receiving the focus.
- `6` - The program is started in a minimized window, the focus remains in the current window.
- `10` - The program is started in full screen mode.

The third parameter, `Param`, permits command line parameters to be transferred to the program to be started.

Wait and WaitUntil

The `Wait` statement suspends program execution for a specified time. The waiting period is specified in milliseconds. The command:

```
Wait 2000
```

specifies a delay of 2 seconds (2000 milliseconds).

The `WaitUntil` statement provides a greater degree of compatibility with VBA parameter usage. `WaitUntil` takes a parameter of type `Date`, with a combined date and time value. The command:

```
WaitUntil Now + TimeValue("00:00:02")
```

specifies the same delay, 2 seconds, as the previous example.

Environ

The `Environ` function returns the environmental variables of the operating system.

Depending on the system and configuration, various types of data are saved here. The following call determines the environment variables of temporary directory of the operating system:

```
Dim TempDir
TempDir=Environ ("TEMP")
```

◆ ◆ ◆ CHAPTER 4

Introduction to the API

OpenOffice.org objects and methods, such as paragraphs, spreadsheets, and fonts, are accessible to OpenOffice.org Basic through the OpenOffice.org application programming interface, or API. Through the API, for example, documents can be created, opened, modified and printed. The API can be used not only by OpenOffice.org Basic, but also by other programming languages, such as Java and C++. The interface between the API and various programming languages is provided by something called **Universal Network Objects** (UNO).

This chapter provides a background on the API. Building on this background, the following chapters will show how the API can be used to make OpenOffice.org do what you want it to do.

Universal Network Objects (UNO)

OpenOffice.org provides a programming interface in the form of the Universal Network Objects (UNO). This is an object-oriented programming interface which OpenOffice.org subdivides into various objects which for their part ensure program-controlled access to the Office package.

Since OpenOffice.org Basic is a procedural programming language, several linguistic constructs have had to be added to it which enable the use of UNO.

To use a Universal Network Object in OpenOffice.org Basic, you will need a variable declaration for the associated object. The declaration is made using the `Dim` instruction (see The Language of OpenOffice.org Basic). The `Object` type designation should be used to declare an object variable:

```
Dim Obj As Object
```

The call declares an object variable named `Obj`.

The object variable created must then be initialized so that it can be used. This can be done using the `createUnoService` function:

```
Obj = createUnoService("com.sun.star.frame.Desktop")
```

This call assigns to the `Obj` variable a reference to the newly created object. `com.sun.star.frame.Desktop` resembles an object type; however in UNO terminology it is called a service rather than a type. In accordance with UNO philosophy, an Obj is described as a reference to an object which supports the com.sun.star.frame.Desktop service. The service term used in OpenOffice.org Basic therefore corresponds to the type and class terms used in other programming languages.

There is, however, one main difference: a Universal Network Object may support several services at the same time. Some UNO services in turn support other services so that, through one object, you are provided with a whole range of services. For example, that the aforementioned object, which is based on the com.sun.star.frame.Desktop service, can also include other services for loading documents and for ending the program.

Note – Whereas the structure of an object in VBA is defined by the class to which it belongs, in OpenOffice.org Basic the structure is defined through the services which it supports. A VBA object is always assigned to precisely one single class. A OpenOffice.org Basic object can, however, support several services.

Properties and Methods

An object in OpenOffice.org Basic provides a range of properties and methods which can be