

One Design One Server One User Experience

**Programming with Actuate Basic** 

Information in this document is subject to change without notice. Examples provided are fictitious. No part of this document may be reproduced or transmitted in any form, or by any means, electronic or mechanical, for any purpose, in whole or in part, without the express written permission of Actuate Corporation.

© 1995 - 2011 by Actuate Corporation. All rights reserved. Printed in the United States of America.

Contains information proprietary to: Actuate Corporation, 2207 Bridgepointe Parkway, San Mateo, CA 94404

www.actuate.com www.birt-exchange.com

The software described in this manual is provided by Actuate Corporation under an Actuate License agreement. The software may be used only in accordance with the terms of the agreement. Actuate software products are protected by U.S. and International patents and patents pending. For a current list of patents, please see http://www.actuate.com/patents.

Actuate Corporation trademarks and registered trademarks include: Actuate, ActuateOne, the Actuate logo, Archived Data Analytics, BIRT, Collaborative Reporting Architecture, e.Analysis, e.Report, e.Reporting, e.Spreadsheet, Encyclopedia, Interactive Viewing, OnPerformance, Performancesoft, Performancesoft Track, Performancesoft Views, Report Encyclopedia, Reportlet, The people behind BIRT, X2BIRT, and XML reports.

Actuate products may contain third-party products or technologies. Third-party trademarks or registered trademarks of their respective owners, companies, or organizations include:

Adobe Systems Incorporated: Flash Player. Apache Software Foundation (www.apache.org): Axis, Axis2, Batik, Batik SVG library, Commons Command Line Interface (CLI), Commons Codec, Derby, Shindig, Struts, Tomcat, Xerces, Xerces2 Java Parser, and Xerces-C++ XML Parser. Bits Per Second, Ltd. and Graphics Server Technologies, L.P.: Graphics Server. Bruno Lowagie and Paulo Soares: iText, licensed under the Mozilla Public License (MPL). Castor (www.castor.org), ExoLab Project (www.exolab.org), and Intalio, Inc. (www.intalio.org): Castor. Codejock Software: Xtreme Toolkit Pro. DataDirect Technologies Corporation: DataDirect JDBC, DataDirect ODBC. Eclipse Foundation, Inc. (www.eclipse.org): Babel, Data Tools Platform (DTP) ODA, Eclipse SDK, Graphics Editor Framework (GEF), Eclipse Modeling Framework (EMF), and Eclipse Web Tools Platform (WTP), licensed under the Eclipse Public License (EPL). Jason Hsueth and Kenton Varda (code.google.com): Protocole Buffer. ImageMagick Studio LLC.: ImageMagick. InfoSoft Global (P) Ltd.: FusionCharts, FusionMaps, FusionWidgets, PowerCharts. Mark Adler and Jean-loup Gailly (www.zlib.net): zLib. Matt Ingenthron, Eric D. Lambert, and Dustin Sallings (code.google.com): Spymemcached, licensed under the MIT OSI License. International Components for Unicode (ICU): ICU library. KL Group, Inc.: XRT Graph, licensed under XRT for Motif Binary License Agreement. LEAD Technologies, Inc.: LEADTOOLS. Microsoft Corporation (Microsoft Developer Network): CompoundDocument Library. Mozilla: Mozilla XML Parser, licensed under the Mozilla Public License (MPL). MySQL Americas, Inc.: MySQL Connector. Netscape Communications Corporation, Inc.: Rhino, licensed under the Netscape Public License (NPL). Oracle Corporation: Berkeley DB. PostgreSQL Global Development Group: pgAdmin, PostgreSQL, PostgreSQL JDBC driver. Rogue Wave Software, Inc.: Rogue Wave Library SourcePro Core, tools.h++. Sam Stephenson (prototype.conio.net): prototype.js, licensed under the MIT license. Sencha Inc.: Ext JS. Sun Microsystems, Inc.: JAXB, JDK, Jstl. ThimbleWare, Inc.: JMemcached, licensed under the Apache Public License (APL). World Wide Web Consortium (W3C)(MIT, ERCIM, Keio): Flute, JTidy, Simple API for CSS. XFree86 Project, Inc.: (www.xfree86.org): xvfb. Yuri Kanivets (code.google.com): Android Wheel gadget, licensed under the Apache Public License (APL). ZXing authors (code.google.com): ZXing, licensed under the Apache Public License (APL).

All other brand or product names are trademarks or registered trademarks of their respective owners, companies, or organizations.

Document No. 110812-2-130331 July 13, 2011

## Contents

xi

#### Part 1 Working with Actuate Basic

Introducing Actuate Basic
About Actuate Basic
Programming with Actuate Basic
Understanding code elements
About statements 5
About expressions
About operators
Using an arithmetic operator7
Using a comparison operator
Using logical operators
Using the concatenation operator 10
Adhering to coding conventions
Commenting code
Breaking up a long statement
Adhering to naming rules
Using the code examples
Chapter 2
Understanding variables and data types 12
About variables
About variables
About variables       14         Declaring a variable       14         Using a global variable       15         Using a local variable       16
About variables       14         Declaring a variable       14         Using a global variable       15         Using a local variable       16         About slass variable       16
About variables       14         Declaring a variable       14         Using a global variable       15         Using a local variable       16         About class variables       16
About variables       14         Declaring a variable       14         Using a global variable       15         Using a local variable       16         About class variables       16         Declaring an array       16         16       16         17       16
About variables       14         Declaring a variable       14         Using a global variable       15         Using a local variable       16         About class variables       16         Declaring an array       16         About multidimensional arrays       17         About dumamic arrays       17
About variables       14         Declaring a variable       14         Using a global variable       15         Using a local variable       16         About class variables       16         Declaring an array       16         About multidimensional arrays       17         About functions used with an array       18
About variables14Declaring a variable14Using a global variable15Using a local variable16About class variables16Declaring an array16About multidimensional arrays17About dynamic arrays17About functions used with an array18About data types18
About variables       14         Declaring a variable       14         Using a global variable       14         Using a local variable       15         Using a local variable       16         About class variables       16         Declaring an array       16         About multidimensional arrays       17         About dynamic arrays       17         About functions used with an array       18         About data types       18
About variables       14         Declaring a variable       14         Using a global variable       15         Using a local variable       16         About class variables       16         Declaring an array       16         About multidimensional arrays       17         About functions used with an array       18         Using a standard data type       18         Using a standard data type       18
About variables14Declaring a variable14Using a global variable15Using a local variable16About class variables16Declaring an array16Declaring an array16About multidimensional arrays17About dynamic arrays17About functions used with an array18Using a standard data type18Using an Actuate Foundation Class data type20Assigning a data type20
About variables14Declaring a variable14Using a global variable15Using a local variable16About class variables16Declaring an array16About multidimensional arrays17About dynamic arrays17About functions used with an array18Using a standard data type18Using an Actuate Foundation Class data type20Assigning a data type20Using the As keyword21

Using a type-declaration character	21
About enums	22
About constants	23
Working with Variant data	23
About numeric Variant data	24
About functions used for a Variant variable	24
Working with String data	25
Declaring a String	25
Using binary string data	25
Manipulating a string	26
Formatting a string using Str or Str\$	27
Formatting a string using Format or Format\$	27
Comparing strings	28
Changing the capitalization of a string	28
Removing spaces from a string	29
Embedding special characters in a string	29
Embedding quotation marks in strings	29
Embedding tabs and new-line characters in strings	30
Working with numeric data	30
About numerical data types	30
About the Currency data type	31
Converting a string to a number	31
Working with date and time data	31
Using date and time display formats	32
Formatting date and time values	32
Working with a user-defined data type	33
Using an alias	33
Using a structure	33
Using a class	34
Converting a data type	34
Chapter 3	
Writing and using a procedure	37
About procedures	38
About scope in procedures	38
About methods	

	Thour scope in procedures	00
	About methods	38
	About global procedures	39
	Declaring a Sub procedure	39
	Declaring a Function procedure	40
	Creating a global procedure	40
D	eclaring an argument	41
	About argument data types	41
	Passing an argument by reference	41

Passing an argument by value	42
Calling a procedure	42
Calling a Sub procedure	42
Calling a Function procedure	42
Overloading a procedure	42
Using a control structure	43
Using a nested control structure	44
Exiting a control structure	44
Exiting a Sub or Function procedure	44
Chapter 4	
Calling an external function	45
Understanding external C functions	46
Using a C function with Actuate Basic	46
Unloading an external library	46
Declaring a C function	47
Declaring the C function as a Sub procedure	47
Declaring the C function as a Function procedure	47
Understanding C function declaration issues	48
Specifying the library of a C function	48
Passing an argument by value or reference	48
About flexible argument types	49
Aliasing a non-standard C function name	49
Determining an Actuate Basic argument type	49
Calling a C function	51
Calling a C function with a specific data type	51
Passing a string to a C function	51
Passing an array to a C function	51
Passing a null pointer to a C function	52
Passing a user-defined data type to a C function	52
Passing an object reference variable to a C function	52
About return values from C functions	52
Working with a Java object	53
About Java requirements	53
Creating a Java object	53
Invoking a method and accessing a field on a Java object	53
Invoking a static method and accessing a static field	54
Converting a Java data type	54
Converting a Java String	55
Converting a Java null	55
Converting an array	55
About Java exception and error handling	56
Debugging a Java object	57

#### Part 2 Actuate Basic Language Reference

Chapter 5	
Language summary	61
Arrays	
Classes and instances	
Program flow	62
Conversion	63
Date and time	63
Environment	64
Error trapping	65
File input and output	
Finances	
Graphics and printing	67
Math	
Operators	
Procedures	
Strings	
Variables and constants	
Chapter 6	
Statements and functions	71
Using the code examples	
Abs function	
Acos function	
AddBurstReportPrivileges function	
AddValueIndex function	
Asc function	
AscW function	
Asin function	
Assort statement	<b>Q1</b>

Ascvv function	80
Asin function	81
Assert statement	81
Atn function	82
Beep statement	83
Call statement	84
CCur function	85
CDate function	87
CDbl function	90
ChDir statement	92
ChDrive statement	93
Chr, Chr\$ functions	95
ChrW, ChrW\$ functions	97

CInt function	
Class statement	100
ClearClipboard function	102
CLng function	
Close statement	
Command, Command\$ functions	
Const statement	
ConvertBFileToString function	
ConvertStringToBFile function	
ConvertToXML function	
CopyInstance statement	
Cos function	
CreateJavaClassHandle function	
CreateJavaObject function	
CSng function	
CStr function	
CurDir, CurDir\$ functions	
CVar function	
CVDate function	
Date, Date\$ functions	
DateAdd function	
DateDiff function	
DatePart function	132
DateSerial function	
DateValue function	
Day function	
DDB function	
Declare statement	
DeclareEnd Declare statement	
Dim statement	
DoLoop statement	
End statement	154
EnumEnd Enum statement	
Environ, Environ\$ functions	
EOF function	
Erase statement	
Erl function	
Err function	
Err statement	
Error, Error\$ functions	166
Error statement	
Exit statement	168
Exp function	

ExtendSearchPath function	.171
FileAttr function	.172
FileCopy statement	.174
FileDateTime function	.175
FileExists function	.177
FileLen function	.178
FileTimeStamp function	.179
FindFile function	.181
Fix function	.182
Format, Format\$ functions	.184
ForNext statement	.197
FreeFile function	.199
FunctionEnd Function statement	.200
FV function	.205
Get statement	.208
GetAFCROXVersion function	. 211
GetAppContext function	.212
GetAttr function	.213
GetClassID function	.216
GetClassName function	.217
GetClipboardText function	.219
GetDisplayHeight function	.220
GetFactoryVersion function	.222
GetFontAverageCharWidth function	.223
GetFontDisplayHeight function	.224
GetHeadline function	.224
GetJavaException function	.225
GetLocaleAttribute function	.226
GetLocaleName function	.227
GetObjectIdString function	.227
GetOSUserName function	.228
GetPId function	.228
GetReportContext function	.229
GetReportScalingFactor function	.230
GetROXVersion function	.230
GetSearchFormats function	.231
GetServerName function	.232
GetServerUserName function	.233
GetTextWidth function	.233
GetUserAgentString function	.234
GetValue function	.234
GetValueType function	.236
GetVariableCount function	.238

GetVariableName function	240
GetViewPageFormats function	241
GetVolumeName function	242
Global statement	243
GoTo statement	246
Hex, Hex\$ functions	248
Hour function	250
IfThenElse statement	253
IIf function	255
Input statement	256
Input, Input\$ functions	259
InputB, InputB\$ functions	261
InStr function	261
InStrB function	263
Int function	264
IPmt function	266
IRR function	268
IsDate function	270
IsEmpty function	272
IsKindOf function	273
IsNull function	274
IsNumeric function	275
IsPersistent function	276
IsSearchFormatSupported function	277
IsViewPageFormatSupported function	278
Kill statement	279
LBound function	280
LCase, LCase\$ functions	282
Left, Left\$ functions	283
LeftB, LeftB\$ functions	284
Len function	285
LenB function	286
Let statement	287
Line Input statement	289
ListToArray function	290
Loc function	291
LockUnlock statement	293
LOF function	298
Log function	299
LSet statement	300
LTrim, LTrim\$ functions	301
Mid, Mid\$ functions	302
Mid, Mid\$ statements	304

MidB, MidB\$ functions	.305
MidB, MidB\$ statements	.307
Minute function	.308
MIRR function	.311
MkDir statement	.313
Month function	.315
MsgBox function	.318
MsgBox statement	.321
Name statement	.323
NewInstance function	.325
NewPersistentInstance function	.326
Now function	.327
NPer function	.328
NPV function	.330
Oct, Oct\$ functions	.332
On Error statement	.333
Open statement	.335
Option Base statement	.340
Option Compare statement	.342
Option Strict statement	.343
ParseDate function	.344
ParseNumeric function	.348
Pmt function	.349
PPmt function	.351
PreciseTimer function	.354
Print statement	.354
Put statement	.357
PV function	.361
QBColor function	.363
Randomize statement	.364
Rate function	.366
ReDim statement	.368
Rem statement	.371
Reset statement	.372
Resume statement	.373
RevInStr function	.374
RGB function	.376
Right, Right\$ functions	.379
RightB, RightB\$ functions	.380
RmDir statement	.381
Rnd function	.383
RSet statement	.384
RTrim, RTrim\$ functions	.385

SafeDivide function	387
Second function	387
Seek statement	390
Seek2 function	392
Select Case statement	394
Set statement	397
SetAttr statement	398
SetBinding function	400
SetClipboardText function	401
SetDefaultPOSMFile function	402
SetHeadline statement	402
SetStructuredFileExpiration function	403
SetValue function	405
Sgn function	406
Shell function	407
ShowFactoryStatus statement	409
Sin function	409
Sleep statement	410
SLN function	.411
Space, Space\$ functions	412
Sqr function	413
Static statement	414
Stop statement	416
Str, Str\$ functions	417
StrComp function	418
String, String\$ functions	420
StringW, StringW\$ functions	421
StrSubst function	422
SubEnd Sub statement	423
SVGAttr function	427
SVGColorAttr function	428
SVGDbl function	429
SVGFontStyle function	430
SVGStr function	432
SVGStyle function	434
SYD function	436
Tab function	438
Tan function	440
Time, Time\$ functions	441
Timer function	442
TimeSerial function	442
TimeValue function	445
Trim, Trim\$ functions	447

TypeEnd Type statement
TypeAs statement
UBound function
UCase, UCase\$ functions
Val function
VarType function
Weekday function
WhileWend statement
Width statement
Write statement
Year function

#### Appendix A

Operators
* operator
+ operator
- operator
/ operator
\ operator
^ operator
& operator
And operator
BAnd operator
BNot operator
BOr operator
Comparison operators
Eqv operator
Imp operator
Is operator
Like operator
Mod operator
Not operator
Or operator
XOr operator
Appendix B
Kevwords
Appendix C
Trigonometric identities 501
Index

#### About Programming with Actuate Basic

*Programming with Actuate Basic* provides information for using the functions and statements in the Actuate Basic programming language.

Programming with Actuate Basic includes the following chapters:

- About Programming with Actuate Basic. This chapter provides an overview of this guide.
- Part 1. Working with Actuate Basic. This part describes and provides information about the elements of Actuate Basic, working with variables, procedures, and external functions in Actuate Basic.
- *Chapter 1. Introducing Actuate Basic.* This chapter describes the elements of Actuate Basic and its syntax and coding conventions.
- *Chapter 2. Understanding variables and data types.* This chapter provides information about working with variables in Actuate Basic.
- *Chapter 3. Writing and using a procedure.* This chapter provides information about working with procedures in Actuate Basic.
- *Chapter 4. Calling an external function.* This chapter describes how to call a C function in an Actuate Basic report.
- Part 2. Actuate Basic Language Reference. This part provides lists of functions and statements and information about keywords, operators, and trigonometric identities.
- Chapter 5. Language summary. This chapter provides grouped lists of functions and statements that are frequently used together to perform specific programming tasks.
- *Chapter 6. Statements and functions.* This chapter contains an alphabetical reference for all functions and statements.

- *Appendix A. Operators.* This appendix contains information about Actuate Basic operators.
- *Appendix B. Keywords.* This appendix contains information about Actuate Basic reserved words.
- *Appendix C. Trigonometric identities.* This appendix contains information about trigonometric identities.

# Part One

Working with Actuate Basic

# 1

### **Introducing Actuate Basic**

This chapter contains the following topics:

- About Actuate Basic
- Programming with Actuate Basic
- Understanding code elements
- Adhering to coding conventions
- Using the code examples

For an alphabetical list of all Actuate Basic functions and statements, including a general description, syntax, and parameters and return values, see Chapter 6, "Statements and functions."

#### **About Actuate Basic**

Actuate Basic is an object-oriented programming language you can use for creating sophisticated e.reports and distributing them over the web. Actuate Basic consists of standard Actuate Basic functions and statements and uses object-oriented language extensions. Developers use Actuate Basic to augment and extend the functionality of e.Report Designer Professional.

Actuate Basic is similar to other structured programming languages. It provides built-in functions and control structures, such as If...Then...Else constructs and For and While loops. It also supports using variables and creating your own methods and procedures. Actuate Basic is modeled after Microsoft Visual Basic, Version 3. It is designed for programming reports and has none of Visual Basic's form-related syntax.

This chapter describes the elements of Actuate Basic and its syntax and coding conventions. Table 1-1 lists key features of Actuate Basic.

Feature	Functionality	For information, see
Strong data typing	Detecting mismatched type errors when the report compiles instead of when it runs.	Chapter 2, "Understanding variables and data types"
User-defined data types	Combining variables of different types into a single type.	Chapter 2, "Understanding variables and data types"
Calls to external functions	Using a function stored in an external library. For example, you can create and access a Java object, call a C function, and convert an Actuate Basic string to a Java String.	Chapter 4, "Calling an external function"
Portability	Running a report on a Windows or UNIX server.	Chapter 6, "Statements and functions"

 Table 1-1
 Actuate Basic features

#### **Programming with Actuate Basic**

Actuate e.Report Designer Professional provides design tools that support dynamic content delivery. Using e.Report Designer Professional, you create a report by placing components in a report design and setting properties to customize their appearance. e.Report Designer Professional translates the report design into corresponding Actuate Basic code.

You can also write code in Actuate Basic to control the report generation process, add application-specific logic, or design a report that responds to user-triggered events. To accomplish these tasks, you typically override predefined methods in the Actuate Foundation Classes (AFCs) or create new methods. For information about AFC classes and methods, see *Programming with Actuate Foundation Classes*.

Programming in Actuate Basic involves writing procedures. The procedures you write can be methods or general procedures available to the entire report. You declare a method in a class. You store a general procedure in an Actuate Basic source (.bas) file. When you include a BAS in your report design, you can use the general procedures in it the same way you use Actuate Basic's built-in statements and functions. For more information about procedures, see Chapter 3, "Writing and using a procedure."

#### Understanding code elements

To program in Actuate Basic, you must familiarize yourself with its syntax, naming conventions, control structures, and so on. This section provides an overview of the following standard Actuate Basic code elements:

- About statements
- About expressions
- About operators

#### About statements

A statement is a complete set of instructions directing Actuate Basic to execute a specific task within a procedure. A procedure typically contains a series of Actuate Basic statements that perform an operation or calculate a value. Figure 1-1 shows an example of statements within a procedure.

The simplest and most common statement in a procedure is the assignment statement. It consists of a variable name followed by the equal sign and an expression:

```
<variable> = <expression>
```

```
Sub SetLabelBackgroundColor( anyControl As
AcLabelControl)
Dim Label As MyLabel
If GetClassName(anyControl) = "MyLabel" Then
Set Label = anyControl
Label.BackgroundColor = Red
End If
End Sub
```

#### Figure 1-1 Procedure

The expression can be as simple or complex as necessary but it must evaluate to a valid data type. For more information about data types, see Chapter 2, "Understanding variables and data types."

For example, the following assignment statements are valid:

```
Area = PI * Radius ^ 2
StartTime = StopTime - Now
Net_Worth = Assets - Liabilities
Label.Text = "Page " + Str$(curPageNum) + " of " & NumPages
```

In these examples, the assignment statement stores information. The examples compute the value of an expression, which appears to the right of the assignment operator (=). They store the result in the variable, which appears to the left of the operator. The data type of the variable must be the same as the data type the computed expression returns. For example, assigning a double value to an integer variable rounds the double to the nearest integer.

If necessary, you can use a function to convert elements of different data types to one common type. The last statement in the previous example uses the Str\$ function to convert a numeric expression to a string.

Actuate Basic statements can be recursive, meaning that they can call themselves repeatedly to perform a task. Actuate Basic sets a run-time stack limit of 200. A report that exceeds this limit causes a fatal error in e.Report Designer Professional and e.Report Designer. A report using recursion with a large number of iterations can exceed this limit.

#### **About expressions**

An expression consists of values and operators combined in such a way that the expression evaluates to one of the Actuate Basic data types. The type of the operands in an expression determines the expression type.

In an expression that contains an operation, Actuate Basic considers the types of all operands to determine the type of the result. For example, if you add two integers, the result of the expression is an integer. If you mix data types in an expression, the result is the type of the widest range. For example, the expression 3 \* 4.55 produces a double.

To avoid mixing types in an expression, use type-declaration characters with numeric constants. These characters force a constant to take on a specific type. For example, 10 is an integer, but 10@ is a currency value. Thus, 10@ + 20@ yields a currency value of thirty dollars on the U.S. English locale. For more information about type-declaration characters, see "Using a type-declaration character" in Chapter 2, "Understanding variables and data types."

In an expression that contains a variable that evaluates to Null, the result of the expression is Null. For example, if the value of the variable NextOrderQuantity is Null, the expression 2 \* NextOrderQuantity produces a value of Null. Only the concatenation operator treats Null values differently, as described in "Using the concatenation operator." Actuate Basic functions also return Null when you provide an argument that evaluates to Null. For example, if the value of VacationStartDate is Null, the function call, DateAdd( "d", 5, VacationStartDate ) returns Null.

#### **About operators**

An operator is a symbol or keyword that performs an operation, such as adding or comparing, on an expression. Actuate Basic provides the following types of operators:

- Arithmetic
- Comparison
- Logical
- Concatenation

For more information about operators Actuate Basic supports, see Appendix A, "Operators."

#### Using an arithmetic operator

Use an arithmetic operator with a numeric expression. You can also use an arithmetic operator with a date expression to subtract one date from another and to add or subtract a number from a date.

Table 1-2 lists the arithmetic operators that Actuate Basic supports.

Operator	Description	Syntax
٨	Exponentiation. Supports computing powers and roots. Actuate Basic computes the	number ^ exponent

 Table 1-2
 Supported arithmetic operators

(continues)

Operator	Description	Syntax
∧ (continued)	power using a positive exponent and computes a root using a negative exponent. For example, 10 ^ 3 evaluates to 1000.	
*	Multiplication	expression1 * expression2
/	Floating point division. Performs standard division and returns a floating point number. For example, 3/2 returns 1.5 as a floating point, even though both operands are integers.	expression1 / expression2
\	Integer division. Returns only the integer portion of the division. For example, $10\2$ evaluates to 5 and $3\2$ evaluates to 1.	expression1 $\ expression2$
Mod	Modulus (remainder)	expression1 Mod expression2
BAnd	Bitwise And	expression1 BAnd expression2
BOr	Bitwise Or	expression1 BOr expression2
+	Addition	expression1 + expression2
-	Subtraction	expression1 - expression2

**Table 1-2** Supported arithmetic operators (continued)

#### Using a comparison operator

Use a comparison operator to compare two expressions of the same type and return True or False. Table 1-3 lists the comparison operators that Actuate Basic supports.

**Table 1-3**Supported comparison operators

Operator	Description	Syntax
>	Greater than	expression1 > expression2
<	Less than	expression1 < expression2
>=	Greater than or equal to	expression1>= expression2
<=	Less than or equal to	expression1<= expression2
=	Equal to	expression1 = expression2
	Although the equal to operator is the same as the assignment	

Operator	Description	Syntax
= (continued)	operator, the context of the operator determines which one Actuate Basic uses. The assignment operator is valid only when it immediately follows a variable in an assignment statement. For all other contexts, the = operator is a comparison operator. The expressions are	expression1 = expression2
	case-sensitive.	
<>	Not equal to	expression1 <> expression2
Like	String comparison Expression and pattern are case-sensitive.	expression Like pattern
Is	Object reference variable comparison	objectref1 Is objectRef2

Table 1-3Supported comparison operators

#### **Using logical operators**

Use logical operators to compare two logical expressions and return True or False. Table 1-4 lists the logical operators Actuate Basic supports.

Table 1-4	Supported logical operators	
Operator	Description	Syntax
Not	Performs a logical negation on an expression. Returns False if the expression is True.	Not expression
And	Performs a logical conjunction on two expressions. Returns True if both expressions are True. Returns False if either expression is False.	expression1 And expression2
Or	Performs a logical disjunction on two expressions. Returns True if one or both expressions is True.	expression1 Or expression2
Xor	Performs a logical exclusion on two expressions. Returns True if only one expression is True.	expression1 Xor expression2
		(continues)

		,
Operator	Description	Syntax
Eqv	Performs a logical equivalence on two expressions. Returns a value of True if both expressions evaluate to the same logical value.	expression1 Eqv expression2
Imp	Performs a logical implication on two expressions. Returns a value of True if both expressions are False or the second condition is True.	expression1 Imp expression2

#### Table 1-4 Supported logical operators (continued)

#### Using the concatenation operator

The concatenation operator (&) combines two values to produce a string that contains both original values. The original values do not change. Actuate Basic casts each value to a string, then concatenates the strings. If one of the values is Null, the concatenation operator produces a string that is the same as the other value. If both values are Null, the concatenation operator produces Null.

The following examples produce the same result. The difference between the examples is that using a value other than a string produces a variant value, which Actuate Basic casts to a string.

```
6 & 7
"6" & "7"
"6" & Str$(7)
6 & 7 & Null
```

#### Adhering to coding conventions

Adhering to coding conventions makes your code easy to read and understand. For example, you can comment your code to explain its purpose, use a consistent style to name language elements, and break up long statements at logical places. This section describes the following common coding conventions:

- Commenting code
- Breaking up a long statement
- Adhering to naming rules

#### **Commenting code**

Code comments make a procedure easy to understand and maintain, especially if other programmers work with your code. Actuate Basic recognizes two comment markers:

- The apostrophe (')
- Rem

Actuate Basic treats characters to the right of a ' character or Rem as comments and does not execute those lines. A comment can follow a statement or occupy an entire line. A comment cannot follow a line continuation character (+). The following example shows the different types of comment:

```
Rem This is a comment that takes up an entire line.

'This is a comment that takes up an entire line.

Text1.BackGroundColor = Red ' Print negative numbers in red.
```

#### Breaking up a long statement

Typically, you write code with one Actuate Basic statement to a line and no terminator at the end of the line. Sometimes, however, you must break a long statement over several lines. To do so, use a continuation character (+) at the beginning of any lines after the first one, as shown in the following example:

```
Datal.RecordSource = "SELECT * FROM Titles, Publishers "
+ & "WHERE Publishers.PubId = Titles.PubID AND
+ Publishers.State = 'CA'"
```

The continuation character must be the continued line's first character.

#### Adhering to naming rules

When naming an element such as a variable, constant, or procedure, adhere to the following rules:

- The name can contain up to 40 characters.
- The first character of the variable name must be a letter. A letter is any upperor lowercase character in the US ASCII (A-Z and a-z) character set and does not include non-English characters such as å or ô.
- The name cannot start with a number, including double-byte numeric characters.
- Subsequent characters in a name can be letters, digits, or the underscore character (\_).
- The name cannot duplicate an existing variable name in the same scope.

- The name cannot contain reserved words, such as Function, Type, Sub, If, and End. For a list of reserved words, see Appendix B, "Keywords."
- The name cannot contain an operator such as \*, ^, or %.
- The name cannot contain spaces.

#### Using the code examples

Actuate products include example code to help you program specific functionality into a report. For more information about using example code, see "Using the code examples" in Chapter 6, "Statements and functions."

#### Chapter

# 2

### Understanding variables and data types

This chapter contains the following topics:

- About variables
- Declaring an array
- About data types
- Working with Variant data
- Working with String data
- Working with numeric data
- Working with date and time data
- Working with a user-defined data type
- Converting a data type

#### About variables

A variable is a named location in memory used for temporary data storage. When you write Actuate Basic code, you often store values. For example, you can compare values, perform a calculation and store the result, or store values such as global configuration settings. Actuate Basic uses variables to store these values.

#### **Declaring a variable**

You declare a variable using the Dim, Global, or Static statements, as shown in the following examples:

Dim Total As Integer Global FileName As String Static Counter As Integer

Actuate Basic does not support implicit declaration of variables. When you implicitly declare a variable, you can use the variable name in your code without declaring it first. Because good programming practice discourages implicit variable declarations, Actuate Basic requires that you explicitly declare a variable before you use it.

The scope of a variable determines which procedures can access it. Depending on how and where you declare it, a variable has one of three scopes described in Table 2-1.

Scope	Accessed by	Declared using	Declared in
Global	Entire application	Global	Declarations section of an Actuate Basic source (.bas) file included in a report design
Local	The procedure or method in which you declare it	Dim or Static	A procedure or method
Class	The class in which you declare it	Dim or Static	A class

Table 2-1	Scopes of variables
-----------	---------------------

When you add a variable to a class, Actuate Basic generates an appropriate Dim or Static declaration. For more information about the Global, Dim, and Static statements, see Chapter 6, "Statements and functions." For information about class variables, see *Programming with Actuate Foundation Classes*.

#### Using a global variable

A global variable is accessible to the entire application. Use a global variable for storing values that apply to an entire application. For example, to store file names and default paths for accessing data.

The following example shows an example of an Actuate Basic source (.bas) file that has global variables defined in a declaration section:

```
Declare
Global XVar As Integer
Global YVar As String
End Declare
```

Although global variables are useful in certain situations, minimize their use. Global variables can add complexity to the logic of a report and can contribute to the creation of complex state machines. For this reason, use a local variable when possible.

#### How to create a global variable using a new source file

To create a global variable using a new source file:

- 1 In e.Report Designer Professional, create the Actuate Basic source (.bas) file:
  - 1 Choose Tools→Library Organizer→New.
  - 2 In New Library, name the file and select Source File (\*.bas). Choose Save.
  - 3 In Library Organizer, choose OK.

A page for creating the source file appears.

- **2** In the source file, create a declarations section using Declare...End Declare.
- **3** In the body of the declarations section, declare the global variable.
- **4** Save the source file.

The report includes the source file as a library.

#### How to initialize a global variable

To initialize a global variable after you declare it:

- **1** Write a Sub procedure in your source file that assigns an initial value to the global variable.
- **2** Override the Start method of the report so that it calls the Sub procedure.

When report generation begins, Actuate initializes the global variable.

#### Using a local variable

A local variable is accessible only within the procedure or method in which you declare it. A local variable is useful for a temporary calculation, such as counting in a For or Do loop. Limiting the scope of a variable supports reusing variable names. For example, if you declare a variable called Total in a procedure and you later write another procedure for a similar task, you can use the variable name Total in the new procedure.

Declare a local variable using Dim or Static, as shown in the following example:

```
Dim intTemp As Integer
Static intPermanent As Integer
```

A variable you declare using Dim exists only as long as the procedure or method is executing. When a procedure finishes executing, Actuate discards the values of local variables and reclaims the memory. The next time the procedure executes, the local variables reinitialize.

A variable you declare using Static retains the same data throughout the report generation process. Use Static to retain the value of a local variable to perform tasks such as calculating a cumulative total. In the following example, a function calculates a cumulative total by adding a new value to the total of previous values stored in the static variable Accumulate:

```
Function CumulativeTotal(ByVal Num As Double) As Double
Static Accumulate As Double
Accumulate = Accumulate + Num
CumulativeTotal = Accumulate
End Function
```

If you declare Accumulate using Dim instead of Static, the function does not retain previously accumulated values. The function returns the same value with which you call it.

#### About class variables

A class variable typically stores values that define the state and attributes of an object of the class. You can declare a class variable using Dim or Static.

#### **Declaring an array**

An array is a list of objects of the same size and data type. Each object in an array is called an array element. Array elements are contiguous. You use an index number to differentiate elements. For example, you can declare an array of integers or an array of doubles, as shown in the following example:

```
Dim Counters(14) As Integer ' Integers, indexed 0-14 or 1-14,

' depending on the Option Base statement

Dim Sums(50) As Double ' Doubles, indexed 0-50 or 1-50

Dim Sums(10 To 20) As Double ' Doubles, indexed 10-20
```

You can also create a Variant array and populate it with elements of different data types, such as integers and strings. For more information about working with data types, see "About data types," later in this chapter.

Using an array, you can set up loops that efficiently manage a number of cases by using the index number. An array has an upper and lower bounds, such as 1-50 or 10-100. Because Actuate Basic allocates space for each index number, avoid declaring an array larger than necessary.

#### About multidimensional arrays

You can declare an array with multiple dimensions. For example, a twodimensional array has rows and columns. The following statement declares an array called Matrix. Matrix is a fixed-size, two-dimensional, 10 x 20 array with bounds.

Dim Matrix(1 To 10, 1 To 20) As Double

Be aware when adding dimensions to an array that the total storage the array requires increases dramatically, especially for a Variant array.

#### About dynamic arrays

Sometimes, you do not know in advance how large to make an array. Actuate Basic supports changing the size of an array as the report runs. Such an array is called a dynamic array. A dynamic array helps you efficiently manage memory. For example, you can use a large array for a short time and reallocate memory to the system when the report completes. As an alternative, you can declare a fixedsize array of the largest possible size, which can cause your process to run low on memory.

To declare a dynamic array, use a statement similar to the following declaration:

Dim AccordionArray() As Double

Place nothing between the parentheses, then size the array later using the ReDim statement. To change the size of an array without losing the data in it, use ReDim with the Preserve keyword. For example, you can enlarge an array by five elements without losing the values of the existing elements.

ReDim Preserve MyArray(UBound(MyArray) + 5)

You can also use ReDim to decrease the size of an array.

#### About functions used with an array

Table 2-2 lists the Actuate Basic functions you can use for working with arrays. For more information about these functions, see Chapter 6, "Statements and functions."

Programming task	Function/Statement
Change default lower limit	Option Base
Declare and initialize	Dim, Global, ReDim, Static
Test the limits	LBound, UBound
Reinitialize	Erase, ReDim

Table 2-2Functions for arrays

#### About data types

Actuate products use two categories of data types, those provided by Actuate Basic and those that are defined specifically for use with Actuate Foundation Classes (AFC). This chapter discusses the Visual Basic data types. For information about AFC-specific data types, see *Programming with Actuate Foundation Classes*.

The data type determines the type of data the variable can store. A data type provides a way to classify data stored in a variable by defining rules that govern how you work with data and what the variable can contain. The Actuate Basic data types include the standard Visual Basic data types and data types defined for the Actuate Foundation Classes.

#### Using a standard data type

Table 2-3 describes the standard Visual Basic data types that Actuate Basic uses.

Data type	Storage size	Range/Description
Any	N/A	Used only to suppress type checking of arguments passed to dynamic-link library (DLL) procedures.
CPointer	4 bytes 32 bits	0 to 4,294,966,295.
		A CPointer data type variable holds a pointer to data allocated in a C function you create.
Currency	12 bytes 96 bits	-39,614,081,257,132,168,796.771975168 to 39,614,081,257,132,168,796.771975167

 Table 2-3
 Standard Visual Basic data types that Actuate Basic uses

Data type	Storage size	Range/Description		
Date	8 bytes 64 bits	Dates 1 January 100 to 31 December 9999		
		Times 0:00:00 to 23:59:59		
Double	8 bytes 64 bits	-1.797693134862315E308 to -2.23E-308 for negative values. 2.23E-308 to 1.797693134862315E308 for positive values.		
		Double-precision floating point numbers are stored in memory using IEEE 64-bit floating point format. Each floating point value consists of three parts: the sign, the exponent, and the mantissa. In a double- precision number, the sign takes 1 bit, the exponent takes 11 bits, and the mantissa uses the remaining 52 bits plus an additional implied bit.		
Integer	4 bytes 32 bits	-2,147,483,648 to 2,147,483,647		
		Can represent Boolean values. For Boolean values, 1 represents True and 0 represents False. You can also use the Actuate Basic reserved words True and False, which return 1 and 0, respectively.		
		Can represent enumerated values. An enumerated value can contain a finite set of unique whole numbers, each of which has special meaning in the context in which it is used. Enumerated values provide a convenient way to select among a known number of choices.		
Long	4 bytes 32 bits	-2,147,483,648 to 2,147,483,647		
Single	8 bytes 64 bits	±2.2250738585072014E-308 to ±1.7976931348623158E+308		
String	1 byte per character	0 to approximately 2,147,483,647 characters or memory limit. Because some storage overhead is required, a string cannot actually be 2,147,283,647 characters long.		
		For more information about the String data type, see "Working with String data," later in this chapter.		
User-defined (defined with Type)	Defined by elements	Range of each element depends on its fundamental type.		

 Table 2-3
 Standard Visual Basic data types that Actuate Basic uses (continued)

(continues)

Data type	Storage size	Range/Description
Variant	As needed	Depends on value stored, up to the range of a Double.
		The default data type for Actuate Basic.
		For more information about the Variant data type, see "Working with Variant data," later in this chapter.

 Table 2-3
 Standard Visual Basic data types that Actuate Basic uses (continued)

#### Using an Actuate Foundation Class data type

An Actuate Foundation Class (AFC) data type is either a typedef or a structure.

A typedef is an alias for a Basic data type, typically an Integer. Actuate Foundation Classes use a typedef to create a type that adds rules to those a simple data type provides. These rules govern limitations on range, direct special display formatting, and specify related constants. For example, typedef rules about the numbers that represent colors restrict them to the range of 0 to hexadecimal FFFFFF.

Some data types are enumerations, or enums. An enum is a data type in which the value can be only one of its associated constants. For example, a Boolean can be only True or False. Other data types also can have constants. In such cases, the constant provides convenient shorthand for a value.

A structure is a group of variables that describe a single item. The structure member can be an Actuate Basic data type, such as an Integer, Boolean, or other AFC-defined structure. In some cases, structures are nested, as in the following example:

```
AcFont
AcColor
```

For a complete list of the AFC data types, see *Programming with Actuate Foundation Classes.* 

#### Assigning a data type

A variable has an associated data type. When declaring a variable, you can explicitly assign a data type to it. If you do not assign a data type, Actuate Basic assigns the Variant data type. Use the Option Strict statement to require that all variable declarations have an assigned data type. For more information about Option Strict, see Chapter 6, "Statements and functions." You can assign a data type to a variable in one of two ways:

- Using the As keyword
- Using a type-declaration character

#### Using the As keyword

To declare a variable and assign it a data type using the As keyword, use the following syntax:

Dim <variable> [As <type>]

The following examples show valid variable declarations:

Dim MyAccount As Currency Dim YourAccount

These examples declare both variables and initialize them to the default value of the data type. For these examples, MyAccount is a Currency variable initialized to 0.00. YourAccount is a Variant initialized to Empty.

#### Using a type-declaration character

You can assign a data type to a variable by appending a type-declaration character to the end of the variable name.

Table 2-4 lists the type-declaration characters Actuate Basic supports.

Data type	Type declaration character	Type declaration character		
Integer	% (ANSI character 37)	% (ANSI character 37)		
Long	& (ANSI character 38)			
Single (real)	! (ANSI character 33)			
Double (real)	# (ANSI character 35)			
Currency	@ (ANSI character 64)			
String	\$ (ANSI character 36)			
Variant	None			

 Table 2-4
 Supported type-declaration characters

The following examples show valid variable declarations:

Dim	OurAccount@	ı.	declare	Currency variable
Dim	CustomerName\$	ı.	declare	String variable
Dim	QuantitySold%	1	declare	Integer variable

#### About enums

An enum is a data type whose value is one of a set of named values. For example, the value of a TrafficLightColor enum might be RedLight, YellowLight, or GreenLight.

You define an enum using the Enum statement. To declare an enum use the following syntax:

```
Enum <user-efined enumeration type>
    <name 1> [= value 1]
    <name 2> [= value 2]
    ...
    <name N> [= value N]
End Enum
```

The following example declares an enum. You place this statement within a Declare...End Declare block.

```
Enum CarMotion
Forward
Reverse
Stopped
End Enum
```

To use the enum type, declare a local variable as that type. You can then assign that variable the values specified in the enum declaration.

```
Dim myCarState As CarMotion
myCarState = Stopped
```

An enum is very similar to an Integer and a set of global constants. In the above example, myCarState behaves like an Integer, and Forward, Reverse, and Stopped behave like global constants.

The Enum statement syntax also allows an enum value to be assigned an explicit numeric equivalent, as shown in this example:

```
Enum CarMotion
Reverse = -1
Stopped = 0
Forward = 1
End Enum
```

If the constant assignment is missing from an enum value, that value is assigned the previous constant value plus one. If the first enum value assignment is missing, it is set to zero.

In practice, you should never need to know the numeric equivalents of enum values. Instead, you should always use the enum value names.
#### **About constants**

A constant is a reserved location in memory for storing a value that does not change during the life of a report.

The following examples set constant values:

Const Pi = 3.14159265358979 Const FirstName = "James" Const MyAccount = 25.43

These examples do not contain type declarations. In such cases, Actuate Basic assigns the most appropriate data type. In the example, Pi is a Double, FirstName is a String, and MyAccount can be a Single, a Double, or a Currency. In such a case, Actuate Basic assigns the data type that uses the least amount of space, in this case a Single. To specify that MyAccount is Currency, use the following syntax:

```
Const MyAccount@ = 25.43@
```

#### Working with Variant data

A Variant variable can change its type depending on the program logic. The Variant is the default data type for Actuate Basic. This means that it is the data type that all variables become if you do not explicitly declare them to be of another type.

A Variant can contain numeric, string, or date data, as well as the special values empty and Null. The Variant maintains an internal representation of the data type it stores. This internal representation determines how Actuate Basic manages the value when you perform an operation on the variable. When you work with a Variant, Actuate Basic typically performs any necessary conversions. Actuate Basic uses the most compact representation possible to represent the variable. Later operations on the variable can result in data type changes. You can determine how Actuate stores the data in a Variant using the VarType function. VarType returns a value that indicates which data type a Variant contains. For more information about VarType, see Chapter 6, "Statements and functions."

A Variant variable contains the value Empty until you assign another value to it. A Variant containing Empty is 0 if it is used in a numeric context and a zerolength string if it is used in a string context. Empty is not the same as a Null value. Null indicates that the Variant variable intentionally contains no valid data.

The Variant is a convenient way to declare a variable for a small report or frequently used variable. However, a Variant reduces the performance of the program and can cause unintended data type conversions. For example, the result of the + operator can be ambiguous when you use it with two Variants. If both Variants are numbers, Actuate Basic adds them. If both Variants are strings, Actuate Basic concatenates them. If one Variant is a string and the other is a

number, Actuate Basic first attempts to convert the string to a number. If the conversion fails, a Type mismatch error occurs. To avoid such issues, use fewer Variants and more explicitly typed variables in your code.

Always use a Variant when the data could contain date information, Empty, or a Null. You can also use a Variant in place of any fundamental data type to work with data in a more flexible way. If the contents of a Variant variable are digits, they might either be the string representation of the digits or the actual value, depending on the context. For example:

```
Dim MyVar As Variant
MyVar = 98052
```

In the preceding example, MyVar contains a numeric representation of the actual value 98052. Arithmetic operators work as expected on Variant variables that contain numeric values or string data that can be interpreted as numbers. If you use the + operator to add MyVar to another Variant containing a number or to a variable of a numeric data type, the result is an arithmetic sum.

#### About numeric Variant data

Generally, Actuate Basic maintains numeric Variant data in its original fundamental data type within the Variant. For example, if you assign an Integer to a Variant, subsequent operations treat the Variant as if it were an Integer. However, if an arithmetic operation is performed on a Variant containing an Integer, a Long, or a Single, and the result exceeds the normal range for the original data type, the result is promoted within the Variant to the next larger data type. An overflow error occurs when Variant variables containing Currency and Double values exceed their respective ranges.

#### About functions used for a Variant variable

Table 2-5 lists the Actuate Basic functions you can use for working with Variant variables. For more information about these functions, see Chapter 6, "Statements and functions."

Programming task	Function/Statement
Determine whether an argument can be converted to a date	IsDate
Test whether a Variant variable contains a value	IsEmpty
Determine whether the Variant contains the Null value	IsNull
Test if the type of a variable is or can be converted to Integer, Long, Single, Double, or Currency	IsNumeric
Determine the data type of a Variant	VarType

 Table 2-5
 Functions for working with Variant variables

#### Working with String data

A String variable contains text. Although a string can contain digits, you cannot perform mathematical operations on the digits in a string. Within a string, Actuate Basic interprets digits as characters. For example, you can store zip codes and telephone numbers as string data in tables because you do not perform calculations with them.

String characters that all Windows applications use consist of one- or two-byte codes. These codes for String characters depend on the character encoding used by a locale or the encoding explicitly set for a report design. The first 128 characters (0-127) correspond to the letters and symbols on a standard U.S. keyboard. These first 128 characters are the same as defined by the ASCII character set. The second 128 characters (128-255) represent other characters, such as letters in international alphabets, accents, currency symbols, and fractions, and differ depending on the locale. Internally, Actuate products use the two-byte Unicode UCS-2 encoding, which ensures a consistent interpretation of literal strings. Nonetheless, depiction of a character in a report depends on the availability of the code point in the chosen font and the current encoding.

#### **Declaring a String**

Declare a variable as a String data type if it always contains text and you never expect to treat it as a number in a calculation.

Dim myString As String

You can assign a String to this variable and modify the variable using String functions. Use quotation marks to delimit a literal string in an expression.

```
myString = "Actuate Basic"
newString = Left$(myString,7)
```

A String variable or argument is a variable-length string. The string grows or shrinks as you assign new data to it.

#### Using binary string data

A Visual Basic program typically stores binary data in a String variable. This technique works because American National Standards Institute (ANSI) strings are typically single-byte arrays of characters. Actuate Basic code can read and write binary data on a byte-by-byte basis, using strings and manipulating the standard String functions, regardless of whether the data has a meaningful ANSI equivalent.

#### Manipulating a string

You can manipulate a string, including a string containing binary data, using Actuate Basic String functions. There are three types of String functions, two for character strings and another for binary strings. You can use the character versions without any preparatory steps. You use the AscW and ChrW character String functions for two-byte Unicode characters. The other character String functions interpret character values as one or two bytes according to the current encoding.

Table 2-6 lists the functions you can use for manipulating character strings. For more information about these functions, see Chapter 6, "Statements and functions."

Functions	Description
Asc, AscW	Returns the character code for the first character.
Chr, ChrW	Returns a string containing the specified character.
InStr	Returns the first occurrence of one string within another.
Left, Right	Returns a specified number of characters from the right or left sides of a string.
Mid	Returns a specified number of characters from a string.

**Table 2-6** Functions for manipulating character strings

#### How to use a binary String function

To use a binary String function:

- 1 Place the binary data in a byte array using Get or another function.
- **2** Assign the byte array to a string.

This assignment copies the data into a string variable. It does not translate the binary data.

**3** Use one of the functions listed in Table 2-7 to manipulate the binary data in the string.

Function	Description
InStrB	Returns the first occurrence of a byte in a binary string.
MidB	Returns a specified number of bytes from a binary string.
LeftB, RightB	Returns a specified number of bytes from the left or right side of a binary string.

**4** Assign the contents of the manipulated binary string back to a byte array.

For more information about the functions you use for manipulating binary data, see Chapter 6, "Statements and functions."

For information about working with a multibyte character string, see *Working in Multiple Locales using Actuate Basic Technology.* 

#### Formatting a string using Str or Str\$

The simplest way to format a string is to use the Str or Str\$ functions. The syntax is:

```
Str(<expression>)
Str$(<expression>)
```

Str returns a Variant data type. Str\$ returns a String data type. Str and Str\$ do not perform a conversion if you pass them a String value.

When you convert a number using Str or Str\$, the function places either a leading space or a minus sign in front of the string. If the number is positive, the leading space implies the plus sign. If the number is negative, the minus sign is placed in front of the string.

The converted string follows the formatting rules of the user's locale. The following statement, which passes a numeric value, returns 123,456 with a leading space in a French locale, and 123.456 with a leading space in an English locale:

Str\$(123.456) ' Passing a numeric value

The following statement, which passes a String value, returns 123,456 with a leading space in both French and English locales:

Str("123,456") ' Passing a String value

#### Formatting a string using Format or Format\$

The Format and Format\$ functions format a numeric expression, date Variant, or String according to the specified pattern and locale.

Format returns a Variant data type. Format\$ returns a String. The syntax is:

```
Format(<expression to format>)
Format(<expression to format>, <format pattern>)
Format$(<expression to format>, <format pattern>, <locale>)
```

where

- <expression to format> is the expression to display according to the specified format pattern.
- <format pattern> is a predefined Actuate Basic keyword or a string of format characters that specifies how the expression appears.

 <locale> is a String expression that specifies the locale to use for determining <format pattern> and the output format.

For more information about the Format and Format\$ functions, see Chapter 6, "Statements and functions."

#### **Comparing strings**

The StrComp function returns a Variant value indicating the result of a string comparison. The syntax for StrComp is:

```
StrComp(<string expression 1>, <string expression 2>[,<compare
  method>])
```

where

- <string expression 1> and <string expression 2> are any valid string expressions. Actuate Basic converts these strings to the Variant data type before comparing them.
- <compare method> specifies the type of string comparison. You can omit this argument or use a value of 0 or 1. Use 0, the default value, to perform a binary, or case-sensitive, comparison. Use 1 to perform a textual, or case-insensitive, comparison. If compare method is Null, an error occurs. If you omit this argument, the Option Compare setting determines the type of comparison.

Table 2-8 shows the return values of a string comparison.

If	StrComp returns	
string1 is less than string2	-1	
string1 is equal to string2	0	
string1 is greater than string2	1	
string1 or string2 is Null	Null	

 Table 2-8
 Return values of a string comparison

For an example of using StrComp, see Chapter 6, "Statements and functions."

#### Changing the capitalization of a string

Table 2-9 lists the functions you can use to change the capitalization of a string. For more information about these functions, see Chapter 6, "Statements and functions."

Function	Description
LCase	Returns a Variant in which all letters are lowercase
LCase\$	Returns a String in which all letters are lowercase
UCase	Returns a Variant in which all letters are uppercase
UCase\$	Returns a String in which all letters are uppercase

**Table 2-9** Functions for changing the capitalization of a string

#### Removing spaces from a string

Table 2-10 lists the functions you can use to remove leading or trailing spaces from a string. For more information about these functions, see Chapter 6, "Statements and functions."

 Table 2-10
 Functions for removing spaces from a string

Function	Description
LTrim	Returns a copy of a String without leading spaces
RTrim	Returns a copy of a String without trailing spaces
Trim	Returns a copy of a String with neither leading nor trailing spaces

#### Embedding special characters in a string

Actuate Basic supports embedding quotation marks, tabs and new-line characters in a string.

#### Embedding quotation marks in strings

Actuate Basic supports two techniques of embedding a literal quotation within a string, as shown in the following example:

The CEO said, "Sales are fantastic!"

To make this sentence into a valid string expression using the first technique, insert an additional set of quotation marks to display each set in the string. Actuate Basic interprets two quotation marks in a row as an embedded quotation mark. To assign the previous string to a variable, use the following code:

Text.Aside = "The CEO said, ""Sales are fantastic!"""

In the second technique, Actuate Basic supports back quotes (`) to delimit string constants. The following code creates a string with the same value as the previous example:

```
Text.Aside = 'The CEO said, "Sales are fantastic!"'
```

#### Embedding tabs and new-line characters in strings

Tab and new-line characters in string constants can also be embedded into strings. Enclose string constants that contain these characters in braces ({ and }). This syntax supports pasting large blocks of text such as HTML, XML, or SQL directly from an external application into method code.

The following example creates a string with both tabs and new-line characters:

```
ObtainSelectStatement = {"
   SELECT
    Customers.custID,
   Customers.name
   FROM
    customers
   "}
```

#### Working with numeric data

Actuate Basic offers several data types that can contain numeric data. The type you choose depends on how you use the numbers. The following sections describe the different types of numeric data and when to use each.

#### About numerical data types

The Integer and Long data types represent numbers that have no fractional component.

The Single and Double data types can express floating point numbers.

You can express a floating point value as:

mmmmEeee

where

- mmmm is the mantissa
- eee is the exponent, a power of 10

You can assign all numeric variables to each other and to variables of the Variant type. Actuate Basic rounds the fractional part of a floating point number before assigning the floating point number to an Integer.

#### About the Currency data type

The Currency data type represents a monetary value with a precision of four decimal places. Use the Currency data type for calculations involving money and for fixed point calculations in which accuracy is important. A floating point number has a larger range than a Currency data type. A floating point number is subject to small rounding errors. Because Currency is a scalar data type, it is not subject to rounding errors within its range.

A Currency variable is stored as 96-bit (12-byte) number in an Integer format, scaled by 1,000,000,000 (10<sup>9</sup>). This scale produces a fixed point number with 20 digits to the left of the decimal point and 9 digits to the right.

To ensure that a Currency value maintains its precision, append the currency type declaration symbol, @, to the value after you define it, as shown in the following example:

Dim DollarAmt As Currency DollarAmt = 922337203685476.5807@

The preceding example ensures that DollarAmt is not subject to rounding errors. In this example, if you do not specify the Currency type declaration symbol, DollarAmt evaluates to 922337203685477.

#### Converting a string to a number

Use the Val function to convert a String to a numeric value. Val returns a Double data type. The syntax for Val is:

```
Val("<string expression>")
```

For information about how Val converts strings, see Chapter 6, "Statements and functions."

#### Working with date and time data

Actuate Basic stores a Date variable as an IEEE 64-bit floating point number. This number represents dates ranging from 1 January 100 to 31 December 9999 and times from 0:00:00 to 23:59:59. You can assign any recognizable literal date value to a Date variable. You must enclose a literal date within number sign characters (#) in the following format:

#1/1/2002# or #11/12/2003#

The user's locale determines the interpretation of a literal Date value. If you plan to distribute a report to multiple locales, DateSerial is a better choice.

#### Using date and time display formats

A Date variable uses the short date format your computer recognizes. The time appears in either 12- or 24-hour format, according to your computer's time format.

When you convert other numeric data types to Date, values to the left of the decimal represent date information and values to the right of the decimal represent time. Midnight is 0 and midday is .5. A negative number represents a date before December 30, 1899.

#### Formatting date and time values

You can perform mathematical operations on a date or time value. Adding or subtracting an integer operates on the date. Adding or subtracting a fraction operates on the time. Table 2-11 lists the Actuate Basic date functions. For more information about these functions, see Chapter 6, "Statements and functions."

Function	Description
DateSerial	Returns a date or date serial number from the year, month, and day numbers entered. Always specify a four-digit year. The supported date range for report scheduling is January 1, 1980 through December 31, 2036. The supported date range for all other data processing and display, including database access, is January 1, 100 through December 31, 9999.
Day	Extracts the day component from a date and returns a number.
Weekday	Determines the day of the week for this date and returns it as a number (1 to 7), where Sunday is 1.
Month	Extracts the month component of a date and converts it to a number.
Date	Returns the current date.
Year	Extracts the year component from a date and returns a number.

 Table 2-11
 Actuate Basic date functions

You can use date and time literals by enclosing them within number signs (#). For example:

If Today > #2/25/2003# Then ...

You can also include a time.

If Now > #2/25/2003 10:00pm# Then ...

If you do not include a time, Actuate Basic uses midnight as the start of the day. If you do not include a date, Actuate Basic uses December 30, 1899 as the start of the calendar.

To determine whether a Variant or other value can convert to a Date data type, use the IsDate function. You can then use the CDate() function to convert the value to a Date data type.

For information about working with numeric data in different locales, see *Working in Multiple Locales using Actuate Basic Technology*.

#### Working with a user-defined data type

You can use Actuate Basic's data types for most data in a report that uses strings, numeric data, and dates. Many reports require logically structuring related data. These reports are less prone to error when you create custom, or user-defined, data types. Actuate Basic supports three user-defined data types:

- Alias
- Structure
- Class

#### Using an alias

An alias is the simplest type of user-defined data. When you use an alias, you declare a data type that has the properties of an existing data type. The syntax is:

Typedef <new data type> As <existing data type>

Typically, the new data type is one of the Actuate Basic data types. The following example shows an AFC type:

Typedef AcColor As Integer

You can also declare an alias to a structure, as shown in the following example:

Typedef MyPoint As AcPoint

#### Using a structure

A user-defined data type is a structure in which the elements contain previously defined data types. These data types can be either Actuate Basic data types or user-defined data types.

For example, if you write a program to catalog computers, you can define a structure that represents related information about the computer systems, as shown in the following example:

```
Type SysInfo
ComputerName As String
CPU As String
Memory As Long
DiskDriveType As String
DiskDriveSize As Single
Price As Currency
PurchaseDate As Date
End Type
```

After you declare a structure, you can declare a variable of this data type in the same way that you declare a variable of a fundamental type.

Dim SystemOne As SysInfo, SystemTwo As SysInfo

When you declare a variable, you can assign and retrieve values from the elements of the variable using the dot notation, as shown in the following example:

```
SystemOne.Price = 1999.95
SystemOne.PurchaseDate = #2/25/03#
```

If both variables are of the same user-defined type, you can also assign one userdefined variable to another. The following example assigns all elements of the first variable to the same elements of the second variable:

```
SystemTwo = SystemOne
```

You can nest user-defined types. After you define a data type, you can include it in another user-defined type. Nesting can be as complex as your application requires. To keep your code readable and easy to debug, however, keep all nested user-defined types in one module.

#### Using a class

A class is a user-defined data type that defines the attributes of an object in a report. To create an object, you declare a variable that contains a reference to the object and assign a class as its type. For more information about classes and objects, see *Programming with Actuate Foundation Classes*.

#### Converting a data type

Actuate Basic provides functions to convert a value into a specific data type. For example, to convert a value to Currency, use the CCur function:

```
WeeklyPay = CCur(hours * rate)
```

Table 2-12 lists the functions you can use for converting a data type. For more information about these functions, see Chapter 6, "Statements and functions."

Function	Converts an expression to
CCur	Currency
CDate	Date
CDbl	Double
CInt	Integer
CLng	Long
CSng	Single
CStr	String
CVar	Variant
CVDate	Variant of VarType 7 (Date)

 Table 2-12
 Functions for converting a data type

For each locale, these functions convert separators in a numeric expression according to the locale's rules. The Decimal, Grouping, CurrencyDecimal and CurrencyGrouping parameters of localemap.xml define locale-specific rules.

For more information about locale-specific data, see *Working in Multiple Locales* using Actuate Basic Technology.

#### Chapter

# 3

## Writing and using a procedure

This chapter contains the following topics:

- About procedures
- Declaring an argument
- Calling a procedure
- Overloading a procedure
- Using a control structure

#### About procedures

A procedure is a set of commands that perform a specific set of operations. Actuate Basic supports two types of procedures, Sub and Function. A Sub procedure does not return a value. A Function procedure returns a value.

A procedure can simplify programming by breaking a program into smaller logical components. You use these components as building blocks that enhance and extend Actuate Basic.

A procedure is useful for condensing a repeated or shared task, such as a calculation, text and control formatting, and database operations.

Using procedures has two major benefits:

- When you break a program into discrete logical units, you can debug these units more easily than an entire program without procedures.
- You can reuse a procedure in multiple reports, typically without modification.

#### About scope in procedures

The procedures you create can have either class or global scope. The scope of the procedure determines which parts of a program can access that procedure. A procedure you declare in a class has class scope. This means that it is accessible only to objects of that class. A procedure with class scope is a method of the class.

A procedure you declare in an Actuate Basic source (.bas) file has global scope. This means that it is accessible from any part of the report. The following sections describe methods and global procedures in more detail.

#### About methods

A method is a procedure you declare within a class declaration. A report typically includes multiple objects of a class. A method performs an action on an object in a class. The following example shows a method that checks the values of a parameter:

```
Sub Start()
If IsNull(StartDate)
+ Or (DateDiff("d", StartDate, DateSerial(2006,1,1) > 0)) Then
StartDate = DateSerial(2006, 1, 1)
End If
Super::Start()
End Sub
```

For more information about methods, see *Programming with Actuate Foundation Classes*.

Usually, your code does not need to specify which object is currently executing. Sometimes, however, a method or property refers explicitly to a particular object. The Me keyword supports referring to the object in which the code is running. Use Me as if it were the Name property of the object, as shown in the following example:

```
Sub PrintTotal( )
   Me.Print
End Sub
```

#### About global procedures

You can create a global procedure by creating a new Actuate Basic source (.bas) file or by using an existing BAS.

The following example shows an outline of a BAS that uses global procedure declarations:

```
Sub ProcA( )
...
End Sub
Function FuncAr( ) As Integer
...
FuncA = <expression>
End Function
```

After you include a module containing procedure declarations, you call your procedures the same way as you call Actuate's predefined functions.

#### **Declaring a Sub procedure**

The syntax for a Sub procedure is:

```
Sub <procedure name>( [<arguments>] )
        <statements>
End Sub
```

where

- <arguments> is an optional list of argument names separated by commas. Each argument looks like a variable declaration and acts like a variable in the procedure. For more information about arguments, see "Declaring an argument," later in this chapter.
- <statements> are executed each time the procedure is called. You can place a Sub procedure in a global module, where it is accessible throughout the report, or in a class, where it becomes a method that operates on or is accessible only to an instance of that class.

#### **Declaring a Function procedure**

Actuate Basic includes built-in functions that return a value with an appropriate data type. For example, Now returns a date value containing the current date and time.

You can write your own procedures using the Function statement. The following example shows the syntax:

```
Function <procedure name>( <arguments> ) [As <type>]
        <statements>
        <procedure name> = <expression>
End Function
```

A Function procedure is similar to a Sub procedure in that it can take arguments, execute a series of statements, and change the value of its arguments. The arguments for a Function procedure work the same way as arguments for a Sub procedure, with the following differences:

- Typically, you call a function by including the Function procedure name and arguments on the right side of a larger statement or expression. The variable on the left of the equal sign stores the value the Function returns.
- A Function procedure has a data type, just as a variable does, that determines the type of the return value. If you omit the As clause, the function type is Variant.
- You return a value by assigning it to the procedure name within the procedure. When the function returns, this value can become part of the larger expression from which you called the function.

The following example shows how to write a function that calculates and returns the area of a rectangle, given the length and width:

```
Function RectArea (Width As Single, Length As Single) As Single
RectArea = Width * Length
End Function
```

#### Creating a global procedure

To create a global procedure using a new source file, perform the following steps.

- 1 Create an Actuate Basic source (.bas) file.
  - 1 Choose Tools→Library Organizer→New.
  - **2** In New Library, type a name for the file. In Save as type, choose Source File (\*.bas).
  - 3 Navigate to the directory in which to save the file. Choose Save.
  - 4 In Library Organizer, choose OK.

A window for creating the source file appears.

- **2** In the source file, write the procedure using the Sub or Function statement. Include parentheses after the procedure name, even if the parentheses are empty.
- **3** Save the source file.

The report includes the source file.

#### **Declaring an argument**

An argument passes a value to the procedure. You declare an argument as a variable in a procedure declaration. For example, a procedure that performs a calculation usually requires a value for processing, as shown in the following examples. You pass this value to the procedure when you call it. The arguments are the names the procedure uses for the values you supply. The first value you supply gets the first parameter name in the list, the second value gets the second parameter name, and so on.

```
Function Sum2 (Num1 As Integer, Num2 As Integer) As Integer
Sum = Num1 + Num2
End Function
Function ExtendedCost (Cost As Currency, Quantity As Integer)
+ As Currency
ExtendedCost = Cost * Quantity
End Function
```

#### About argument data types

The default data type of a procedure argument is Variant. You can declare another data type for a procedure argument variable. For example, the following function accepts a string and an integer:

```
Function ExtractLeftString( S As String, N As Integer)
    <statements>
End Function
```

#### Passing an argument by reference

Actuate Basic passes a variable to a procedure by reference. Passing a variable by reference gives the procedure access to the variable's location in memory. Using this information, the procedure can change the value of the variable.

If you declare a data type for an argument passed by reference, you must pass a value of that data type when you call the procedure. You can convert the data type if necessary.

#### Passing an argument by value

The ByVal keyword passes a variable by value. When calling an internal procedure, Actuate Basic passes a copy of the value to the procedure. In this case, if the procedure changes the value, only the copy changes, not the original. When the procedure finishes, the copy of the variable goes out of scope.

When calling an external procedure, Actuate Basic passes the address of the string data. In this case, if the procedure changes the value, the original value also changes.

#### Calling a procedure

The techniques for calling a procedure depend on the type of procedure, its scope, and the way your report uses it. The following sections describe how to call Sub and Function procedures.

#### Calling a Sub procedure

A call to a Sub procedure is a stand-alone statement. Because a Sub procedure does not return a value, you cannot call it by using its name within an expression. A Sub procedure can modify values of any variables it receives as arguments.

You call a Sub procedure in one of two ways, as shown in the following examples:

```
Call MyProc (Argument1, Argument2)
MyProc (Argument1, Argument2)
```

#### **Calling a Function procedure**

Because a Function procedure returns a value, you typically call it by using its name as part of a larger expression. You call a custom function the same way you call a built-in function. All the following examples call a function named UpdateTotal:

```
Print UpdateTotal
NewTotal = UpdateTotal()
If UpdateTotal < 0 Then
   ShowFactoryStatus( "Error: Total is negative.")
End If</pre>
```

#### **Overloading a procedure**

To overload a procedure is to define multiple versions of it. The versions use the same procedure name but different argument lists. The lists can differ in the

number of arguments, the order of arguments, or the data types of the arguments. Overloading produces a group of closely-related procedures that can call one another as necessary.

For example, you can write a square root function that operates on integers and another square root function for doubles. In Actuate Basic, you can give both procedures the same name, square\_root. By overloading square\_root, you give it more than one possible meaning.

Actuate Basic distinguishes between versions of the function by the type and number of arguments. The following examples show an overloaded procedure:

Function square\_root( intInput As Integer) As Integer Function square\_root( doubleInput As Double) As Double

#### Using a control structure

Use a control structure to control the flow of report generation. Most reports contain decision points that support using structures and loops to change statement order. Table 3-1 describes the control structures that Actuate Basic supports.

Statement	Description	Syntax
DoLoop	Executes a block of statements while a specified condition is True or until it becomes True. You can use several variations of the DoLoop statement but each evaluates a numeric conditional as a Boolean to determine whether to continue executing. The first syntax in this table tests the condition statement. If False, the loop code is skipped. If true, the statements execute and the loop repeats until the condition evaluates to False. The second syntax guarantees that the loop will execute at least once.	Do While <condition> <statements> Loop Do <statements> Loop While <condition></condition></statements></statements></condition>
		(continues)

 Table 3-1
 Supported control structures

Statement	Description	Syntax
IfThen Else	Executes one block of specified statements based on a	If <condition1> Then [<statement 1="" block="">]</statement></condition1>
	conditional expression. You can include only a single Else clause regardless of whether you have ElseIf clauses.	ElseIf <condition2> Then [<statement 2="" block="">] Else [<statement 3="" block="">] End If</statement></statement></condition2>
ForNext	Repeats a block of instructions a specified number of times.	For <counter> = <start> To <end> [Step <step size="">] <statements> Next <counter></counter></statements></step></end></start></counter>

#### Table 3-1 Supported control structures (continued)

#### Using a nested control structure

You can place a control structure inside another control structure, such as a For...Next loop within an If...Then block. A control structure inside another control structure is called a nested structure.

Actuate Basic supports nesting a control structure to as many levels as you need.

#### Exiting a control structure

To exit a loop without performing any further iterations or statements within the loop, use the Exit For and Exit Do statements.

#### Exiting a Sub or Function procedure

To exit a procedure without processing any remaining statements in it, use the Exit Sub and Exit Function statements. These statements terminate the Sub or Function procedure and transfer control to the line following the procedure's End statement.

#### Chapter

# 4

### Calling an external function

This chapter contains the following topics:

- Understanding external C functions
- Declaring a C function
- Calling a C function
- Working with a Java object
- Converting a Java data type
- About Java exception and error handling
- Debugging a Java object

#### **Understanding external C functions**

This chapter describes how to call a C function in an Actuate Basic report. It does not describe how to program a C function or publish it in a library. Refer to C documentation for that information.

Actuate Basic provides a rich set of statements and functions you can use in your report. Sometimes, however, your program can accomplish more or execute more efficiently using C functions. Many companies and third-party developers supply libraries of C functions. By taking advantage of these C libraries, you can extend the functionality of Actuate Basic.

Actuate Basic supports calling an external C function if the function is accessible to a report. In the Windows environment, an Actuate Basic report can call a C function stored in a dynamic link library (DLL). On a UNIX system, a report can call a C function stored in a shared library.

#### Using a C function with Actuate Basic

Because C functions reside in external files, you must provide information to Actuate Basic so that it can find and execute the appropriate C function. Using a C function involves two basic steps:

- Declare the C function in Actuate Basic using the Declare statement. This step supplies the information Actuate Basic requires to convert and execute the C function.
- Call the C function the same way you call an Actuate statement or function. Actuate software loads the library then executes the C function.

You declare the C function only once but you can call it any number of times.

To make a C function accessible to all parts of your Actuate application, write the Declare statements in an Actuate Basic source (.bas) file, using a text or code editor. Then, include the BAS in your report by choosing Tools→Library Organizer.

#### Unloading an external library

As part of a library's memory management, the library must de-allocate memory used by a variable that an external call returns. For a variable you use with Actuate, you can create a memory cleanup function, AcCleanup(char \*aPointer). If you implement an AcCleanup function in your external library, Actuate calls AcCleanup after it finishes using the variable. If you do not implement AcCleanup, Actuate does not call AcCleanup.

#### **Declaring a C function**

The first step in using an external C function is to declare it. You must provide the following information in the Declare statement:

- The name of the C function
- The name of the DLL or shared library that contains the C function
- Arguments that the C function takes
- A return data type value, if any

You can also provide a name with which to call the C function by using the Alias keyword. For more information about using Alias, see "Aliasing a non-standard C function name," later in this chapter.

For more information about the Declare statement, see Chapter 6, "Statements and functions."

#### Declaring the C function as a Sub procedure

If the C function does not return a value, declare it as a Sub procedure, using the following syntax:

```
Declare Sub <function name> Lib <"library name">
    [Alias <"alias name">] [(<argument list>)]
```

The following example declares a Sub procedure:

```
Declare Sub LogError Lib "Kernel32"(ByVal uErr As Integer,
+ lpvInfo As Any)
```

#### Declaring the C function as a Function procedure

If the C function returns a value, declare it as a Function procedure, using the following syntax:

```
Declare Function < function name> Lib <"library name">
    [Alias <"alias name">] [(<argument list>)] [As <type>]
```

The following example declares a Function procedure:

```
Declare Function GetSystemMetrics Lib "USER32" (i As Integer)
+ As Integer
```

#### **Understanding C function declaration issues**

To declare and call a C function, ensure you have documentation for those procedures. A C function declaration can become complex for the following reasons:

- C often passes arguments by value, whereas Actuate Basic's default behavior is to pass arguments by reference.
- C type declarations differ from Actuate Basic type declarations.
- A C function can have a name that is not a valid identifier in Actuate Basic.

The following sections explain the syntax of the Declare statement in more detail so that you can create the correct declaration for the C function.

#### Specifying the library of a C function

On a Windows system, you specify the name of a DLL in the Lib<"library name"> clause. On a UNIX system, you specify the name of a shared library.

<"library name"> can be a file specification that includes a path. For example:

```
Declare Function GetSystemMetrics
+ Lib "C:\WINNT\SYSTEM32\USER32" (i As Integer) As Integer
```

If you omit the path, Actuate searches for the library in the following order:

- The directory in which Actuate is installed
- The current directory
- The Windows system directory
- The Windows directory
- On a Windows system, the directories listed in the PATH environment variable
- On a UNIX system, the directories listed in the LD\_LIBRARY\_PATH, LIBPATH, or SHLIB\_PATH environment variables

If you omit the file extension, Actuate assumes a .dll extension.

#### Passing an argument by value or reference

By default, Actuate Basic passes all arguments by reference. However, many C functions expect you to pass arguments by value. If you pass an argument by reference to a procedure that expects a value, the procedure cannot interpret the data.

To pass an argument by value, use the ByVal keyword before the argument declaration in the Declare statement. Each time you call the procedure, Actuate Basic passes the argument by value, as shown in the following example:

```
Declare Function GetFreeSystemResources Lib "User32" + (ByVal fuSysResource As Integer) As Integer
```

The arguments you pass by reference to C functions are strings and arrays.

#### About flexible argument types

Some C functions can accept more than one type of data for the same argument. To pass more than one type of data, declare the argument using the As Any keyword to remove type restrictions, as shown in the following example:

Declare Function SendMessage Lib "User32"

+ (ByVal hWnd As Integer, ByVal msg As Integer,

+ ByVal wp As Integer, ByVal lp As Any) As Long

When you call the function, you can pass either a string or a long integer as its last argument.

```
FindItem = SendMessage(aList.hWnd, LB_FINDSTRING, -1, target)
```

#### Aliasing a non-standard C function name

Occasionally, a C function has a name that is an invalid Actuate Basic identifier. A C function name is invalid if it starts with an underscore, contains a hyphen, or has the same name as an Actuate Basic reserved word. When the C function name is an invalid Basic identifier, use the Alias keyword, as shown in the following example:

```
Declare Function LOpen Lib "Kernel32" Alias "_lopen"
+ (ByVal fn As String, ByVal f As Integer) As Integer
```

The preceding example calls LOpen in the Basic code. \_lopen is the name of the C function in the DLL or shared library.

You can also use Alias when you want to use a name different than the actual C function name. For example, to substitute a shorter name for a long C function name.

#### **Determining an Actuate Basic argument type**

To call a C function from Actuate Basic, you must translate it into a valid Declare statement. Table 4-1 lists the Actuate Basic argument types to declare in the Declare statement, based on the C argument types of the functions you call.

Table 4-1Relationships of C argument types to Actuate Basic<br/>typesC typeActuate Basic type

int \*x, int &x

x As Integer

(continues)

C type	Actuate Basic type
int x	ByVal x As Integer
long *x, long &x	x As Long
long x	ByVal x As Long
double *x, double &x	x As Double, x As Single, x As Date
double x	ByVal As Single, ByVal As Double, ByVal As Date
AcCurrency *x <sup>1</sup>	x As Currency
AcCurrency x	ByVal x As Currency
char *x, LPCSTR x (null terminated string)	ByVal x As String
wchar_t *x, LPCTSTR x (null terminated Unicode string)	ByVal x() As String
union {    int, long, double, AcCurrency, RWCString } *x	x As Variant, x As Any
actual data (int, long, double, AcCurrency, char*)	ByVal x As Variant, ByVal x As Any
void **x (pointer to any C/C++ pointer)	x As CPointer
void *x	ByVal x As CPointer
int *x, int x[]	x() As Integer, ByVal x() As Integer
long *x, long x[ ]	x() As Long, ByVal x() As Long
double *x, double x[ ]	x() As Single, ByVal x() As Single, x() As Double, ByVal x() As Double, x() As Date, ByVal x() As Date
AcCurrency *x, AcCurrency x[]	x() As Currency, ByVal x() As Currency
RWCString **x, RWCString &x[]	x() As String
char **x, char *x[ ]	ByVal x() As String
void **x, void *x[ ]	x() As CPointer, ByVal x() As CPointer
wchar_t **x, wchar_t *x[ ]	ByVal x() As String

 Table 4-1
 Relationships of C argument types to Actuate Basic types (continued)

1. AcCurrency is a C structure with a size of 12 bytes.

```
struct AcCurrency
{
    unsigned long low;
    unsigned long mid;
    long high;
};
The currency value is stored in fixed point format.
    control of the point format.
```

```
<96-bit integer value in AcCurrency structure> = <currency
value> * 10<sup>9</sup>
```

#### Calling a C function

Call a C function as you do an Actuate Basic statement or function, ensuring that you pass the correct arguments. Actuate Basic cannot verify the arguments you pass.

#### Calling a C function with a specific data type

Actuate Basic provides a range of data types, including some that C functions do not support, such as variable-length strings, AnyClass, and Object. The following sections discuss data-type issues to consider when passing an argument in a function call.

#### Passing a string to a C function

Procedures in most DLLs expect standard C strings, which end in a null character (binary zero). If a C function expects a null-terminated string as an argument, declare the argument as a string with the ByVal keyword. In a string declaration, the ByVal keyword is misleading. ByVal tells Actuate Basic to convert the string to a null-terminated string, not that the string is passed by value. In fact, strings are always passed to C functions by reference.

#### Passing an array to a C function

You pass individual elements of an array in the same way you pass any variable of the same type as the base type of the array. You can also pass an entire array of any type except user-defined types. Actuate copies data in the Actuate Basic array into a C array before passing it to the C function.

If you pass a multidimensional array, the C function reverses the dimensions. For example, if you declare an Actuate Basic array using the following declaration:

Dim x (3, 4, 5) As Integer

C converts it to:

Int x [5][4][3]

Passing an array by value is faster than passing it by reference. Use the ByVal keyword to pass an array by value when possible.

#### Passing a null pointer to a C function

In Actuate Basic, any of the fundamental data types can be null. Therefore, to pass a null pointer to a C function, use the Null keyword, as shown in the following examples:

```
Declare Sub Func1 Lib "LIBABC" (ByVal p As CPointer)
Dim ptr As CPointer
Func1(ptr) ' pass CPointer ptr to the C function
Func1(Null) ' pass null pointer to the C function
Declare Sub Func2 Lib "LIBXYZ" (ByVal s As String)
Dim s As String
s = "text"
Func2(s) ' pass null-termintated string to the C function
Func2(Null) ' pass null pointer to the C function
```

#### Passing a user-defined data type to a C function

You cannot pass a user-defined data type to a C function.

#### Passing an object reference variable to a C function

You cannot pass an object reference variable to a C function. An object reference variable is a complex data structure that a C function cannot interpret.

#### About return values from C functions

Table 4-2 shows how C return values convert to Actuate Basic data types.

C type	Actuate Basic type
int	Integer
long	Long
double	Single, Double, Date
char* (null terminated)	String
(char*) NULL	String (empty)
any pointer (void*, etc.)	CPointer

 Table 4-2
 Relationships of C return values to Actuate Basic data types

#### Working with a Java object

Actuate supports access to Java objects using the Java Native Interface (JNI). To perform the following tasks, access a Java object using Actuate Basic:

- Create a Java object.
- Invoke an instance method on the instances.
- Invoke a static method on the class of the object.
- Access an instance variable on the instances.
- Access a static variable on the class of the object.

#### About Java requirements

To access a Java object using Actuate Basic, you must use Java Runtime Environment (JRE) or Java Development Kit (JDK) Version 1.2 or higher. You must verify the installation of the JRE or JDK. If a required DLL is missing, JRE can close an Actuate report when creating a Java Virtual Machine.

Add the class definitions of the Java classes to the CLASSPATH variable.

#### Creating a Java object

Before creating a Java object, you must declare a variable to refer to the object. For example:

Dim theObject As Object

You then use the CreateJavaObject function to create the Java object using the following syntax:

```
Set theObject = CreateJavaObject("<class identifier>")
```

### Invoking a method and accessing a field on a Java object

To perform actions on the object, you invoke the methods of a Java object using the object.method syntax. If the method does not exist for the object, Actuate displays a user error message.

To access a field on the object, name the field following the handle to the object. For example:

someint = theObject.internalValue

To set a field to a specific value, use the following syntax:

```
theObject.internalValue = 10
```

Use the same syntax to access static fields and methods.

Actuate makes no distinction between code that accesses a field and code that invokes a method without parameters. For example, the following lines of code are considered the same:

```
someint = theObject.InternalValue()
someint = theObject.InternalValue
```

Similarly, if an instance method and a static method have the same name and the same or similar signatures, Actuate makes no distinction between them. Specifically, Actuate makes no distinction between integral data types such as long, short, and int, and float data types such as single and double. For example, there is no distinction between the following methods:

```
Myfunction(short, short, long)
Myfunction(int, int, int)
```

#### Invoking a static method and accessing a static field

You can invoke a static method and access a static field without instantiating an object of the class. In the following example, TheStaticMethod is the name of the method to call and theStaticField is the name of the field to access:

```
Dim theClassObject As Object
Set theClassObject = CreateJavaClassHandle( <classname> )
theClassObject.TheStaticMethod( <parameters> )
theClassName.theStaticField = 10
```

#### Converting a Java data type

Table 4-3 shows how Java data types convert to Actuate Basic data types.

Java data type	Actuate Basic type
boolean	Boolean
byte	Integer
char	Integer
double	Double
float	Single
int	Integer
long	Integer
Object	Object

 Table 4-3
 Relationships of Java to Actuate Basic data types

Table 4-3	Relationships of Java to Actuate Basic data types
-----------	---

Java data type	Actuate Basic type
short	Integer
void	No type

#### **Converting a Java String**

Actuate e.Report Designer Professional supports conversions between a Java String and an Actuate Basic string. Actuate converts only the entire string. Other Actuate string operations do not work on a Java String. e.Report Designer Professional supports the following operations:

- Converting an Actuate Basic string to a Java String object.
- Passing an Actuate Basic string to a method that expects a Java String object.
- Converting a Java String object to an Actuate Basic string. A Java String object does not convert automatically. Use methods on the Java object to copy a string from a Java String object.

#### **Converting a Java null**

You cannot assign a Java null to an Actuate data type. In C and C++, Null is a constant with a value similar to 0, 0L, or ((void\*)0), depending on the compiler and memory model options.

In Java, null is a keyword denoting a special value for a reference. Null does not necessarily have a value of 0. You cannot convert it to a primitive data type such as Integer or Boolean.

As a workaround to this issue, return an appropriate value instead of null from a method in the Java class file.

#### **Converting an array**

Actuate supports automatic conversion of single-dimension arrays of primitive types to and from Actuate Basic. The assignment operator (=) copies a Java array where each element is a primitive type, such as an int, into an Actuate Basic array. In an assignment operation, you can copy an entire Actuate Basic array of primitive types into a Java array. The operation copies each element in the source array to the destination array until the end of either the source or the destination array. The following example shows how to copy an array of 10 elements:

```
Dim basicIntArr( 10 ) As Integer
Dim javaIntArr As Object
Set javaIntArr = <something returning a Java array>
' Convert a Java integer array to a Basic array
basicIntArr = javaIntArr
```

You access elements in an array using the methods on the array object.

#### About Java exception and error handling

The Actuate Basic Err function returns an integer value corresponding to each user error. You can use either the number or the constant for error handling. Table 4-4 lists the error names and their meanings.

Error name	Meaning
E_JVMCLASSPATHNOTFOUND	Could not get environment variable CLASSPATH
E_JVMCREATEJVMFAILED	Failed to create Java Virtual Machine
E_JVMCLASSNOTFOUND	Could not find the Java class
E_JVMCREATEOBJECTFAILED	Failed to create Java object
E_JVMINVALIDJAVAHANDLE	Invalid Java object or class handle
E_JVMMETHODFIELDACCESS FAILED	Failed to access a method or a field
E_JVMTYPECONVERSIONFAILED	Type conversion failed
E_JAVAEXCEPTIONOCCURRED	Java exception occurred

Table 4-4 Errors

Actuate captures and keeps the last exception object. You can test for the E\_JAVAEXCEPTIONOCCURRED user error and access the exception object using the GetJavaException function. For example:

```
Dim javaObj As Object
On Error Goto HandleError
Set javaObj = CreateJavaObject( "JavaClassName" )
javaObj.TryToDoSomething() ' This call can cause Java Exception
' Write code to handle normal behavior without exception
' then exit the method
HandleError:
Dim errorCode As Integer
errorCode = Err
If errorCode = E_JAVAEXCEPTIONOCCURRED Then
Dim exceptionObj As Object
Set exceptionObj = GetJavaException()
```

```
If exceptionObj.toString() = "SomeJavaExceptionType" Then
            ' Do something or resume execution
    End If
End If
```

#### Debugging a Java object

The Actuate debugger recognizes all external objects. When you declare the object, the debugger indicates that the object type is unknown. When you create the object, the debugger shows the object type. A numeric identifier indicates the object. Handles to the same object show the same identifier.

For a Java String, the debugger indicates that the object is a Java Object and shows the beginning of the string value.

For a Java array, the debugger indicates that the object is a Java array object and shows the object type.
# Part TWO

Actuate Basic Language Reference

# Chapter

# 5

# Language summary

This chapter provides grouped lists of functions and statements that are frequently used together to perform specific programming tasks:

- Arrays
- Classes and instances
- Program flow
- Conversion
- Date and time
- Environment
- Error trapping
- File input and output
- Finances
- Graphics and printing
- Math
- Operators
- Procedures
- Strings
- Variables and constants

# Arrays

Table 5-1Working with a	rrays
Programming task	Function/Statement
Change default lower limit	Option Base
Declare and initialize	Dim, Global, ReDim, Static
Test the limits	LBound, UBound
Reinitialize	Erase, ReDim

# **Classes and instances**

Table 5-2	Working	with	classes	and	instances
-----------	---------	------	---------	-----	-----------

Programming task	Function/Statement
Declare a class	Class
Create an instance of a class	NewInstance, NewPersistentInstance, IsPersistent, CreateJavaObject, CreateJavaClassHandle
Access class variables	GetClassID, GetClassName, GetValue, GetValueType, GetVariableCount, GetVariableName, IsKindOf, SetBinding
Copy class variable values	CopyInstance

# **Program flow**

Programming task	Function/Statement
Branch	FollowHyperlink, GoTo, On Error
Exit or pause the program	End, Stop
Loop	DoLoop, ForNext, WhileWend
Make decisions	IfThenElse, IIf, Select Case

# Conversion

Programming task	Function/Statement
ANSI value to string	Chr[\$]
Binary image file to string	ConvertBFileToString
Color to SVG attribute	SVGColorAttr
Date to serial number	DateSerial, DateValue
Decimal number to others	Hex[\$], Oct[\$]
Font to SVG style	SVGFontStyle
Lists of values into an array	ListToArray
Number to string	Format[\$], Str[\$]
Number to SVG attribute	SVGAttr
Number to SVG string	SVGDbl
One data type to another data type	CCur, CDate, CDbl, CInt, CLng, CSng, CStr, CVar, CVDate, Fix, Int
Serial number to date	Day, Month, Weekday, Year
Serial number to time	Hour, Minute, Second
String to ANSI value	Asc
String to binary image file	ConvertStringToBFile
String to number	Val
String to SVG attribute	SVGAttr
String to SVG string	SVGStr
String to SVG style	SVGStyle
Time to serial number	TimeSerial, TimeValue

### **Table 5-4**Performing conversions

# Date and time

**Table 5-5**Working with dates and times

Programming task	Function/Statement
Control how dates are converted	ParseDate

(continues)

Table 5-5	Working with	dates and	times	(continued)
	- 3			

Programming task	Function/Statement
Convert date to serial number	DateSerial, DateValue
Convert serial number to date	CVDate, Day, Month, Weekday, Year
Convert serial number to time	Hour, Minute, Second
Convert time to serial number	TimeSerial, TimeValue
Convert to date data type	CDate, CVDate
Determine whether an argument can be converted to a date	IsDate
Get current date or time	Date[\$], Now, Time[\$], IsDate
Manipulate date or time	DateAdd, DateDiff, DatePart
Process time	Timer

# Environment

Programming task	Function/Statement
Clear Clipboard	ClearClipboard
Display comments in the status line of the factory dialog	ShowFactoryStatus
Find environment variables	Environ[\$]
Get Clipboard text	GetClipboardText
Get command-line arguments	Command[\$]
Get user's login name	GetServerUserName
Get user's machine name	GetServerName
Get name of Actuate application which is currently running	GetAppContext
Get user's operating system login name	GetOSUserName
Get scaling factor	GetReportScalingFactor

**Table 5-6**Manipulating the environment

Programming task	Function/Statement
Get stage of a report's lifecycle	GetReportContext
Get the name of the Encyclopedia volume on which the file is stored	GetVolumeName
Get user agent string	GetUserAgentString
Get version number of the AFC library	GetAFCROXVersion
Get version number of the factory that the report object executable (.rox) file is using	GetFactoryVersion
Get version number of ROX file	GetROXVersion
OLE	CreateObject
Run other programs	Shell
Temporarily suspend report execution	Sleep
Set Clipboard text	SetClipboardText
Sound beep	Веер

# **Error trapping**

Programming task	Function/Statement
Get error messages	Error, Error\$, GetJavaException
Get error status	Err, Erl
Insert explanatory remarks	Rem
Prevent division-by-zero error	SafeDivide
Prevent logic error by predetermining whether a condition is true	Assert

(continues)

Programming task	Function/Statement
Set Err function to a given value	Err
Simulate run-time errors	Error
Trap errors while program is running	Assert, On Error, Resume

**Table 5-7**Trapping errors (continued)

# File input and output

Table 5-8Working with files

Programming task	Function/Statement
Access or create a file	Open
Add directories to search for included images	ExtendSearchPath
Close a file	Close, Reset
Control output appearance	Space, Tab, Width
Copy one file to another	FileCopy
Determine which open report object instance (.roi) file is the default used when New Persistent() is called to instantiate an object	SetDefaultPOSMFile
Determine which formats are supported in a DHTML environment for displaying report pages and results of a search operation	GetSearchPageFormats, GetViewPageFormats, IsSearchFormatSupported, IsViewPageFormatSupported
Get information about a file	EOF, FileAttr, FileDateTime, FileExists, FileLen, FileTimeStamp, FreeFile, Loc, LOF, Seek, Seek2
Manage disk drives and directories	ChDir, ChDrive, CurDir[\$], MkDir, RmDir
Manage files	Kill, LockUnlock, Name
Read from a file	Get, Input, Input[\$], InputB[\$], Line Input
Retrieve strings from resource files	StrLoad[\$]

Table 5-8Working with files

Programming task	Function/Statement
Search for a file through the existing search path	FindFile
Set object aging rules on a report object web (.row) file	SetStructuredFileExpiration
Set or get file attributes	GetAttr, SetAttr
Set read-write position in a file	Seek
Write to a file	Print, Put, Write

# Finances

Table 5-9	Working with finances
-----------	-----------------------

Programming task	Function/Statement
Depreciation of assets	DDB, SLN, SYD
Get annuity information	FV, IPmt, NPer, Pmt, PPmt, PV, Rate
Get net present value and rate of return	IRR, NPV, MIRR

# **Graphics and printing**

**Table 5-10**Working with graphics and printing

Programming task	Function/Statement
Decode string into binary image file	ConvertStringToBFile
Encode binary image file into string	ConvertBFileToString
Dynamically adjust the height of a container to accommodate a large amount of text	GetDisplayHeight (Windows only)
Print text	Print, Write
Work with colors	QBColor, RGB

# Math

	•
Programming task	Function/Statement
General calculation	Exp, Log, Sqr
Generate random numbers	Randomize, Rnd
Get absolute value	Abs
Get the sign of expressions	Sgn
Numeric conversions	Fix, Int
Trigonometry	Atn, Cos, Sin, Tan

### Table 5-11 Working with mathematical expressions

# Operators

Table 5-12	Working with	mathematical	operators
------------	--------------	--------------	-----------

Programming task	Function/Statement
Arithmetic	*, +, -,  /, ^, Mod
Comparison	<, =<, >, >=, =, <>, &, Is
Concatenation	&
Logical expressions	And, BAnd, Eqv, Imp, Not, BNot, Or, BOr, Xor
Pattern matching	Like

# **Procedures**

Table 5-13	Working with procedures
------------	-------------------------

Programming task	Function/Statement
Call Sub procedure	Call
Declare a reference to an external procedures	Declare
Define a procedure	FunctionEnd Function, SubEnd Sub
Exit from a procedure	Exit, FunctionEnd Function, SubEnd Sub

# Strings

Programming task	Function/Statement
Assign value	Let
Compare two strings	StrComp
Convert to lowercase or uppercase letters	LCase[\$], UCase[\$]
Copy to and from Clipboard	SetClipboardText, GetClipboardText
Create strings of repeating characters	<pre>Space[\$], String[\$], StringW[\$]</pre>
Find the length of an expression in bytes	LenB
Find the length of a string	Len
Format strings	Format[\$]
Justify strings	LSet, RSet
Manipulate strings	InStr, Left[\$], LTrim[\$], Mid[\$], RevInStr, Right[\$], RTrim[\$], Trim[\$]
Manipulate bytes	InStrB, LeftB[\$], RightB[\$], MidB[\$]
Replace part of a string with a different substring	StrSubst
Set string comparison rules	Option Compare
Translate from one language to another	Open[\$]
Work with UCS-2 encoding values	AscW, ChrW[\$]
Work with ASCII and ANSI values	Asc, Chr[\$]

Table 5-14Working with strings

# Variables and constants

Programming task	Function/Statement
Add search indexes on variables of components	AddValueIndex
Assign object references to variables	Set
Declare aliases for existing object types	TypeAs
Declare user-defined variables or record structures	TypeEnd Type
Declare variables and constants	Const, Dim, Global, Static
Get information about Variant variables	IsDate, IsEmpty, IsNull, IsNumeric, VarType

 Table 5-15
 Working with variables and constants

## Chapter

# 6

# **Statements and functions**

This chapter provides an alphabetical list of all Actuate Basic functions and statements. Each entry includes a general description of the function or statement, its syntax, and a description of its parameters and return values.

For conceptual information about working with Actuate Basic, see Part 1, "Working with Actuate Basic."

# Using the code examples

The examples in this chapter illustrate how to use individual statements and functions. They are not necessarily real-world examples of how to create a report. The easiest way to use the Actuate Basic code examples is to copy the code from the code example window in online help and paste it in Method Editor in e.Report Designer Professional.

Not all examples have a Declaration section. If the example you are using does not include a Declare...End Declare statement, you do not have to paste anything into a Basic source (.bas) file.

To examine the effect the sample programs have, it is best to step through them using the e.Report Designer Professional debugger. For information about the debugger, see *Accessing Data using e.Report Designer Professional*.

### How to open an example in online help

- 1 Choose Help→Contents.
- 2 In the Table of Contents, navigate to Programming with Actuate Basic→Statements and functions→Using the code examples.
- **3** Under the topic title, choose Example. The code example displays.

### How to use a code example in a report design

- 1 Create a new report or open an existing report. This procedure opens Forecast.rod in eRDPro\Examples\DesignAndLayout\Forecast as an example.
- **2** If the code example has a Declaration section, perform the following steps:
  - 1 Create a Basic source (.bas) file by choosing Tools→Library Organizer. Library Organizer appears.
  - 2 Choose New. New Library appears.
  - **3** In File name, type the name for the new file. In Save as type, choose Source file (.bas). Choose Save. The source file appears under Libraries included in your report.
  - 4 Perform the following steps in Library Organizer:
    - 1 Choose OK.
    - 2 In the code example window of the online help, select the code.
    - 3 Copy and paste the Declare section into your BAS.
    - 4 Save the .bas file.
- **3** In the e.Report Designer Professional application:

- 1 In Report Structure, double-click the ForecastApp report section object. The Properties page appears within the Properties window.
- 2 Choose Methods. Methods page appears.



4 In the editor, paste in the remaining code from the example under:

Super::Start( )

-

- **4** Compile and run your report:
  - 1 Choose Report → Build and Run. The function runs and displays any prompts.
  - **2** Respond to the prompts that appear. The function displays the final result and your report appears.
- **Example** The following example demonstrates the FV function. The example shows where to paste the different parts of a code example. To use this example, perform the following steps in this order:
  - Paste the following code into your Actuate Basic source code (.bas) file:

```
Declare
Global Const ENDPERIOD = 0
Global Const BEGINPERIOD = 1
End Declare
```

 Paste the following code into the Start subroutine of your AcReport object under Super::Start():

```
Dim EachPmt As Double, APR As Double
Dim TotalPmts As Double, PayWhen As Integer
Dim PresentVal As Double, FutureVal As Double
Dim Msg as String, Fmt As String
' Specify money format
Fmt = "$#, ##0.00"
EachPmt = 2000 ' Amount to save each month
APR = 0.0325 ' The annual percentage rate for the interest
TotalPmts = 60 ' The number of months to save
' Assume payment at month end
PayWhen = ENDPERIOD
PresentVal = 12000 ' Amount in the savings account now
' Now do the real work
FutureVal = FV(APR / 12, TotalPmts, -EachPmt, -PresentVal,
+ PayWhen)
' Format the resulting information for the user
```

```
Msg = "Starting with " & Format(PresentVal, Fmt)
+ & " and saving " & Format(EachPmt, Fmt)
+ & " every month at an interest rate of "
+ & Format(APR, "##.00%") & " for a period of "
+ & TotalPmts & " months will give you "
+ & Format(FutureVal, Fmt) & "."
ShowFactoryStatus( Msg )
```

- Build and run the report.
- Respond to the prompt.
- The result appears, followed by the report.

# **Abs function**

Returns the absolute value for a number or expression.

```
Syntax Abs(<number expression>)
```

### Parameters <number expression>

Number, numeric expression, or Variant of VarType 8 (String) that specifies the number for which you want to find the absolute value. If <number expression> is a String, it is parsed according to the formatting rules of the current run-time locale. For example, the following statement returns 123.456 on a French run-time locale and 123456.00 on an English run-time locale:

Abs("123,456")

**Returns** Number, numeric expression, or Variant of VarType 8 (String). If <number> evaluates to Null, Abs returns Null.

**Example** The following example demonstrates the return value of Abs:

```
Sub Start()
  Dim Msg as String
  Super::Start( )
  ' Returns 25
  Msg = "The result of Abs( 25 ) is " & Abs( 25 )
  ShowFactoryStatus ( Msq )
  ' Also returns 25
  Msq = "The result of Abs(-25) is " & Abs(-25)
  ShowFactoryStatus ( Msq )
  ' Returns 0
  Msg = "The result of Abs( ABC ) is " & Abs( "ABC" )
  ShowFactoryStatus ( Msq )
  ' Returns 123
  Msg = "The result of Abs( 123four ) is " & Abs( "123four" )
  ShowFactoryStatus ( Msg )
End Sub
```

See also Sgn function

# **Acos function**

Returns the arc cosine of an angle.

Acos( <cosval>)</cosval>	
<cosval> A number, numeric expression, or Variant of VarType 8 (String) that specifies value for which you want to find the arc cosine. The value of <cosval> should in the range of -1 to 1. If <cosval> is a String, it is parsed according to the formatting rules of the current run-time locale.</cosval></cosval></cosval>	
To convert between radians and degrees, use: radians = degrees * Pi/180.	
Double, in the range of 0 to Pi radians.	
If <cosval> evaluates to Null, Acos returns Null.</cosval>	
The following example generates the cosine of an angle expressed in radians, then returns the arc cosine of the cosine:	
<pre>Sub Start() Dim Pi As Double Dim CosVal as Double Dim Msg As String Super::Start() Pi = 3.14159265358979 ' Use Pi/4, or 45 degree, angle CosVal = Cos( Pi / 4) Msg = "The arccosine of " &amp; CosVal &amp; " is: " &amp; Acos( CosVal ) ShowFactoryStatus( Msg ) End Sub</pre>	

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Asin function Atn function Cos function Sin function Tan function

# AddBurstReportPrivileges function

Adds one privilege setting to a role or user.

### Syntax AddBurstReportPrivileges(<id>, <userOrRole>, <privileges>)

**Description** This function sets privileges on a burst report for a single user or role. You can call this method several times to set privileges of multiple roles or users. The setting affects the current burst report only. You call this method from SetBurstReportPrivileges() in the AcReport class. If AddBurstReportPrivileges() is not called during a burst report generation, the burst report will have the same privileges as its primary report.

### Parameters <id>

String expression specifying the role or user id like role name, for example, "Admin", or a user name, such as "Jon".

### <userOrRole>

String expression specifying either "Role" or "User".

### <privileges>

String expression containing list of privileges separated by commas. For example, "READ, WRITE."

All parameters are case insensitive.

**Example** The following example overrides the subreport's SetBurstReportPrivileges() method to add read and write rights for the user bob, and secure read rights to the role sales:

```
Sub SetBurstReportPrivileges( row As AcDataRow )
   AddBurstReportPrivileges( "bob", "User", "READ,WRITE" )
   AddBurstReportPrivileges( "Sales", "Role", "SECURE READ" )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

# AddValueIndex function

Adds a search index on a variable of a component.

Syntax AddValueIndex(<intFileID>, <strClassName>, <strAttrName>)

**Description** Use AddValueIndex to add a search index to an ROI. A search index speeds the retrieval of information when searching a report, although at the cost of a somewhat larger ROI and somewhat longer Factory run. AddValueIndex() must be called in the Factory during report generation. You cannot add an index to an ROI once it is created.

Each index speeds access to one variable in one class. The easiest way to create an index is to set the Searchable property of a class to True. This adds an index on the DataValue variable of that class. Use AddValueIndex() to search on any other variable and add an index on that variable. You must create a separate index for each variable you want indexed. For each, you must specify the ROI in which to create the index, the class name, and the variable name within the class.

A typical circumstance in which you specify a different variable is when a hyperlink specifies a value that is related to the data stored in the DataValue variable. For example, a data control displays a customer's name, while an associated hyperlink uses the customer's ID. When indexing this hyperlink, specify an additional variable, perhaps called LinkTag, that contains the customer's ID.

### Parameters <intFileID>

Integer expression. The file ID of the report object instance (.roi) (POSM file) to which you want to add the new index. The value returned from creating or opening an ROI.

This is in most cases the ROI opened in the Factory. An easy way to get this ID is from any persistent object in your ROI. To do so, use code like the following:

```
Dim roiFile As Integer
roiFile = GetPosmFile( thePersistentObject )
AddValueIndex( roiFile, "yourClass", "yourVar" )
```

You must designate a valid ROI that is open in the Factory.

### <strClassName>

String expression. The fully qualified name of the class of the component with the variable you want to index.

You must specify an existing class by its fully qualified name, and the class must have persistent instances.

To determine the fully qualified class name, use one of the following techniques:

- Consult the Class tab of the Property Sheet for the component to find its name and scope. Specify <strClassName> according to the form: "scope1::scope2::name".
- If you have an instance of the class, call GetClassName on the object.

### <strAttrName>

String expression. The name of a variable in the class to be indexed. The attribute name:

- Must be a variable in the class specified by <strClassName>.
- Must be a variable, not a property. The variable can have the same name as a property.

- Cannot be a structure such as Size or Rectangle.
- Must be of a simple scalar data type such as Integer, Date, Currency, String, or Double. Can also be a type alias for a scalar type such as Boolean or Twips.

### Returns Boolean

- True if all arguments are correct.
- False if any of the arguments are invalid.
- **Tips** A convenient place to insert calls to AddValue is in the Finish() method for your subclass of AcReport, the top component in the Structure View.
  - Hyperlinks use the search engine. Therefore, to improve your report's performance when executing hyperlinks, use AddValueIndex to index hyperlinks.
  - Do not confuse properties and variables of an object. They often have the same names, especially when a property specifies the default value for a variable. To avoid confusion, use the Component Editor's Variables page to see what variables are available.
  - To facilitate debugging, use Verify to ensure that, if any argument to AddValueIndex is invalid, the Factory will halt. For example, use code like the following:

```
Verify(AddValueIndex( 0, "AFrame::ItemCode", "DataValue" ))
```

**Example** The following example assumes you have a report that displays a list of customers by name. Each customer is identified by a customer number variable, but you do not want to display this number, so you created a CustomerNumber variable on the CustomerName control in the CustomerFrame. The root AcReport component is CustomerListing.

Add an index on the CustomerNumber variable using the following code in the Finish method of CustomerListing:

```
Sub Finish( )
Dim theROIFile As Integer
Super::Finish( )
' Get the ROI file handle for the ROI that contains this
' component
theROIFile = GetPOSMFile( me )
' Add a custom index for the CustomerNumber variable in the
' CustomerListing::CustomerFrame::CustomerName control.
' Quit the Factory if something goes wrong
Verify( AddValueIndex( theROIFile,
+ "CustomerListing::CustomerFrame::CustomerNumber",
+ "CustomerListing::CustomerFrame::CustomerNumber",
+ "CustomerNumber" ) )
End Sub
```

# **Asc function**

Returns the ANSI code that corresponds to the given character.

Syntax Asc(<string expression>)

### Parameters <string expression>

String expression that represents a character to translate to a corresponding code in the ANSI character set.

**Returns** Integer between 0 and 65535.

The return value is the character code value produced by converting <string expression> from Unicode to Multibyte character set (MBCS) using the current run-time encoding. The following rules apply to the conversion:

- If <string expression> is longer than one character, Asc translates only the first character.
- If <string expression> evaluates to Null, Asc returns Null.
- If the string in <string expression> does not match the run-time encoding, Asc might return an incorrect value. For example, if the report runs on a system that uses English (ASCII) encoding, passing a Japanese string produces incorrect results because Japanese characters cannot be converted to ASCII.

### **Tips** To translate an ANSI code number to its corresponding character, use Chr\$.

• If the run-time encoding is UCS-2, Asc behaves in the same way as AscW.

# **Examples** The following Asc() statements are equivalent. Each returns the numeric value 90.

```
Sub Start( )
Super::Start( )
ShowFactoryStatus( Str(Asc("Z")) )
ShowFactoryStatus( Str(Asc(Mid$("XYZ",3,1))) )
ShowFactoryStatus( Str(Asc(UCase$("zebra"))) )
End Sub
```

The following example creates a name and displays the ANSI value of each letter in that name:

```
Sub Start( )
Dim I As Integer
Dim Msg As String, AName As String
Super::Start( )
AName = "Pallavi Sharma"
```

```
For I = 1 To Len( AName )
    Msg = Msg & Mid$( AName, I, 1 ) & " ---> "
+    & Asc( Mid$( AName, I, 1 ) )
    Next I
    ShowFactoryStatus( Msg )
End Sub
```

See also AscW function Chr, Chr\$ functions ChrW, ChrW\$ functions

# **AscW** function

Returns the code point of the given character in UCS-2 encoding.

Syntax	AscW( <string expression="">)</string>
Parameters	<pre><string expression=""> String expression that represents a character to translate to a corresponding code point in UCS-2 encoding. If <string expression=""> is longer than one character, AscW translates only the first character.</string></string></pre>
Returns	Integer between 0 and 65535. If <string expression=""> evaluates to Null, AscW returns Null.</string>
Example	The following example creates a name and displays the ANSI value of each letter in that name:
	<pre>Sub Start( )    Dim I As Integer    Dim Msg As String, AName As String    Super::Start( )    AName = "Vishál Tayal"    For I = 1 To Len( AName )         Msg = Msg &amp; Mid\$( AName, I, 1 ) &amp; "&gt; " +</pre>

For information about using the code examples, see "Using the code examples," earlier in this chapter.

**See also** ChrW, ChrW\$ functions

# **Asin function**

Gives the arc sine of an angle.

Syntax Asin(<sinval>)

### Parameters <sinval>

A number, numeric expression, or Variant of type 8 (String) that specifies, in the value for which you want to find the arc sine. The value of <sinval> should be in the range of -1 to 1. If <sinval> is a String, it is parsed according to the formatting rules of the current run-time locale.

To convert between radians and degrees, use: radians = degrees \* Pi/180.

**Returns** Double, in the range of -Pi/2 to Pi/2 radians.

If <sinval> evaluates to Null, Asin returns Null.

**Example** The following example generates the sine of an angle expressed in radians and returns the arcsine of the sine:

```
Sub Start( )
Dim Pi As Double
Dim SinVal as Double
Dim Msg As String
Super::Start( )
Pi = 3.14159265358979
' Use Pi/4, or 45 degree, angle
SinVal = Sin( Pi / 4 )
Msg = "The arcsine of " & SinVal & " is: " & Asin( SinVal )
ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Acos function Atn function

Cos function Sin function Tan function

# Assert statement

Halts the program and displays an error message when a given condition is False. **Syntax** Assert(<condition>)

### Atn function

**Description** When <condition> is True, Assert does nothing. When False, Assert halts the program and displays an error message. Use Assert to trap errors.

### Parameters <condition>

Any numeric or string expression that can be evaluated. When <condition> evaluates to zero or Null, the condition is False, otherwise <condition> is True.

**Example** The following example uses Assert to halt a program before a function attempts to divide a number by zero. This prevents the program from attempting to perform an illegal operation.

```
Function PercentOf(num As Integer, denom As Integer) As Integer
Assert(denom <> 0)
PercentOf = num * 100 / denom
End Function
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Erl function Err function Error statement Error, Error\$ functions On Error statement

# Atn function

Gives the arctangent of a number.

Syntax	Atn( <tanval>)</tanval>
Parameters	<tanval> Number, numeric expression, or Variant of VarType 8 (String) that specifies the tangent value for which you want to find the arctangent. If <number> is a String, it is parsed according to the formatting rules of the current run-time locale.</number></tanval>
	For example, the following statement returns 0.89 on a French run-time locale and 1.57 on an English run-time locale:
	Atn("1,234")
	To convert to radians from degrees, use radians = degrees $* Pi/180$ .
Returns	The arctangent, in radians, as a double.
	If <number> evaluates to Null, Atn returns Null.</number>
Example	The following example generates the tangent of an angle in radians and returns the value of the arctangent given that tangent value:

```
Sub Start( )
Dim tanVal As Double, dblDegree As Double, Pi As Double
Dim Msg As String
Super::Start( )
Pi = 3.14159265358979
' Use Pi / 4, or 45 degree, angle
tanVal = tan( pi / 4 )
Msg = "The arctangent of " & tanVal & " is " & Atn(tanVal)
ShowFactoryStatus( Msg )
End Sub
```

See also Acos function Asin function Cos function Sin function

Tan function

# **Beep statement**

Sounds a short tone through the computer's speaker.

- Syntax Beep
- **Description** To use Beep, sound must be turned on in the sounds section of the Windows Control Panel. You cannot control the tone or the duration of the sound using Beep. Beep is not supported when a report runs on iServer.
  - **Example** The following example prompts the user for a number. If the number falls outside a certain range recognized as valid, the computer beeps and displays a message.

```
Sub Start( )
Dim Number As Integer, Msg As String
Super::Start( )
Number = Rnd * 10
' Validate range
If Number >= 1 And Number <= 5 Then
Msg = "The number " & Number & " is between 1 and 5."
Else
' Beep if not in range
Beep
Msg = "The number " & Number & " is not between 1 and 5."
End If
ShowFactoryStatus( Msg )
End Sub</pre>
```

# **Call statement**

Transfers program control to an Actuate Basic Sub procedure or to a Windows dynamic-link library (DLL) procedure.

Syntax [Call] <procedure name> [(<argument list>)]

**Description** You are never required to use the Call keyword when you call a procedure. If you use the Call keyword to call a procedure that requires arguments, <argument list> must be enclosed in parentheses.

You pass variable arguments to <procedure name> in two ways, by reference or by value. When you pass variables by reference, Actuate Basic assigns their contents (values) actual addresses. The called procedure can alter the variables. When you pass variables by value, Actuate Basic assigns their contents (values) temporary addresses. The called procedure cannot alter the contents of the variables.

### Parameters Call

Optional keyword that indicates that Actuate Basic is to transfer program control to another procedure.

### <procedure name>

Specifies the name of the procedure to which the program is to transfer control.

### <argument list>

List of variables, arrays, or expressions to pass to the procedure as arguments.

By default, arguments in list are passed by reference. To pass an entire array, use its name followed by empty parentheses.

For example, the following statement transfers control to a Sub procedure called ManyBeeps and passes the value 15 to it as an argument:

Call ManyBeeps(15)

The following statement transfers control to a Sub procedure called ManyBeeps and passes the value 15 to it as an argument, but without using the Call keyword:

ManyBeeps(15)

The following statement transfers control to a Sub procedure called MySortArray, and passes the integer array NewColors% to it:

Call MySortArray(NewColors%())

**Tip** If you are having trouble passing arguments to DLL procedures, use the ByVal keyword to pass them by value. In many cases, external DLL routines might not

support all Actuate Basic data types, and using ByVal to call such a DLL routine allows the routine to convert the arguments so that it can accept them.

**Example** The following example shows two ways of calling the MessageBeep procedure in User.exe, a Microsoft Windows DLL. First, you must declare the procedure MessageBeep from the Windows dynamic link library at or near the beginning of your Actuate Basic source code (.bas) file.

Declare Sub MessageBeep Lib "User32" (ByVal N As Integer)

Then, paste the following example in the method editor:

```
Sub Start( )
Dim I As Integer
Super::Start( )
' Call Windows procedure
Call MessageBeep(0)
' Insert delay between calls
Sleep(2)
' Call again without Call keyword
MessageBeep(0)
Prd Outh
```

End Sub

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Declare statement

# **CCur function**

Converts a numeric expression to the Currency data type.

- Syntax CCur(<numeric expression>)
- **Description** CCur is one of nine Actuate Basic data type conversion functions. If <numeric expression> lies outside the acceptable range for Currency, an overflow error occurs.

### Parameters <numeric expression>

Numeric expression to convert from the Date, Integer, Long, Single, Double, String, or Variant data type to the Currency data type. The following conditions apply to <numeric expression>:

- If <numeric expression> is Null, CCur returns Null.
- If <numeric expression> is a String or Variant, it must be interpretable as a number.
- If <numeric expression> is a String, it is parsed according to the formatting rules of the current run-time locale.

For example, each of the following statements returns 123. In the second statement, the string 123 enclosed in quotation marks is considered a numeric expression because it can be interpreted as a number.

In the third statement, four cannot be interpreted as a number.

```
CCur(123)
CCur("123")
CCur("123four")
```

The following statement first converts the string 123four using Val to convert the number 123. Val ignores the string component (four).

```
CCur(Val("123four"))
```

The following statement returns 123.456 on a French run-time locale and 123456.00 on an English run-time locale:

CCur("123,456")

- **Returns** Currency. If <numeric expression> cannot be interpreted as a number, zero is returned.
  - **Tips** To convert an expression to currency without explicitly using CCur, assign the expression to a variable that is the Currency data type.
    - To declare the data type of a variable, use Dim with the As clause.
    - To round a number to precisely the number of decimal places you want and discard anything residual, use Format\$.
    - To ensure that you pass the correct data type to a Sub procedure or a Windows Dynamic Link Library (.dll), use data type conversion functions.
- **Example** The following example generates a random number, converts that number using each of the conversion functions in turn, then displays the result:

```
Sub Start()
  Dim Orig As String, Msg As String
  Super::Start( )
  ' Get a random number between 1 and 256
  Orig = 255 * Rnd + 1
  ' Convert to various formats
  Msg = "Your number is: " & Orig
  ShowFactoryStatus ( Msg )
  Msq = "CCur( Orig ) yields ----> " & CCur( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CDate( Orig ) yields ----> " & CDate( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CDbl( Orig ) yields ----> " & CDbl( Orig )
  ShowFactoryStatus ( Msq )
  Msg = "CInt( Orig ) yields ----> " & CInt( Orig )
  ShowFactoryStatus ( Msg )
```

```
Msg = "CLng( Orig ) yields ----> " & CLng( Orig )
ShowFactoryStatus ( Msg )
Msg = "CSng( Orig ) yields ----> " & CSng( Orig )
ShowFactoryStatus ( Msg )
Msg = "CStr( Orig ) yields ----> " & CStr( Orig )
ShowFactoryStatus ( Msg )
Msg = "CVar( Orig ) yields ----> " & CVar( Orig )
ShowFactoryStatus ( Msg )
```

End Sub

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also CDate function CDbl function CInt function CLng function CSng function CStr function CVar function CVDate function Dim statement Format, Format\$ functions

# **CDate function**

Converts an expression to a Date.

Syntax CDate(<date expression>)

**Description** CDate is one of nine Actuate Basic data type conversion functions. Do not use CDate to convert a long date format that also contains a day-of-the-week string, like Friday, Nov. 12, 1982. The function does not recognize such strings as dates.

Avoid supplying CDate with a date in a format other than the format the locale map specifies.

### Parameters <date expression>

Date expression, or any numeric or string expression that can be interpreted as a date, a time, or both a date and a time:

- Can be a string such as November 12, 1982 8:30 P.M., 12 Nov., 1982 08:30 PM, 11/12/82, 08:30pm, or any other string that can be interpreted as a date, a time, or both a date and a time in the valid range.
- Can be a date serial number such as 30267.854, which corresponds to November 12, 1982 at 8:30 P.M., or any other number that can be mapped to a date, time, or both a date and a time in the valid range.

 For date serial numbers, the integer component represents the date itself while the decimal component represents the time of day on that date, where January 1, 1900 at precisely noon has the date serial number 2.5, and negative numbers represent dates prior to December 30, 1899 (0).

Default date if none specified: 0 (Dec. 30, 1899) Default time if none specified: 00:00:00 (midnight)

The following conditions apply to <date expression>:

- <date expression> must not be empty or zero-length.
- <date expression> cannot contain a day of week.
- <date expression> must specify a date within the range January 1, 100 through December 31, 9999, inclusive.
- <date expression> is parsed according to the formatting rules of the current run-time locale.
- If <date expression> includes a time of day, it must be a valid time, even if CDate is not being used to return anything having to do with a time of day. A valid time is one that is in the range 0:00:00 (12:00:00 A.M.) through 23:59:59 (11:59:59 P.M.). Either the 12- or 24-hour clock can be used. The time must be in one of the following formats:

```
hh:mm:ss
hh:mm
hh
```

- <date expression> cannot include a day of week.
- If <date expression> is a numeric expression, it must be in the range -657434 to +2958465, inclusive.

The following assumes that the AC\_CENTURY\_BREAK property is set to the default value of 30. For information about using AC\_CENTURY\_BREAK, see *Accessing Data using e.Report Designer Professional.* 

- For abridged date expressions ending in 00 through 29 that do not specify the century, relevant date functions estimate the unabridged year by adding 2000. For example, while Year(#4/30/1910#) returns 1910, the abridged expression Year(#4/30/10#) returns 2010 (10 + 2000).
- For abridged date expressions ending in 30 through 99 that do not specify the century, relevant date functions estimate the unabridged year by adding 1900. For example, while Year(#11/12/2082#) returns 2082, the abridged expression Year(#11/12/82#) returns 1982 (82 + 1900).

Note that a date can contain an abridged month, such as Nov. 12, 1982 as opposed to November 12, 1982.

**Examples** The following statements are equivalent. Each assigns 11/12/82 8:30:00 P.M., with its underlying serial number to the variable BDate.

```
BDate = CDate("11/12/82 8:30 pm")
BDate = CDate("12 Nov 1982 20:30")
BDate = CDate("November 12, 1982 8:30PM")
BDate = CDate(30267.854167)
```

### Returns Date

- If <date expression> is Null, CDate returns Null.
- If <date expression> contains a day of week, CDate returns Null.
- If <date string> contains only numbers separated by valid date separators, CDate recognizes the order for month, day, and year according to the settings of the current run-time locale.
- If <date expression> cannot be evaluated to a date, CDate returns Null.
- If <date expression> fails to include all date components, such as day, month, and year, CDate returns Null.
- If <date expression> contains only the time and the time is valid, CDate returns December 30, 1899 with the time.
- Tips To determine if either a numeric or string expression that looks like a date can be converted to a date, use IsDate. However, IsDate cannot determine whether or not a string that looks like a number can be converted to a date, even if the number is in the correct range for date serial numbers.
  - If you use a date expression in one locale, it might be misinterpreted in another locale. For instance, in the United States, 1/2/2005 means January 2nd, 2005, but in France, it means February 1st, 2005. To avoid such ambiguities, use DateSerial to specify all dates.
- **Example** The following example generates a number. For any valid value, the example displays both a date and a date serial number.

```
Sub Start( )
Dim Msg As String, DateMeme As String, ConvDateMeme As Date
Super::Start( )
DateMeme = CStr(Rnd * 125 * 365)
ConvDateMeme = CDate( DateMeme )
Msg = "The date is: " & ConvDateMeme
ShowFactoryStatus( Msg )
Msg = "The serial number for that date is: "
+ & CDbl( ConvDateMeme )
ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also CCur function CDbl function CInt function CLng function CSng function CStr function CVar function CVDate function Dim statement Format, Format\$ functions IsDate function

# **CDbl** function

Converts a numeric expression to the Double data type.

- Syntax CDbl(<numeric expression>)
- **Description** CDbl is one of nine Actuate Basic data type conversion functions. If <numeric expression> lies outside the acceptable range for a Double data type, an overflow error occurs. If <numeric expression> is Null, CDbl returns Null.

### Parameters <numeric expression>

Numeric expression to be converted from the Currency, Date, Integer, Long, Single, String or Variant data type to the Double data type. The following conditions apply to <numeric expression>:

- If <numeric expression> is a String or Variant, it must be interpretable as a number.
- If <numeric expression> is a String, it is parsed according to the formatting rules of the current run-time locale.

For example, each of the following statements returns the value 123. In the second statement, the string 123 enclosed in double quotation marks is considered a numeric expression because it can be interpreted as a number. In the third statement, four cannot be interpreted as a number.

```
CDbl(123)
CDbl("123")
CDbl("123four")
```

The following statement uses Val to convert the string 123four to the number 123. Val ignores the string component (four).

```
CDbl(Val("123four"))
```

The following statement returns 123.456 on a French run-time locale and 123456.00 on an English run-time locale:

```
CDbl("123,456") returns
```

- **Returns** Double. If <numeric expression> cannot be interpreted as a number, zero is returned.
  - **Tips** To convert an expression to double without explicitly using CDbl, assign the expression to a variable that is the Double data type.
    - To declare the data type of a variable, use Dim with the As clause.
    - To round a number to precisely the number of decimal places you want and discard anything residual, use Format\$.
    - To be sure you pass the correct data type to a subprocedure or to a Windows Dynamic Link Library (DLL), use data type conversion functions.
- **Example** The following example generates a random number, converts that number using each of the conversion functions in turn, then displays the result:

```
Sub Start()
  Dim Orig As String, Msg As String
  Super::Start( )
  ' Get a random number between 1 and 256
  Orig = 255 * Rnd + 1
  ' Convert to various formats
  Msg = "Your number is: " & Orig
  ShowFactoryStatus ( Msg )
  Msg = "CCur( Orig ) yields ----> " & CCur( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CDate( Orig ) yields ----> " & CDate( Orig )
  ShowFactoryStatus ( Msq )
  Msg = "CDbl( Orig ) yields ----> " & CDbl( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CInt( Orig ) yields ----> " & CInt( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CLng( Orig ) yields ----> " & CLng( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CSng( Orig ) yields ----> " & CSng( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CStr( Orig ) yields ----> " & CStr( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CVar( Orig ) yields ----> " & CVar( Orig )
  ShowFactoryStatus ( Msg )
End Sub
```

See also CCur function CDate function CInt function CLng function CSng function CStr function CVar function CVDate function Dim statement Format, Format\$ functions Int function

# **ChDir statement**

Changes the current working directory on the specified or current drive.

Syntax ChDir <path name>

### Parameters <path name>

String expression that specifies the name of the target directory.

<path name> has the following syntax:

[<drive:>] [\] [<directory>\<directory>]... (Windows)

[/] [<directory>/<directory>]... (UNIX)

<path name> must contain fewer than 259 characters and must be a valid directory.

### <drive:>

(Windows only.) Character, followed by a colon, that specifies the drive. If you do not specify <drive:>, ChDir uses the current drive. If you specify a drive other the than current one, Actuate Basic only changes the current directory on that drive; it does not change the current drive to <drive:>.

### <directory>

String expression of the name of the directory or subdirectory to make the current directory.

For example, the following statement changes the current working directory on drive D: to Mydocs, even if the current drive is C:, but does not change the current drive to D:

ChDir "D:\Mydocs"

Together, the following two statements change the current directory on the current drive to My files Mailbox:

```
DirName$ = "\Myfiles\Mailbox"
ChDir DirName$
```

- **Tips** To make sure you have set the correct working directory before you issue commands that do not specify full path names, use ChDir.
  - To change the current drive, use ChDrive.
  - ChDir affects the operation of file-related commands like Open and Kill, which use the current working, or default, drive and directory unless you specify a full path name.
- **Example** The following example changes the current directory as the root, then resets the current directory to the original one:

```
Sub Start()
  Dim Msg As String, UsersPath As String
  Super::Start( )
  ' Save the current pathUsersPath = CurDir
   ' Reset to a new path
  ChDir "\"
  Msg = "The current directory has been changed to "
   & CurDir
+
  ShowFactoryStatus ( Msg )
  Msg = "Changing back to the previous current "
     & "directory in five seconds."
+
  ShowFactoryStatus ( Msg )
  Sleep(5)
  ' Change back to user default directory
  Msg = "Now changing back."
  ChDir UsersPath
  Msg = "The current directory has been reset to "
     & UsersPath & "."
+
  ShowFactoryStatus ( Msg )
End Sub
```

See also ChDrive statement CurDir, CurDir\$ functions MkDir statement RmDir statement

# **ChDrive statement**

Changes the current default drive.

Syntax ChDrive <drive letter>

### Parameters <drive letter>

String expression, consisting of a single letter that specifies the drive that is to become the default drive (Windows only). <drive letter> must correspond to the letter of a valid drive, and <drive letter> must be in the range A to <lastdrive>, where <lastdrive> is the maximum drive letter in your computer.

For example, the following statement changes the current working (default) drive to A:

```
ChDrive "A"
```

Together, the following two statements change the current drive to C:

```
DriveSpec$ = "C"
ChDrive DriveSpec$
```

- **Tips** To ensure that you have set the correct working directory before you issue commands that do not specify full path names, use ChDir.
  - To change the current directory, use ChDir.
  - To determine the current drive and directory, use CurDir.
  - ChDir and ChDrive affect the operation of file-related commands like Open and Kill, which use the current working, or default, drive and directory unless you specify a full path name.
  - ChDrive is ignored when the Actuate Basic program is running on UNIX.
- **Example** The following example changes the default directory to a second drive. If it cannot change the directory, it responds with an error message.

```
Sub Start()
  On Error Resume Next
  Dim CurrPath As String, Drive As String
  Dim HasColon As Integer, Msg As String
  Dim NewDrive As String
  Super::Start( )
  ' Get current path
  CurrPath = CurDir
  ' If current drive is invalid Reset error to 0
  If Err = 68 Then
     Drive = "Invalid"
     Err = 0
  Else
     ' Get drive letter
     Drive = Left(CurrPath, 2)
  End If
  Msg = "Your current drive is "& Drive & ". "
  ShowFactoryStatus ( Msg )
```
```
' Set drive to D:
  NewDrive= "D:"
  ' Change drive
  ChDrive NewDrive
  Select Case Err
     ' Device unavailable error
     Case 44
       Msg = "That drive is not available. "
          & "No drive change was made."
+
     ' Disk not ready error
     Case 71
       Msg = "Close the door on your drive and try again"
     ' Illegal function call
     Case 10
       Msg = "Error changing to " & NewDrive
     Case Else
       Msg = "Drive changed to " & UCase$ (NewDrive)
     End Select
  ShowFactoryStatus ( Msg )
End Sub
```

See also ChDir statement CurDir, CurDir\$ functions MkDir statement RmDir statement

### **Chr, Chr\$ functions**

Returns the character that corresponds to the specified character code in the current run-time encoding.

Syntax Chr(<character code>)

Chr\$(<character code>)

#### Parameters <character code>

A numeric expression that represents the character code you want Chr[\$] to translate to a character. The following conditions apply to <character code>:

- If <character code> is not an Integer, Chr[\$] rounds it to the nearest Integer before returning the character. In the United States, <character code> usually refers to ANSI codes.
- <character code> must be between 0 and 65535 or a run-time error occurs.
- <character code> is interpreted according to the current run-time encoding.

For example:

```
Chr$(65) returns A
Chr$(228) returns g on a Greek locale
Chr$(228) returns ä on an English locale
Chr$(Sqr(4) * 45) returns Z
Chr$(90.11) returns Z
```

The following statement prints or displays the copyright symbol (©):

Chr\$(169)

Returns Chr: Variant Chr\$: String

- Chr[\$] returns nonprintable and printable characters.
- Character codes from 0 to 31, inclusive, are the same as the standard nonprintable ASCII codes.
- Character codes 8, 9, 10, and 13 convert, respectively, to backspace, tab, linefeed, and carriage return characters. You cannot see them graphically. Depending on the application, they can affect the visual display of text.
- **Tips** To force a message either to start or to continue to print on a new line, use Chr\$(10), which inserts the linefeed character.
  - To insert quotation marks (") inside a message, use Chr\$(34).
  - To send special control codes to your printer or to another device, use Chr\$ with a print statement directed to that device.
  - To translate from a character to its character code number, use Asc.
  - If the run-time encoding is UCS-2, Chr behaves in the same way as ChrW.

**Example** The following example returns the character corresponding to a generated number. The example displays the character, if it can be displayed, as well.

```
Sub Start( )
   Dim AnsiNum As Integer, Msg As String
   Super::Start( )
   AnsiNum = Rnd * 255
   Msg = "The character that corresponds to " & AnsiNum & " is:"
+   & & " " & Chr$( AnsiNum )
   ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Asc function

AscW function ChrW, ChrW\$ functions

### ChrW, ChrW\$ functions

Translates a UCS-2 code value into a character.

Syntax ChrW(<character code>)

Chrw\$(<character code>)

#### Parameters <character code>

A numeric expression that represents the character code in UCS-2 you want ChrW[\$] to translate to a character. The following conditions apply to <character code>:

- <character code> must be a valid UCS-2 code.
- If <character code> is not an Integer, ChrW[\$] rounds it to the nearest Integer before returning the character. In the United States, <character code> usually refers to ANSI codes.
- <character code> must be between 0 and 65535 or a run-time error occurs.

For example, the following statement returns A:

ChrW\$(65)

The following statement returns  $\gamma$ :

ChrW(947)

Returns ChrW: Variant ChrW\$: String

**Example** The following example returns the character corresponding to a number that is randomly generated. The example also displays the user-selected character, if it can be displayed.

```
Sub Start( )
   Dim AnsiNum As Integer, Msg As String
   Super::Start( )
   AnsiNum = Rnd * 65535
   Msg = "The character that corresponds to " & AnsiNum & " is:"
+   & & " " & ChrW$( AnsiNum )
   ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also AscW function

### **CInt function**

Converts a numeric expression to the Integer data type.

- Syntax CInt(<numeric expression>)
- **Description** CInt is one of nine Actuate Basic data type conversion functions. CInt rounds the fractional part of <numeric expression>, while Fix and Int truncate it. If <numeric expression> lies outside the acceptable range for an Integer, an overflow error occurs. If <numeric expression> is negative, CInt rounds to the next lowest negative number. For example:

CInt(1.5) = 2 CInt(1.4) = 1 CInt(-1.4) = -1 CInt(-1.5) = -2

#### Parameters <numeric expression>

Numeric expression to convert from the Currency, Date, Double, Long, Single, String, or Variant data type to the Integer data type. The following conditions apply to <numeric expression>:

- If <numeric expression> is Null, CInt returns Null.
- If <numeric expression> is a String or Variant, it must be interpretable as a number.
- If <numeric expression> is a String, it is parsed according to the formatting rules of the current run-time locale.

For example, each of the following statements returns 123. In the second statement, the string 123 enclosed in double quotation marks is considered a numeric expression because it can be interpreted as a number. In the third statement, four cannot be interpreted as a number.

```
CInt(123)
CInt("123")
CInt("123four")
```

The following statement first converts the string 123four using Val to convert the number 123. Val ignores the string component (four).

```
CInt(Val("123four"))
```

The following statement returns 123.456 on a French run-time locale and 123456.00 on an English run-time locale:

```
CInt("123,456")
```

**Returns** Integer. If <numeric expression> cannot be interpreted as a number, zero is returned.

- **Tips** To convert an expression to an Integer without using CInt, assign the expression to a variable that is the Integer data type.
  - To declare the data type of a variable, use Dim with the As clause.
  - To round a number to precisely the number of decimal places you want and discard anything residual, use Format\$.
  - To be sure you pass the correct data type to a sub procedure or to a Windows Dynamic Link Library (.dll), use data type conversion functions.
- **Example** The following example generates a random number, converts that number using each of the conversion functions in turn, then displays the result:

```
Sub Start()
  Dim Orig As String, Msg As String
  Super::Start( )
  ' Get a random number between 1 and 256
  Orig = 255 * Rnd + 1
  ' Convert to various formats
  Msg = "Your number is: " & Orig
  ShowFactoryStatus ( Msg )
  Msg = "CCur( Orig ) yields ----> " & CCur( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CDate( Orig ) yields ----> " & CDate( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CDbl( Orig ) yields ----> " & CDbl( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CInt( Orig ) yields ----> " & CInt( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CLng( Orig ) yields ----> " & CLng( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CSng( Orig ) yields ----> " & CSng( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CStr( Orig ) yields ----> " & CStr( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CVar( Orig ) yields ----> " & CVar( Orig )
  ShowFactoryStatus ( Msg )
End Sub
```

See also CCur function CDate function CDbl function CLng function CSng function CStr function CVar function CVDate function Dim statement Format, Format\$ functions Int function

### **Class statement**

Declares and defines a class in Actuate Basic.

Syntax Class <name of class> [Subclass Of <name of superclass>]...[,<name of superclass>...]]

[{DimlStatic}] <variable> [([<subscripts>])] [As <data type>]

[, [{Dim|Static}] <variable> [([<subscripts>])] [As <data type>]]

<nested class(es)>

<method(s)>

End Class

#### Parameters <name of class>

The name you assign to the class. You must supply a class name.

#### <Subclass Of <name of superclass>>

Specifies a superclass from which this class can be derived. Multiple inheritance is supported by specifying more than one superclass.

#### [{Dim|Static]}

These optional keywords define the scope of the variable.

**Dim**—Defines an instance variable. Each instance of the class has a copy of each instance variable.

**Static**—Defines a static variable. All instances of this class and all instances of derived classes share a single copy of the variable.

#### <variable>

Name of the variable belonging to the class. A class can contain any number of variables.

#### <subscripts>

Describes array dimensions according to the following syntax:

[<lower> To] <upper>[,[<lower> To]<upper>]...

If you do not supply subscript values between parentheses, Actuate Basic declares a dynamic array. Range from <lower> to <upper> can be from -2147483648 to 2147483647, inclusive.

#### As <data type>

Clause that declares the data type of <variable>. <data type> can specify any valid Actuate Basic data type, user-defined data type, or class name, for example:

Dim SalesTax Customer As String, Amount As Currency

#### <nested class(es)>

Any number of other class definitions. Actuate automatically creates nested classes for controls within frames, and for sections and frames within the report object.

#### <method(s)>

Procedure that performs actions on the objects of the class. Your class can contain any number of methods. Each method is defined using the Sub or Function statements.

- Tip Class statements can be recursive. Actuate Basic sets a run-time stack limit of 200. A report using recursion with a large number of iterations might exceed this limit.
- **Example** The following example creates the class PerDay that stores a counter and date in each class instance. PerDay also has an overall counter that sums the counts of all instances of the class. As the counts are updated, the local, or instance, counters are independent, while the global, or static, counter increases every time an instance counter is incremented.

To use this example, paste the class definition at or near the beginning of your Actuate Basic source code (.bas) file.

```
Class PerDay
```

```
' PerDay stores a count for a corresponding date
Dim DateOfDay As Date, CountOfDay As Integer
' CountTotal sums CountOfDay for all instances of PerDay
Static CountTotal As Integer
' Initialize sets the starting date and count for the class
' instance and updates the total count.
Sub Initialize (NewDate As Date, NewCount As Integer )
  DateOfDay = NewDate
  CountOfDay = NewCount
  CountTotal = CountTotal + NewCount
End Sub
' IncrCount increments the local and global counts by one.
Sub IncrCount( )
  CountOfDay = CountOfDay + 1
  CountTotal = CountTotal + 1
End Sub
' Display shows the values of this instance of PerDay plus
' the global total
```

Then, paste the following example code in the method editor:

```
Sub Start()
  Dim FirstPerDay As PerDay, SecondPerDay As PerDay
   Dim NewDate As Date
   Super::Start( )
   ' Instantiate two instances of class PerDay.
   Set FirstPerDay = New PerDay
   Set SecondPerDay = New PerDay
   ' Get today's date for the first PerDay instance.
  NewDate = Date
   FirstPerDay.Initialize( NewDate, 1 )
   ' Both the instance and global counts are 1.
   FirstPerDay.Display( )
   ' Use last week's date for the second PerDay instance.
   NewDate = NewDate - 7
   SecondPerDay.Initialize( NewDate, 1 )
   SecondPerDay.IncrCount( )
   ' The second instance count is 2. The global count is 3.
   SecondPerDay.Display( )
   ' The first instance count is still 1. The global count is 3.
   FirstPerDay.Display( )
End Sub
Dim statement
```

```
See also Dim statement
Function...End Function statement
Option Base statement
Static statement
Sub...End Sub statement
```

### **ClearClipboard function**

Clears the contents of the operating environment Clipboard.

Syntax ClearClipboard

Returns Integer

- Returns 1 (True) if the Clipboard was successfully cleared.
- Returns 0 (False) if the Clipboard could not be cleared.
- **Example** The following example places the current date on the Clipboard, then displays the contents of the Clipboard:

```
Sub Start()
Dim Msg As String
Super::Start()
On Error Resume Next
Msg = "The Clipboard contains: " & GetClipboardText
ShowFactoryStatus( Msg )
Msg = "Placing today's date on the clipboard."
ShowFactoryStatus( Msg )
ClearClipboard
SetClipboardText(Format$(Date, "dddd, mm/dd/yyyy"))
Msg = GetClipboardText
ShowFactoryStatus( "The Clipboard now contains: " & Msg )
End Function
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also GetClipboardText function SetClipboardText function

## **CLng function**

Converts a numeric expression to the Long data type.

Syntax CLng(<numeric expression>)

**Description** CLng is one of nine Actuate Basic data type conversion functions. If <numeric expression> lies outside the acceptable range for a Long data type, an overflow error occurs. If <numeric expression> is Null, CLng returns Null.

#### Parameters <numeric expression>

Numeric expression to convert from the Currency, Date, Double, Integer, Single, String, or Variant data type to the Long (Long integer) data type. The following conditions apply to <numeric expression>:

- If String or Variant, <numeric expression> must be interpretable as a number.
- If <number expression> is a String, it is parsed according to the formatting rules of the current run-time locale.

For example, each of the following statements returns the value 123. In the second statement, the string 123 enclosed in double quotation marks is considered a numeric expression because it can be interpreted as a number.

```
CLng(123)
CLng("123")
```

The following statement generates an error. The string 123four is not considered a numeric expression because four cannot be interpreted as a number.

```
CLng("123four")
```

The following statement first converts the string 123four using Val to convert the number 123. Val ignores the string component (four).

```
CLng(Val("123four"))
```

The following statement returns 123.456 on a French run-time locale and 123456.00 on an English run-time locale:

CLng("123,456")

- **Returns** Long. If <numeric expression> cannot be interpreted as a number, zero is returned.
  - **Tips** To convert an expression to long without using CLng, assign the expression to a variable that is the Long data type.
    - To declare the data type of a variable, use Dim with the As clause.
    - To round a number to precisely the number of decimal places you want and discard anything residual, use Format\$.
    - To be sure you pass the correct data type to a sub procedure or to a Windows Dynamic Link Library (.dll), use data type conversion functions.
- **Example** The following example generates a random number, converts that number using each of the conversion functions in turn, then displays the result:

```
Sub Start( )
Dim Orig As String, Msg As String
Super::Start( )
' Get a random number between 1 and 256
Orig = 255 * Rnd + 1
' Convert to various formats
Msg = "Your number is: " & Orig
ShowFactoryStatus ( Msg )
Msg = "CCur( Orig ) yields ----> " & CCur( Orig )
ShowFactoryStatus ( Msg )
Msg = "CDate( Orig ) yields ----> " & CDate( Orig )
ShowFactoryStatus ( Msg )
Msg = "CDbl( Orig ) yields ----> " & CDbl( Orig )
ShowFactoryStatus ( Msg )
```

```
Msg = "CInt( Orig ) yields ----> " & CInt( Orig )
ShowFactoryStatus ( Msg )
Msg = "CLng( Orig ) yields ----> " & CLng( Orig )
ShowFactoryStatus ( Msg )
Msg = "CSng( Orig ) yields ----> " & CSng( Orig )
ShowFactoryStatus ( Msg )
Msg = "CStr( Orig ) yields ----> " & CStr( Orig )
ShowFactoryStatus ( Msg )
Msg = "CVar( Orig ) yields ----> " & CVar( Orig )
ShowFactoryStatus ( Msg )
End Sub
```

See also CCur function CDate function CDbl function CInt function CSng function CStr function CVar function CVDate function Dim statement Format, Format\$ functions

### **Close statement**

Concludes input/output to a file.

Syntax Close [ [#] <open file number> ] ... [, [#] <open file number> ] ...

- **Description** If you use Close for a file that was opened for Output or Append, Actuate Basic flushes the buffers associated with the file and writes any remaining data in them to disk.
  - The association of a file with <open file number> ends when Close is executed. You can then reopen the file using the same or a different file number, or you can reuse the file number to open a different file.
  - Once a file has been closed, the file number cannot be referenced by any other file-related statements (such as Get or Put) until you issue another Open with that file number.
  - If <open file number> does not refer to a currently open file, Actuate Basic ignores the Close statement and continues with the next command.

- Close with no arguments closes all open files.
- Actuate Basic automatically closes all open files at program termination, whether or not you use Close.

#### Parameters <open file number>

Numeric expression for the number you used in a previously issued Open statement that opened the target file to be closed.

Default: All open files are closed.

For example, the following statement closes the file previously opened as #1:

Close #1

The following statement closes the files opened as 1, 5, and 7:

Close 1, 5, 7

The following statement closes all open files:

Close

- **Tip** You should close each open file before ending your program.
- **Example** The following example generates three test files by opening the test files, writing to the test files, and closing the test files. The example overrides Start to call a subroutine that generates the three test files. It then deletes the files. To use this example, paste the procedure Make3Files after the End Sub of the Start procedure or save it in your Actuate Basic source code (.bas) file.

```
Sub Start()
  Dim Msg As String
  Super::Start( )
  ' Create test files
  Make3Files
  Msg = "Three test files have been created on your disk at "
     & CurDir$ & ". They will now be deleted."
+
  ShowFactoryStatus ( Msg )
  ' Remove files from disk.
  Kill "TESTFIL?"
End Function
' The following procedure creates the sample data files.
Sub Make3Files()
  Dim I As Integer, FreeNum As Integer, TestFileName As String
  ShowFactoryStatus( "We are now in the Make3Files sub." )
  For I = 1 to 3
     ' Get next free file handle
     FreeNum = FreeFile
     TestFileName = "TESTFIL" & FreeNum
     ' Opening the file automatically creates it
     Open TestFileName For Output As FreeNum
```

```
' Print one line of text in the file
Print #FreeNum, "This is a test file."
Next I
' Close all files
Close
End Sub
```

See also Open statement Reset statement Stop statement

### **Command, Command\$ functions**

Returns the argument component of the command line of erdpro.exe.

Syntax Command

Command\$

**Description** In the following example, commandargs represents what Command returns if you launched Actuate Basic with the following command:

erdpro.exe /CMD commandargs

The function returns commandargs Secondcomm 32, the launch argument—assuming you launched the application with the following command:

erdpro commandargs Secondcomm 32

Actuate Basic Compiler does not generate an EXE. Instead, it generates a report object executable (.rox) file, which runs either in the ROX compiler, which is built into e.Report Designer Professional, or through iServer.

- Returns Command: Variant Command\$: String
- **Example** The following example displays the command-line arguments of erdpro.exe that you are currently using:

```
Msg = "The command-line string you passed to this "
+ & "program when you launched it is " & Command
End If
ShowFactoryStatus( Msg )
End Sub
```

### **Const statement**

Declares symbolic constants to be used in place of literal values.

Syntax [Global] Const<const name>=<expression> [,<const name>=<expression>]...

**Description** The following conditions apply when using symbolic constants:

- Const must be in a Declare section.
- Any <const name> you use must be defined by Const before you can refer to it.
- A constant cannot be a Variant, and its data type cannot change.
- Unlike a variable, a constant cannot be changed while your program is running.
- You can use a constant anywhere you might use an expression.
- The data type of <const name> is based on <expression> if there is no type declaration postfix specified. A string <expression> always yields a String <const name>, but a numeric <expression> yields the simplest data type that can represent it once it has been evaluated.
- A constant declared in a sub or function procedure is local to that procedure.

#### Parameters [Global]

Keyword indicating that the <const name>s can be accessed by all procedures in the program.

#### <const name>

Name of the constant to which the value of <expression> is assigned. The name must follow standard variable naming conventions.

#### <expression>

Expression assigned to <const name>. <expression> can consist of literals, other constants, or any arithmetic or logical operators. The expression cannot contain or use any of the following:

- Variables
- User-defined functions

- Intrinsic Actuate Basic functions such as Chr[\$]
- **Tips** To simplify maintenance and debugging, put Global Const definitions in a single module.
  - To make constants easier to recognize when debugging, use all uppercase letters for your constant names.
  - To declare the data type of a constant, use a type declaration postfix character like %, &, !, #, @, or \$.
- **Example** The following example defines the constant PI, generates the value of a radius, then uses PI to compute the circumference and area of a circle. To use this example, paste the following code at or near the beginning of your Actuate Basic source code (.bas) file:

```
Declare

' Define constantGlobal Const PI = 3.141592654

End Declare
```

Then, paste the following function into the method editor:

```
Sub Start()
  Dim Area As Double, Circum As Double
  Dim Msg As String, Radius As Integer
  Super::Start( )
  ' The radius of a circle in centimeters
  Radius = Rnd * 10
  ' Compute circumference
  Circum = 2 * PI * Radius
  ' Compute area
  Area = PI * (Radius ^ 2)
  Msq = "If the radius of a circle is " & Radius & " cm"
  ShowFactoryStatus ( Msg )
  Msg = "the circumference of the circle is " & Circum & " cm."
  ShowFactoryStatus ( Msg )
  Msg = "Its area is " & Area & " sq cm."
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Dim statement Global statement Let statement Static statement

## ConvertBFileToString function

Encodes a binary data file into a string.

- Syntax ConvertBFileToString(<filename>)
- **Description** Use ConvertStringToBFile to conditionally embed different images in your report instances—based upon different data or run-time parameters.

Use ConvertBFileToString, and the complementary function ConvertStringToBFile, to send the same report design to two different customers, A and B, and include binary data A in the report to A, and binary data B in the report to B. Instead of embedding either set of binary data in a given report, or creating two slightly different reports, you write conditional code in a single report. The instances of your report display different data at run time as required.

ConvertBFileToString should be used only in conjunction with its complementary function, ConvertStringToBFile.

 Parameters
 <filename>

 String expression that specifies the binary file to convert. Can include optional drive and path information. The default path is the current default drive and directory. <filename> cannot include wildcard characters and must refer to an existing file.

 Returns
 String. If <filename> is not found or the file cannot be read, a run-time error occurs.

See also ConvertStringToBFile function

## ConvertStringToBFile function

Decodes a string from Actuate's internal representation back into a binary file.

Syntax ConvertStringToBFile(<stringval>)

**Description** Use ConvertStringToBFile to conditionally embed different sets of binary data in your report instances—based upon different data or run-time parameters.

Use ConvertStringToBFile, and the complementary function ConvertBFileToString, to send the same report design to two different customers, A and B, and include binary data A in the report to A, and binary data B in the report to B. Instead of embedding either set of binary data in a given report, or creating two slightly different reports, you write conditional code in a single report. The instances of your report display different sets of binary data at run time as required.

 ConvertStringToBFile should be used only in conjunction with its complementary function, ConvertBFileToString.

	<ul> <li>The parameter you pass in (<stringval>) must be the return from the complementary function ConvertBFileToString.</stringval></li> </ul>		
	<ul> <li>The returned file name is the name of the temporary file. Remove the temporary file when it is no longer needed.</li> </ul>		
Parameters	<stringval> String expression that specifies the string to convert into a binary file.</stringval>		
Returns	String		
Тір	• ConvertStringToBFile creates a temporary file. Be sure to include code that clean up the temporary file after it is no longer needed.		
See also	ConvertBFileToString function		

### **ConvertToXML** function

Returns a string with escaped characters for those characters that have special meaning to XML.

Syntax ConvertToXML(<convert>, <XMLcharset>)

#### Parameters <convert>

String expression. Contains the string you wish to modify. Certain characters have special meaning to XML. ConvertToXML puts escape characters in the string in place of the special character so that the string itself is properly formatted for use in XML. For example, the & character has special meaning to XML syntax, and including it in a string to be used in XML can cause unpredictable results. Calling ConvertToXML replaces the ampersand (&) character with the string & amp;. The string that contained A & B is now returned as A & amp; B.

#### <XMLcharset>

String expression. Contains the encoding value to use in conversion. If set to " ", no encoding is used.

Returns String

#### **Example** The following example demonstrates the return value of ConvertToXML:

```
Sub Start( )
Dim data as String
Dim Msg as String
Super::Start( )
data = "A & B"
Msg = "The result of ConvertToXML(data) is " &
ConvertToXML(data)
ShowFactoryStatus( Msg )
End Sub
```

### **CopyInstance statement**

Copies the values of the variables from one instance to another instance, even when the classes of the two instances are unrelated.

**Syntax** CopyInstance(<source>, <destination>, <excludeProperty>)

Both the names and data types of the variables in <source> must match the names and data types of the corresponding variables in <destination>.

#### Parameters <source>

Any instance (object). The instance that contains the variables from which you want to copy values.

#### <destination>

Any instance (object). The instance that contains the variables to which you want to copy the values from <source>.

The classes of the two instances do not need to be related.

CopyInstance matches variables on name and data type.

For example, in the case of a variable in <source> named Amount with type Currency, CopyInstance copies the variable's value from <source> to <destination> only if <destination> has a variable with the same name and type.

CopyInstance ignores or skips variables that do not match on both name and data type.

#### <excludeProperty>

String that specifies the properties to be excluded when copying an instance.

- **Tip** CopyInstance is useful when you work with multi-input filters. CopyInstance simplifies writing complex data streams, for example, when creating a union.
- **Example** The following example creates the classes Size and Quantity. Both classes contain a variable named Title of type String, but no other similar variables.

To use this example, paste the class definitions at or near the beginning of your Actuate Basic source code (.bas) file.

```
Class Size

Dim Height As Single, Title As String

Dim Msg As String

Sub Initialize(NewHeight As Single, NewTitle As String)

Height = NewHeight

Title = NewTitle

End Sub
```

```
Sub Display( )
     Msg = "Title is " & Title & " and Height is " & Height
     ShowFactoryStatus ( Msg )
  End Sub
End Class
Class Quantity
  Dim Quantity As Integer, Title As String
  Dim Msg As String
  Sub Initialize (NewQuantity As Integer, NewTitle As String)
     Quantity = NewQuantity
     Title = NewTitle
  End Sub
  Sub Display( )
     Msg = "Title is " & Title & ". Quantity is " & Quantity
     ShowFactoryStatus ( Msg )
  End Sub
End Class
```

Then, paste the following example code into the method editor:

```
Sub Start( )
Dim SizeObj As Size, QuanObj As Quantity
Super::Start( )
Set SizeObj = New Size
Set QuanObj = New Quantity
SizeObj.Initialize(5.67, "A size")
SizeObj.Display
QuanObj.Display
CopyInstance(SizeObj, QuanObj, "")
QuanObj.Display
End Sub
```

Only SizeObj is initialized. After CopyInstance is called, only the Title variable in QuanObj has a value. The remaining QuanObj variable, Quantity, is not set because it has no match in SizeObj.

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also SetBinding function

### **Cos function**

Returns the cosine of an angle.

Syntax Cos(<angle>)

#### CreateJavaClassHandle function

#### Parameters <angle>

Number, numeric expression, or Variant of VarType 8 (String) that specifies, in radians, the angle for which you want to find the cosine.

If <angle> is a String, it is parsed according to the formatting rules of the current run-time locale. If <angle> evaluates to Null, Cos returns Null.

- Returns Double
  - **Tip** To convert between radians and degrees, use: radians = degrees \* Pi/180.
- **Example** The following example prompts the user for an angle expressed in radians, then returns the cosine of the angle:

```
Sub Start( )
Dim Angle As Double, Pi As Double
Dim Msg As String
Super::Start( )
Pi = 3.14159265358979
' A random angle in radians
Angle = Pi * Rnd
Msg = "The cosine of " & Angle & " is: " & Cos( Angle )
ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapterUsing the code examples.

See also Acos function Asin function Atn function Sin function Tan function

### CreateJavaClassHandle function

Creates a handle to a Java class.

- Syntax CreateJavaClassHandle("<class identifier>")
- **Description** Use CreateJavaClassHandle to access static class fields and static class methods. You can also use CreateJavaClassHandle to create an instance of the class by invoking a class constructor method via the class handle.

CreateJavaClassHandle throws a user error for the following problems:

- CLASSPATH environment variable not found
- Java Virtual Machine not found or cannot be created
- Class not found

Method or field not found

#### Parameters <class identifier>

java.lang.String expression that specifies the name of the Java class. The Java class must be located in a path defined in the CLASSPATH environment variable. If the class identifier contains a package name and a class name, only the class name is used in invoking the class constructor.

- **Returns** A Basic object which is a handle to the Java class object if the method succeeded, and an undefined handle if the call failed.
- **Example** On the first line of the following example, the variable theClassHandle is declared as a type Object. On the second line, CreateJavaClassHandle is used to create the handle to the Java class java.lang.String and assign it to theClassHandle.

```
Dim theClassHandle As Object
Set theClassHandle = CreateJavaClassHandle("java.lang.String")
```

The following example shows how to create a Java class handle and how to use that handle to invoke a constructor to create an instance of the class:

```
Dim theClassHandle As Object
Set theClassHandle = CreateJavaClassHandle("java.lang.String")
Dim theInstanceHandle As Object
Set theInstanceHandle = theClassHandle.String( "Hello")
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

### CreateJavaObject function

Creates an instance of a class without invoking a constructor.

**Syntax** CreateJavaObject("<class identifier>")

**Description** CreateJavaObject allocates an object of the given ClassIdentifier. The call does not invoke any constructor, including default constructor, of the class. Instance fields of the class are not initialized in the allocated object. To initialize the instance, either call an initialization method on the instance or invoke the constructor through a handle to the class.

CreateJavaObject throws a user error for the following problems:

- CLASSPATH environment variable not found
- Java Virtual Machine not found or cannot be created
- Class not found
- Object creation failed

CreateJavaObject cannot receive a Null instance.

#### CSng function

Parameters	<class identifier=""> String expression that specifies the name of the Java class. The Java class must be located in a path defined in the CLASSPATH environment variable. The class identifier must be either the class name or the class name and package name. The class identifier cannot contain a partial path.</class>	
Returns	A Basic object which is a handle to the instance if the method succeeded, and an undefined handle if the call failed.	
Example	On the first line of the following example, the variable theObject is dimensioned as a type Object. On the second line, CreateJavaObject is used to create the Java object, readfile, and assign it to theObject.	
	Dim theObject As Object Set theObject = CreateJavaObject("readfile")	

For information about using the code examples, see "Using the code examples," earlier in this chapter.

### **CSng function**

Converts a numeric expression to the Single data type.

Syntax CSng(<numeric expression>)

**Description** CSng is one of nine Actuate Basic data type conversion functions. If <numeric expression> lies outside the acceptable range for a Single data type, an overflow error occurs. If <numeric expression> is Null, CSng returns Null.

#### Parameters <numeric expression>

Numeric expression to convert from the Currency, Date, Double, Integer, Long, String, or Variant data type to the Single (or single precision floating point) data type. The following conditions apply to <numeric expression>:

- If <numeric expression> is a String or Variant, it must be interpretable as a number.
- If <numeric expression> is a String, it is parsed according to the formatting rules of the current run-time locale.

For example, each of the following statements returns 123. In the second statement, the string 123 enclosed in double quotation marks is considered a numeric expression because it can be interpreted as a number.

```
CSng(123)
CSng("123")
```

The following statement generates an error. The string 123four is not considered a numeric expression because four cannot be interpreted as a number.

```
CSng("123four")
```

The following statement first converts the string 123four using Val to convert the number 123. Val ignores the string component, four.

```
CSng(Val("123four"))
```

The following statement returns 123.456 on a French run-time locale and 123456.00 on an English run-time locale:

CSng("123,456")

- **Returns** Single. If <numeric expression> cannot be interpreted as a number, zero is returned.
  - **Tips** To convert an expression to single without using CSng, assign the expression to a variable that is the Single data type.
    - To declare the data type of a variable, use Dim with the As clause.
    - To round a number to precisely the number of decimal places you want and discard anything residual, use Format[\$].
    - To be sure you pass the correct data type to a sub procedure or to a Windows Dynamic Link Library (.dll), use data type conversion functions.
- **Example** The following example generates a random number, converts that number using each of the conversion functions in turn, then displays the result:

```
Sub Start()
  Dim Orig As String, Msg As String
  Super::Start( )
  ' Get a random number between 1 and 256
  Orig = 255 * Rnd + 1
  ' Convert to various formats
  Msg = "Your number is: " & Orig
  ShowFactoryStatus ( Msg )
  Msg = "CCur( Orig ) yields ----> " & CCur( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CDate( Orig ) yields ----> " & CDate( Orig )
  ShowFactoryStatus ( Msq )
  Msg = "CDbl( Orig ) yields ----> " & CDbl( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CInt( Orig ) yields ----> " & CInt( Orig )
  ShowFactoryStatus ( Msq )
  Msg = "CLng( Orig ) yields ----> " & CLng( Orig )
  ShowFactoryStatus ( Msq )
  Msg = "CSng( Orig ) yields ----> " & CSng( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CStr( Orig ) yields ----> " & CStr( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CVar( Orig ) yields ----> " & CVar( Orig )
  ShowFactoryStatus ( Msg )
End Sub
```

See also CCur function CDate function CDbl function CInt function CLng function CStr function CVar function CVDate function Dim statement Format, Format\$ functions

### **CStr function**

Converts an expression to the String data type.

Syntax CStr(<valid expression>)

**Description** CStr is one of nine Actuate Basic data type conversion functions.

#### Parameters <valid expression>

Expression to convert from the Currency, Date, Double, Integer, Long, Single, or Variant data type to the String data type. <valid expression> is parsed according to the formatting rules of the current run-time locale.

For example, the following statement returns the string 123four:

```
CStr(123) & "four"
```

The following statement generates an error, because it mixes string and numeric data types:

CStr(123) + 5

Returns String

If <valid expression> is Null, CStr returns Null.

- **Tips •** To declare the data type of a variable, use Dim with the As clause.
  - To ensure that you pass the correct data type to a sub procedure or a Windows Dynamic Link Library (.dll), use data type conversion functions.
- **Example** The following example generates a random number, converts that number using each of the conversion functions in turn, then displays the result:

```
Sub Start( )
Dim Orig As String, Msg As String
```

```
Super::Start( )
  ' Get a random number between 1 and 256
  Orig = 255 * Rnd + 1
  ' Convert to various formats
  Msg = "Your number is: " & Orig
  ShowFactoryStatus ( Msq )
  Msg = "CCur( Orig ) yields ----> " & CCur( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CDate( Orig ) yields ----> " & CDate( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CDbl( Orig ) yields ----> " & CDbl( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CInt( Orig ) yields ----> " & CInt( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CLng( Orig ) yields ----> " & CLng( Orig )
  ShowFactoryStatus ( Msq )
  Msq = "CSng( Orig ) yields ----> " & CSng( Orig )
  ShowFactoryStatus ( Msq )
  Msg = "CStr( Orig ) yields ----> " & CStr( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CVar( Orig ) yields ----> " & CVar( Orig )
  ShowFactoryStatus ( Msg )
End Sub
```

See also CCur function CDate function CDbl function CInt function CLng function CSng function CVar function CVDate function Dim statement Format, Format\$ functions

### **CurDir, CurDir\$ functions**

Returns the current default directory for the specified drive, or for the current drive.

Syntax CurDir[(<drive:>)]

CurDir\$[(<drive:>)]

**Description** While DOS always maintains a distinct current directory for each drive in the system, UNIX keeps only the current directory under the current drive. The current directory is the one the system searches first for a file name whenever you start such a search without specifying a path name.

#### Parameters <drive:>

String expression, consisting of a single character followed by a colon that specifies the drive for which you want to determine the current directory. The default is the current drive. The following conditions apply to <drive:>:

- First character of <drive:> must correspond to the letter of a valid DOS drive.
- First character of <drive:> must be in the range A to <lastdrive>, where <lastdrive> is the maximum drive letter you set in your Config.sys file.

For example, the following statement assigns to the variable DefaultVar a string containing the current directory path on the default drive:

DefaultVar = CurDir

The following statement assigns to the variable B\$ a string containing the current directory path on drive A:

B\$ = CurDir("A")

The following statement assigns to the variable PlaceMark a string containing the current directory path on drive C:

```
PlaceMark = CurDir$("C:")
```

- Returns CurDir: Variant CurDir\$: String
  - **Tips** Since the path CurDir[\$] returns includes the drive letter, it also determines which drive is currently the current drive.
    - To be sure you have set the correct working directory before you issue commands that do not specify full path names, use CurDir[\$], ChDrive, and ChDir.
    - <drive> is ignored when the Basic program is running on a UNIX server.
- **Example** The following example displays a message box indicating the current directory on the current drive:

```
Sub Start( )
Dim Msg As String
Super::Start( )
Msg = "The current directory is: "
' Get current directory
Msg = Msg & CurDir
ShowFactoryStatus( Msg )
End Sub
```

See also ChDir statement ChDrive statement MkDir statement RmDir statement

### **CVar function**

Converts a numeric expression to the Variant data type.

Syntax CVar(<valid expression>)

**Description** CVar is one of nine data type conversion functions. If <any valid expression> lies outside the acceptable range for a Variant data type, an overflow error occurs.

#### Parameters <valid expression>

Expression to convert to the Variant data type from the Currency, Date, Double, Integer, Long, Single, or String data type.

For example, the following statements return True, nonzero:

CVar(123) & "four" = "123four" CVar(123) + 5 = 128 CVar(123) + 5 & "append" = "128append"

#### Returns Variant

If <valid expression> is Null, CVar returns Null.

- **Tips** To convert an expression to variant without explicitly using CVar, assign the expression to a variable that is the Variant data type.
  - To declare the data type of a variable, use Dim with the As clause.
  - To be sure you pass the correct data type to a Sub procedure or to a Windows Dynamic Link Library (.dll), use data type conversion functions.

# **Example** The following example generates a random number, converts that number using each of the conversion functions in turn, then displays the result:

```
Sub Start( )
Dim Orig As String, Msg As String
Super::Start( )
' Get a random number between 1 and 256
Orig = 255 * Rnd + 1
' Convert to various formats
Msg = "Your number is: " & Orig
ShowFactoryStatus ( Msg )
Msg = "CCur( Orig ) yields ----> " & CCur( Orig )
```

```
ShowFactoryStatus ( Msg )
  Msg = "CDate( Orig ) yields ----> " & CDate( Orig )
  ShowFactoryStatus ( Msq )
  Msg = "CDbl( Orig ) yields ----> " & CDbl( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CInt( Orig ) yields ----> " & CInt( Orig )
  ShowFactoryStatus ( Msq )
  Msg = "CLng( Orig ) yields ----> " & CLng( Orig )
  ShowFactoryStatus ( Msq )
  Msq = "CSnq( Orig ) yields ----> " & CSng( Orig )
  ShowFactoryStatus (Msq)
  Msg = "CStr( Orig ) yields ----> " & CStr( Orig )
  ShowFactoryStatus ( Msg )
  Msg = "CVar( Orig ) yields ----> " & CVar( Orig )
  ShowFactoryStatus ( Msg )
End Sub
```

See also CCur function CDate function CDbl function CInt function CLng function CSng function CStr function CVDate function Dim statement Format, Format\$ functions

### **CVDate function**

Converts an expression to a Variant of VarType 7 (Date).

Syntax CVDate(<date expression>)

#### Parameters <date expression>

Date expression, or any numeric or string expression that can be interpreted as a date, a time, or both a date and a time. The default date if none is specified is 0 (Dec. 30, 1899) and the default time if none is specified is 00:00:00 (midnight).

The following conditions apply to <date expression>:

Can be a String such as November 12, 1982 8:30 P.M., 12 Nov., 1982 08:30 P.M., 11/12/82, 08:30 P.M., or any other String that can be interpreted as a date, a time, or both a date and a time in the valid range.

- Can be a date serial number such as 30267.854, which corresponds to November 12, 1982 at 8:30 P.M., or any other number that can be mapped to a date, a time, or both a date and a time in the valid range.
- For date serial numbers, the Integer component represents the date itself while the decimal component represents the time of day on that date, where January 1, 1900 at precisely noon has the date serial number 2.5, and negative numbers represent dates prior to December 30, 1899 (0).
- If <date expression> includes a time of day, it must be a valid time, even if CVDate is not being used to return anything having to do with a time of day. A valid time is one that is in the range 0:00:00 (12:00:00 A.M.) through 23:59:59 (11:59:59 P.M.). Either the 12- or 24-hour clock can be used.
- If <date expression> is a numeric expression, the numeric expression must be in the range -657434 to +2958465, inclusive.
- <date expression> cannot contain a day of week.
- <date expression> must not be empty or zero-length.
- <date expression> must specify a date within the range January 1, 100 through December 31, 9999, inclusive.
- <date expression> is parsed according to the formatting rules of the current run-time locale.

The following assumes that the AC\_CENTURY\_BREAK property is set to the default value of 30. For information about using AC\_CENTURY\_BREAK, see *Accessing Data using e.Report Designer Professional.* 

- For abridged date expressions ending in 00 through 29 that do not specify the century, relevant date functions estimate the unabridged year by adding 2000. For example, while Year(#4/30/1910#) returns 1910, the abridged expression Year(#4/30/10#) returns 2010 (10 + 2000).
- For abridged date expressions ending in 30 through 99 that do not specify the century, relevant date functions estimate the unabridged year by adding 1900. For example, while Year(#11/12/2082#) returns 2082, the abridged expression Year(#11/12/82#) returns 1982 (82 + 1900).

Note that a date can contain an abridged month, such as Nov. 12, 1982 as opposed to November 12, 1982.

For example, the following statements are equivalent. Each assigns 11/12/82 8:30:00 P.M., with its underlying serial number, to the variable BDate:

```
BDate = CVDate("11/12/82 8:30 pm")
BDate = CVDate("12 Nov 1982 20:30")
BDate = CVDate("November 12, 1982 8:30PM")
BDate = CVDate(30267.854167)
```

#### Returns Date

- If <date expression> is Null, CVDate returns Null.
- If <date expression> cannot be evaluated to a date, CVDate returns Null.
- If <date expression> contains a day of week, CVDate returns Null.
- If <date expression> fails to include all date components, such as day, month, and year, CVDate returns Null.
- If <date string> contains only numbers separated by valid date separators, CVDate recognizes the order for month, day, and year according to the settings of the current run-time locale.
- **Tips** Do not use CVDate to convert a long date format that also contains the day-of-the-week string, like Friday, Nov. 12, 1982. The function does not recognize such strings as dates.
  - Avoid supplying CVDate with a date in a format other than formats specified in the locale map.
  - To determine if either a numeric or string expression that looks like a date can be converted to a date, use IsDate. However, IsDate cannot determine whether a string that looks like a number can be converted to a date, even if the number is in the correct range for date serial numbers.
  - If you use a date expression in one locale, it might be misinterpreted in another locale. For instance, in the United States, 1/2/2005 means January 2nd, 2005, but in France, it means February 1st, 2005. To avoid such ambiguities, use DateSerial to specify all your dates.
- **Example** The following example generates a date. For any valid entry, it displays both a date and a date serial number.

```
Sub Start( )
Dim DateMeme As String, ConvDateMeme As Date
Dim Msg As String
Super::Start( )
DateMeme = CStr( Rnd * 125 * 365 )
ConvDateMeme = CVDate( DateMeme )
Msg = "The date is: " & ConvDateMeme
ShowFactoryStatus( Msg )
Msg = "The serial number for that date is: "
+ & CDbl( ConvDateMeme )
ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also CCur function CDate function CDbl function CInt function CSng function CStr function CVar function Dim statement Format, Format\$ functions IsDate function ParseDate function

### **Date, Date\$ functions**

Returns the current system date.

#### Syntax Date

Date\$

For example, the following statement assigns the current system date to a variable:

```
CurrDate = Date
```

### Returns Date: Variant

Date\$: String

- Date returns a Variant of VarType 7 (Date) that contains a date stored internally as a Double.
- Date\$ returns a date within the range January 1, 1980 through December 31, 2036, inclusive.
- Date\$ returns a 10-character string in the form mm-dd-yyyy, where mm represents the current month (01-12), dd the day (01-31), and yyyy the year (1980-2036). The locale for the report determines the order in which year, month, and day appear. The separator is always a hyphen, '-'. For example, for the Japanese locale, Date\$ returns a string like 2005-10-24, even though the separator character for a Japanese date is '/'.
- **Example** The following example examines the system date and determines how many business days remain in the current week, not counting today:

```
Sub Start( )
Dim CurrentDay As Integer, BusDaysLeft As Integer
Dim Verb As String, DayNoun As String, Msg As String
Super::Start( )
' Weekday returns an Integer
CurrentDay = Weekday(Date)
```

```
BusDaysLeft = 6 - CurrentDay
  If BusDaysLeft < 0 or BusDaysLeft > 5 Then BusDaysLeft = 0
  If BusDaysLeft = 1 Then
     Verb = "is "
     DayNoun = "day "
  Else
     Verb = "are "
     DayNoun = "days "
  End If
  Msg = "Today is " & Format$(Date, "dddd") & "."
  ShowFactoryStatus ( Msg )
  Msq = "There " & Verb & BusDaysLeft & " business " & DayNoun
  ShowFactoryStatus ( Msq )
  Msg = "left in the current week, not counting today."
  ShowFactoryStatus ( Msg )
End Sub
```

See also CVDate function Format, Format\$ functions Now function Time, Time\$ functions

### **DateAdd function**

Returns a date to which a specified time interval has been added.

Syntax DateAdd(<interval code>, <number of intervals>, <date expression>)

#### Parameters <interval code>

String expression that specifies the interval of time, such as quarter, month, or second to add. You can use only one <interval code> at a time.

Table 6-1 lists each valid time period and the <interval code> that corresponds to it.

<interval code=""></interval>	Time period	Description
уууу	Year	Adds a year.
qq	Half a year	Adds half a year.
q	Quarter	Adds a quarter of a year.
m	Month	Adds a month.
у	Day of the year	Adds the day of the year.

<interval code=""></interval>	Time period	Description
d	Day of the month	Adds the day of the month.
W	Day of the week	Adds the day of the week.
WW	Week of the year	Adds the week of the year. The week begins on a Sunday.
www	Custom week of the year.	Adds the week of the year. The date you specify determines the day on which the week begins.
h	Hour	Adds the hour.
n	Minute	Adds a minute.
S	Second	Adds a second.

 Table 6-1
 Time periods and the corresponding interval codes

#### <number of intervals>

Numeric expression that specifies the number of intervals to add. Use a positive number to get future dates and a negative number to get past dates.

#### <date expression>

String expression that specifies the date to add to, or a variant expression that contains a valid date. The following conditions apply to <date expression>:

- <date expression> must specify a date within the range January 1, 100 through December 31, 9999, inclusive.
- <date expression> cannot contain a day of week.
- <date expression> is parsed according to the formatting rules of the current run-time locale.

#### Returns Variant

- If you subtract more years than are in <date expression>, an error occurs.
- If <date expression> cannot be evaluated to a date, DateAdd returns Null.
- If <date expression> fails to include all date components, such as day, month, and year, DateAdd returns Null.
- If <date expression> contains a day of week, DateAdd returns Null.
- If <number of intervals> is not a Long value, DateAdd follows the rules for Fix function and rounds <number of intervals> to the nearest whole number.

The following assumes that the AC\_CENTURY\_BREAK property is set to the default value of 30. For information about using AC\_CENTURY\_BREAK, see *Accessing Data using e.Report Designer Professional.* 

• For abridged date expressions ending in 00 through 29 that do not specify the century, relevant date functions estimate the unabridged year by adding 2000.

For example, while Year (#4/30/1910#) returns 1910, the abridged expression Year (#4/30/10#) returns 2010 (10 + 2000).

For abridged date expressions ending in 30 through 99 that do not specify the century, relevant date functions estimate the unabridged year by adding 1900. For example, while Year(#11/12/2082#) returns 2082, the abridged expression Year(#11/12/82#) returns 1982 (82 + 1900).

Note that a date can contain an abridged month, such as Nov. 12, 1982 as opposed to November 12, 1982.

- **Tips** Use DateAdd to add a specified time interval to a date. For example, you can use DateAdd to calculate a date 30 days from a transaction date to display a Send Late Payment Notice field.
  - To calculate the difference between two dates, use DateDiff.
  - If you use a date expression in one locale, it might be misinterpreted in another locale. For instance, in the United States, 1/2/2005 means January 2nd, 2005, but in France, it means February 1st, 2005. To avoid such ambiguities, use DateSerial to specify all your dates.
- **Example** The following example generates a date, adds 30 days to that date, then displays the result:

```
Sub Start( )
   Dim TransactionDate, Msg
   Super::Start( )
   TransactionDate = CDate(CInt(Rnd * 125 * 365))
   Msg = DateAdd("d", 30, TransactionDate)
+   & " is 30 days after the date " & TransactionDate
   ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also DateDiff function

### **DateDiff function**

Calculates and returns the time difference between two specified dates.

**Syntax** DateDiff(<interval >, <date to subtract>, <date to subtract from>)

Parameters <interval>

String expression that specifies the interval of time, such as quarter, month, or second, to use in calculating the difference between two specified dates. You can use only one <interval> at a time.

Table 6-2 lists valid values for <interval> and describes how DateDiff uses each in calculations.

<interval></interval>	Time period	Description
уууу	Year	DateDiff uses the year portion in calculation.
qq	Half of the year	DateDiff calculates the difference by determining the half of the year into which each date falls, then subtracting the halves.
q	Quarter	DateDiff uses the following quarters: January - March, April - June, July - September, and October - December. DateDiff calculates the difference by determining the quarter into which each date falls, then subtracting the quarters.
m	Month	DateDiff uses the month portion of dates in calculation.
у	Day of the year	DateDiff counts the number of days in the specified years.
d	Day of the month	DateDiff counts the number of days in the specified months.
W	Day of the week	DateDiff counts the number of Mondays.
WW	Week of the year	DateDiff counts the number of Sundays.
www	Custom week of the year	DateDiff counts the number of weeks. The date you specify determines the day on which the week begins.
h	Hour	DateDiff counts the number of hours.
n	Minute	DateDiff counts the number of minutes.
S	Second	DateDiff counts the number of seconds.

 Table 6-2
 Time periods and the corresponding interval expressions

#### <date to subtract>

String expression that specifies the date to subtract, or the name of a variant expression containing the valid date. The following conditions apply to <date to subtract>:

- <date to subtract> must specify a date within the range January 1, 100 through December 31, 9999, inclusive.
- <date to subtract> cannot contain a day of week.
- <date to subtract> is parsed according to the formatting rules of the current run-time locale.

#### <date to subtract from>

String expression that specifies the date to subtract from, or the name of a variant expression containing the valid date. The following conditions apply to <date to subtract from>:

- <date to subtract from> must specify a date within the range January 1, 100 through December 31, 9999, inclusive.
- <date to subtract from> cannot contain a day of week.
- <date to subtract from> is parsed according to the formatting rules of the current run-time locale.

The following assumes that the AC\_CENTURY\_BREAK property is set to the default value of 30. For information about using AC\_CENTURY\_BREAK, see *Accessing Data using e.Report Designer Professional.* 

- For abridged date expressions ending in 00 through 29 that do not specify the century, relevant date functions estimate the unabridged year by adding 2000. For example, while Year(#4/30/1910#) returns 1910, the abridged expression Year(#4/30/10#) returns 2010 (10 + 2000).
- For abridged date expressions ending in 30 through 99 that do not specify the century, relevant date functions estimate the unabridged year by adding 1900. For example, while Year(#11/12/2082#) returns 2082, the abridged expression Year(#11/12/82#) returns 1982 (82 + 1900).

Note that a date can contain an abridged month, such as Nov. 12, 1982 as opposed to November 12, 1982.

#### Returns Variant

- If <date to subtract> refers to a later point in time than <date to subtract from>, DateDiff returns a negative number.
- If <date to subtract> cannot be evaluated to a date, DateDiff returns Null.
- If <date to subtract from> cannot be evaluated to a date, DateDiff returns Null.
- If <date to subtract> fails to include all date components (day, month, and year), DateDiff returns Null.
- If <date to subtract from> fails to include all date components (day, month, and year), DateDiff returns Null.
- If <date to subtract> contains a day of week, DateDiff returns Null.
- If <date to subtract from> contains a day of week, DateDiff returns Null.
- If <interval> is w (weekday), DateDiff returns the number of weeks by counting the number of Mondays between the two dates. If <date to subtract from> falls on a Monday, DateDiff does not include that Monday in the
calculation. In the following example, both specified dates fall on a Monday. The result DateDiff returns is 2.

DateDiff("w", "6/26/95", "7/10/95")

 If <interval> is ww (week), DateDiff returns the number of weeks by counting the number of Sundays between the two dates. If <date to subtract from> falls on a Sunday, DateDiff does not include that Sunday in the calculation. In the following example, both specified dates fall on a Sunday. The result DateDiff returns is 2.

```
DateDiff("ww", "6/25/95", "7/9/95")
```

 DateDiff uses the year portion of dates in calculation. For this reason, the following example returns 1, although the dates are one day apart:

DateDiff ("yyyy", "12/31/95", "1/1/96")

 DateDiff uses the month portion of dates in calculation. For this reason, the following example returns 1, although the dates are one day apart:

DateDiff ("m", "12/31/95", "1/1/96")

 DateDiff calculates the difference by checking which quarters the specified dates fall in. For this reason, the following example returns 1, although the dates are one day apart:

```
DateDiff ("q", "12/31/95", "1/1/96")
```

- The actual display format of date and time is set using the Windows Control Panel.
- Tips Use DateDiff to calculate the difference between two dates to display in a calculated field. For example, you can use DateDiff to calculate the number of days between a pre-sales date and a transaction date to display in a Sales Response field.
  - If you are calculating the number of weeks between two dates, you can use either weekday (w) or week (ww) as the interval. Because each starts the calculations from a different day, the results can be different.
  - To add a specified time interval to a date, use DateAdd.
  - If you use a date expression in one locale, it might be misinterpreted in another locale. For instance, in the United States, 1/2/2005 means January 2nd, 2005, but in France, it means February 1st, 2005. To avoid such ambiguities, use DateSerial to specify all your dates.
- **Example** The following example uses two valid date expressions, PreSalesDate and TransactionDate. DateDiff finds the difference, in days, between PreSalesDate and TransactionDate, then displays the result.

```
Sub Start( )
Dim PreSalesDate As Date, TransDate As Date
Dim DDiff As Integer, Msg As String
```

```
Super::Start()
PreSalesDate = CDate( CInt( Rnd * 125 * 365 ) )
TransDate = DateAdd( "d", CInt( Rnd * 90 ), PreSalesDate )
DDiff = DateDiff( "d", PreSalesDate, TransDate )
Msg = "The pre-sales date, " & PreSalesDate & ", is "
ShowFactoryStatus( Msg )
Msg = DDiff & " days before the transaction date, "
ShowFactoryStatus( Msg )
Msg = TransDate & "."
ShowFactoryStatus( Msg )
End Sub
```

See also DateAdd function DateSerial function

# **DatePart function**

Returns a specified component of a given date.

**Syntax** DatePart(<interval code>, <date exprs>)

# Parameters <interval code>

String expression that specifies the interval of time such as quarter, month, or second you want to determine. You can use only one <interval code> at a time.

Table 6-3 lists valid time periods for <interval code>.

<interval code=""></interval>	Time period	Range of return values
уууу	Year	100 - 9999
qq	Half of the year	1 - 2
q	Quarter	1 - 4
m	Month	1 - 12
у	Day of the year	1 - 366
d	Day of the month	1 - 31
W	Day of the week	1 - 7
WW	Week of the year	1 - 53
www	Custom week of the year. The date you specify determines the day on which the week begins.	1 - 53

 Table 6-3
 Time periods for dates and the corresponding interval codes

<interval code=""></interval>	Time period	Range of return values
h	Hour	0 - 23
n	Minute	0 - 59
S	Second	0 - 59

 Table 6-3
 Time periods for dates and the corresponding interval codes

#### <date exprs>

Date expression, or any numeric or string expression that can be interpreted as a date, a time, or both a date and a time:

- Can be a string expression such as November 12, 1982 8:30 P.M., Nov. 12, 1982 08:30 PM, 11/12/82 8:30pm, 20:30, or any other string that can be interpreted as a date, a time, or both a date and a time in the valid range.
- Can be a date serial number such as 30267.854, which corresponds to November 12, 1982 at 8:30 P.M., or any other number that can be mapped to a date, a time, or both a date and a time in the valid range.
- For date serial numbers, the integer component represents the date itself while the decimal component represents the time of day on that date, where January 1, 1900 at precisely noon has the date serial number 2.5, and negative numbers represent dates prior to December 30, 1899 (0).

Default time of day if none specified: 00:00:00 (midnight) Default date if none specified: December 30, 1899

The following conditions apply to <date exprs>:

- <date exprs> must be in the range January 1, 100 through December 31, 9999, inclusive.
- If <date exprs> is a numeric expression, it must be in the range -657434 to +2958465, inclusive.
- If <date exprs> is a variable containing a date serial number, the variable must be explicitly declared as one of the numeric types.
- If <date exprs> includes a time of day, it must be a valid time, even when you are not using DatePart to return anything having to do with time of day. A valid time is one that is in the range 00:00:00 (12:00:00 A.M.) through 23:59:59 (11:59:59 P.M.). Either the 12- or 24-hour clock format may be used.
- <date exprs> cannot contain a day of week.
- <date exprs> is parsed according to the formatting rules of the current run-time locale.

The following assumes that the AC\_CENTURY\_BREAK property is set to the default value of 30. For information about using AC\_CENTURY\_BREAK, see *Accessing Data using e.Report Designer Professional.* 

- For abridged date expressions ending in 00 through 29 that do not specify the century, relevant date functions estimate the unabridged year by adding 2000. For example, while Year(#4/30/1910#) returns 1910, the abridged expression Year(#4/30/10#) returns 2010 (10 + 2000).
- For abridged date expressions ending in 30 through 99 that do not specify the century, relevant date functions estimate the unabridged year by adding 1900. For example, while Year(#11/12/2082#) returns 2082, the abridged expression Year(#11/12/82#) returns 1982 (82 + 1900).

Note that a date can contain an abridged month, such as Nov. 12, 1982 as opposed to November 12, 1982.

Each of the following examples returns True:

```
Print 11 = DatePart("m", "11/12/82 8:30:01 pm")
Print 12 = DatePart("d", "11/12/82 8:30:01 pm")
Print 316 = DatePart("y", "11/12/82 8:30:01 pm")
```

# Returns Integer

- If <date exprs> is Null, DatePart returns Null.
- If <date exprs> cannot be evaluated to a date, DatePart returns Null.
- If <date exprs> fails to include all date components, such as day, month, and year, DatePart returns Null.
- If <date exprs> contains a day of week, DatePart returns Null.
- If DatePart ("w", <date exprs>) returns 1, the day of the week for <date exprs> is Sunday. If DatePart returns 2, the day of the week is Monday, and so on.
- **Tips** To return more than one interval at a time and to exercise more control over output, use Format instead of DatePart.
  - If you use a date expression in one locale, it might be misinterpreted in another locale. For instance, in the United States, 1/2/2005 means January 2nd, 2005, but in France, it means February 1st, 2005. To avoid such ambiguities, use DateSerial to specify all your dates.
- **Example** The following example generates a date and time, then displays all ten values DatePart can return:

```
Sub Start( )
  Dim UserDate As String, Msg As String
  Dim Ans As Integer
  Super::Start( )
  ' Get a date
```

UserDate = CDate( Rnd \* 125 \* 365 ) Msg = "The date is: " & UserDate ShowFactoryStatus ( Msg ) Msg = "The year is: " & DatePart( "yyyy", UserDate ) ShowFactoryStatus ( Msg ) Msg = "The guarter is: " & DatePart( "g", UserDate ) ShowFactoryStatus ( Msg ) Msg = "The month is: " & DatePart( "m", UserDate ) ShowFactoryStatus ( Msg ) Msg = "The day of the year is: " & DatePart( "y", UserDate ) ShowFactoryStatus ( Msg ) Msg = "The day of the month is: " & DatePart( "d", UserDate ) ShowFactoryStatus ( Msg ) Msg = "The day of the week is: " & DatePart( "w", UserDate ) ShowFactoryStatus ( Msq ) Msq = "The week of the year is: " & DatePart( "ww", UserDate) ShowFactoryStatus ( Msg ) Msg = "The hour is: " & DatePart( "h", UserDate ) ShowFactoryStatus ( Msg ) Msg = "The minute is: " & DatePart("n", UserDate) ShowFactoryStatus ( Msg ) Msg = "The second is: " & DatePart("s", UserDate) ShowFactoryStatus ( Msg ) End Sub

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also DateAdd function DateDiff function Day function Format, Format\$ functions Now function Weekday function Year function

# **DateSerial function**

Returns a date variant based on the specified year, month, day, hour, minute, and second.

**Syntax** DateSerial(<year>, <month>, <day>)

DateSerial(<year>, <month>, <day>, <hour>, <minute>, <second>)

#### Parameters <year>

Numeric expression that specifies the year of a date. Must be an integer greater than zero.

#### <month>

Numeric expression that specifies the month of a date. Must be in the range 1 through 12, inclusive.

#### <day>

Numeric expression that specifies the day of a date. Must be in the range 1 through 31, inclusive.

#### <hour>

Numeric expression that specifies the hour of a day. Must be in the range 0 through 23, inclusive.

#### <minute>

Numeric expression that specifies the minute of a hour. Must be in the range 0 through 59, inclusive.

#### <second>

Numeric expression that specifies the second of a minute. Must be in the range 0 through 59, inclusive.

#### Returns Date

- The value DateSerial returns usually looks like a date but is stored internally as a double-precision number known as a date serial number, which is a number that represents a date and/or time from midnight January 1, 1 through December 31, 9999, inclusive.
- The integer component of any date serial number represents the date, such as day, month, and year itself while the decimal or fractional component represents the time of day on that date as a proportion of a whole day—where January 1, 1900 at precisely noon has the date serial number 2.5, and where negative numbers represent dates prior to December 30, 1899.
- The actual display format of date and time is set using the Windows Control Panel.
- **Tips** DateSerial can return a new date based on calculations done on a given date. In the following example, DateSerial returns a date that is thirty days before November 12, 1982—that is, 10/13/1982—and assigns it to the variable DateNew:

```
DateNew = DateSerial(1982, 11, 12)-30
```

 If you use a date expression in one locale, it might be misinterpreted in another locale. For instance, in the United States, 1/2/2005 means January 2nd, 2005, but in France, it means February 1st, 2005. To avoid such ambiguities, use DateSerial to specify all your dates. **Examples** Each of the following statements stores the underlying date serial number for November 12, 1982 (which is 30267) into the double-precision variable DateVar#:

```
DateVar# = CDbl(DateSerial(82, 11, 12))
DateVar# = DateSerial(1982, 11, 12) * 1
```

The following example generates a date and displays the day of the week for that date:

```
Sub Start()
  Dim Msg As String, UserDate As Date, Verb As String
  Dim ValidFlag As Boolean
  Dim DD As Integer, MM As Integer, YY As Integer
  Dim DayOfWeek As String
  Super::Start( )
  Do
     ' Generate day, month, and year values
     DD = Rnd * 31
     MM = Rnd * 12
     YY = Rnd * 125 + 1899
     ' Assume the worst
     ValidFlag = False
     If MM >= 1 And MM <= 12 Then
       If DD >= 1 And DD <= 31 Then
          If YY >= 1753 And YY <= 2078 Then
             ValidFlag = True
          End If
       End If
     End If
  ' Repeat until input is valid
  Loop While ValidFlag = False
  ' Determine date serial
  UserDate = DateSerial( YY, MM, DD )
  ' Find correct verb
  Select Case UserDate
     ' Now is the date serial for the current moment. We need
     ' only its integer component to compare with what the
     ' generated date to determine if the date was before
     ' today, today, or in the future
     Case Is < Int( Now ): Verb = " was a "
     Case Is > Int( Now ): Verb = " will be a "
     Case Else: Verb = " is a "
  End Select
  ' Returns a day of week
  DayOfWeek = Format( UserDate, "dddd" )
```

```
Msg = UserDate & Verb & DayOfWeek & "."
ShowFactoryStatus( Msg )
End Sub
```

See also DateValue function Day function Month function Now function TimeSerial function TimeValue function Weekday function Year function

# **DateValue function**

Returns a date variant that represents the date of the specified string.

Syntax DateValue(<date string>)

# Parameters <date string>

String expression that specifies a date. Can be any string that can be interpreted as a date. The following conditions apply to <date string>:

- <date string> must be in the range January 1, 100 through December 31, 9999, inclusive.
- If a time is appended to the date it must be a valid time, even though DateValue does not return a time.
- <date string> is parsed according to the formatting rules of the current run-time locale.

The following assumes that the AC\_CENTURY\_BREAK property is set to the default value of 30. For information about using AC\_CENTURY\_BREAK, see *Accessing Data using e.Report Designer Professional.* 

- For abridged date expressions ending in 00 through 29 that do not specify the century, relevant date functions estimate the unabridged year by adding 2000. For example, while Year(#4/30/1910#) returns 1910, the abridged expression Year(#4/30/10#) returns 2010 (10 + 2000).
- For abridged date expressions ending in 30 through 99 that do not specify the century, relevant date functions estimate the unabridged year by adding 1900. For example, while Year(#11/12/2082#) returns 2082, the abridged expression Year(#11/12/82#) returns 1982 (82 + 1900).

Note that a date can contain an abridged month, such as Nov. 12, 1982 as opposed to November 12, 1982.

### Returns Date

- If <date string> contains only numbers separated by valid date separators, DateValue recognizes the order for month, day, and year according to the Short Date setting of the current run-time locale.
- If <date string> cannot be evaluated to a date, DateValue returns Null.
- If <date string> fails to include all date components, such as day, month, and year, DateValue returns Null.
- If a day of week is specified, DateValue returns Null.
- The value DateValue returns usually looks like a date but is stored internally as a double-precision number known as a date serial number—that is, a number that represents a date and/or time from midnight January 1, 100 through December 31, 9999, inclusive.
- The integer component of any date serial number represents the date (day, month, and year) itself while the decimal or fractional component represents the time of day on that date as a proportion of a whole day, where January 1, 1900, at precisely noon has the date serial number 2.5, and negative numbers represent dates prior to December 30, 1899.
- **Tip** If you use a date expression in one locale, it might be misinterpreted in another locale. For instance, in the United States, 1/2/2005 means January 2nd, 2005, but in France, it means February 1st, 2005. To avoid such ambiguities, use DateSerial to specify all your dates.
- **Examples** The following statements are equivalent. If the default Short Date format for the current run-time locale is mm/dd/yy, each stores 11/12/10 into the variable DateVar.

```
DateVar = DateValue("November 12, 10")
DateVar = DateValue("November 12, 2010")
DateVar = DateValue("Nov 12, 2010")
DateVar = DateValue("Nov 7, 2010") + 5
DateVar = DateValue("11/12/10")
DateVar = DateValue("11-12-2010")
```

Each of the following statements stores the underlying date serial number for November 12, 1982 (which is 30267) into the double-precision variable DateVar#:

```
DateVar# = DateValue("Nov 12, 1982")
DateVar# = DateValue("Nov 12, 1982") * 1
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also DateSerial function

Day function Month function Now function TimeSerial function TimeValue function Weekday function Year function

# **Day function**

Returns an integer between 1 and 31, inclusive, that represents the day of the month for a specified date argument.

Syntax Day(<date exprs>)

### Parameters <date exprs>

Date expression, or any numeric or string expression that can evaluate to a date. Specifies a date and/or time:

- Can be a string such as November 12, 1982, Nov. 12, 1982, 11/12/82, 11-12-82, or any other string that can be interpreted as a date in the valid range.
- Can be a date serial number such as 30267.854, which corresponds to November 12, 1982 at 8:30 P.M., or any other number that can be mapped to a date in the valid range.
- For date serial numbers, the integer component represents the date itself while the decimal component represents the time of day on that date, where January 1, 1900 at precisely noon has the date serial number 2.5, and negative numbers represent dates prior to December 30, 1899 (0).

The following conditions apply to <date exprs>:

- If <date exprs> is a string expression, it must specify a date in the range January 1, 100 through December 31, 9999, inclusive.
- <date exprs> is parsed according to the formatting rules of the current run-time locale.
- If <date exprs> is a numeric expression, it must be in the range -657434 to +2958465, inclusive.
- If <date exprs> is a variable containing a date serial number, the variable must be explicitly declared as one of the numeric types.

# Returns Integer

■ If <date exprs> cannot be evaluated to a date, Day returns Null. For example:

```
Day ("This is not a date.") returns Null.
```

 If <date exprs> fails to include all date components, such as day, month, and year, Day returns Null, as shown by the following examples:

Day ("Nov 12, 1982") returns 12, but Day ("Nov 1982") returns Null

■ If <date exprs> is Null, Day returns Null.

The following assumes that the AC\_CENTURY\_BREAK property is set to the default value of 30. For information about using AC\_CENTURY\_BREAK, see *Accessing Data using e.Report Designer Professional.* 

- For abridged date expressions ending in 00 through 29 that do not specify the century, relevant date functions estimate the unabridged year by adding 2000. For example, while Year(#4/30/1910#) returns 1910, the abridged expression Year(#4/30/10#) returns 2010 (10 + 2000).
- For abridged date expressions ending in 30 through 99 that do not specify the century, relevant date functions estimate the unabridged year by adding 1900. For example, while Year(#11/12/2082#) returns 2082, the abridged expression Year(#11/12/82#) returns 1982 (82 + 1900).

Note that a date can contain an abridged month, such as Nov. 12, 1982 as opposed to November 12, 1982.

- **Tip** If you use a date expression in one locale, it might be misinterpreted in another locale. For instance, in the United States, 1/2/2005 means January 2nd, 2005, but in France, it means February 1st, 2005. To avoid such ambiguities, use DateSerial to specify all your dates.
- **Examples** The following statements are equivalent. Each assigns 7 to the variable UserDay.

```
UserDay = Day("6/7/64")
UserDay = Day("June 7, 1964 2:35 PM")
UserDay = Day("Jun 7, 1964")
UserDay = Day(23535)
UserDay = Day(4707*5)
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Date, Date\$ functions Hour function Minute function Month function Now function Second function Weekday function Year function

# **DDB** function

Returns the depreciation of an asset for a given, single period using the doubledeclining balance method.

Syntax DDB(<initial cost>, <salvage value>, <asset lifespan>, <single period>)

**Parameters** All parameters must be positive numbers.

#### <initial cost>

Numeric expression that specifies the initial cost of the asset.

#### <salvage value>

Numeric expression that specifies the value of the asset at the end of its useful life.

#### <asset lifespan>

Numeric expression that specifies the length of the useful life of the asset.

Rule for <asset lifespan>: Must be given in the same units of measure as <single period>. For example, if <single period> represents a month, then <asset lifespan> must be expressed in months.

#### <single period>

Numeric expression that specifies the period for which you want DDB to calculate the depreciation.

Rule for <single period>: Must be given in the same units of measure as <asset lifespan>. For example, if <asset lifespan> is expressed in months, then <single period> must represent a period of one month.

The following example calculates the depreciation for the first year under the double-declining balance method for a new machine purchased at \$1400, with a salvage value of \$200, and a useful life estimated at 10 years. The result (\$280) is assigned to the variable Year1Deprec.

Year1Deprec = DDB(1400, 200, 10, 1)

### Returns Double

Double-declining balance depreciation is an accelerated method of depreciation that results in higher depreciation charges and greater tax savings in the earlier years of the useful life of a fixed asset than are given by the straight-line depreciation method (SLN), where charges are uniform throughout.

The method uses the following formula:

Depreciation over <single period> = ((<initial cost> - total depreciation from prior periods) \* 2) / <asset lifespan>.

**Example** The following example initializes various particulars about an asset, then returns the asset's double-declining balance depreciation for a single period:

```
Sub Start()
  Dim Fmt As String, InitCost As Double, SalvageVal As Double
  Dim MonthLife As Double, LifeSpan As Double
  Dim DepYear As Double
  Dim PeriodDepr As Double, Msg As String, YearMonths
  Super::Start( )
  YearMonths = 12
  ' Define money format
  Fmt = "#, ##0.00"
  ' Set Initial cost to $100,000
  InitCost = 100000
  ' Set salvage value to $10,000
  SalvageVal = 10000
  ' Set useful life to 10 years
  LifeSpan = 10
  ' Set year for depreciation to 5
  DepYear = 5
  PeriodDepr = DDB( InitCost, SalvageVal, LifeSpan, DepYear )
  Msg = "InitCost: " & InitCost
    & " SalvageVal: " & SalvageVal
+
+
   & " LifeSpan: " & LifeSpan
     & " The depreciation for year " & DepYear & " is "
     & Format ( PeriodDepr, Fmt ) & "."
  ShowFactoryStatus ( Msg )
End Sub
```

See also SLN function SYD function

# **Declare statement**

Informs Actuate Basic that you are using, and how you are using, an external procedure in a dynamic link library (.dll). A procedure that returns a value is declared as a Function. A procedure that does not return a value is declared as a Sub.

Syntax Declare Sub <proc> Lib <lib name> [Alias <alias name>] [([<arg list>])]

Declare Function <proc> Lib <lib name> [Alias <alias name> ] [([<arg list>])] [As <function data type>]

**Description** Use Declare to declare external procedures—procedures contained in a DLL. Declare needs to appear at or near the beginning of your Actuate Basic source

code (.bas) file. DLL procedures declared in any module are available to all procedures in all modules.

Empty parentheses indicate that the sub or function procedure has no arguments and that arguments should be checked to ensure that none are passed. In the following example, the sub First takes no arguments:

Declare Sub First Lib "MyLIB"()

If you use arguments in a call to First, an error occurs. The following example code would return an error for a sub First that takes no arguments:

```
Declare Sub First Lib "MyLIB" (arg%)
```

When an argument list appears, the number and type of arguments are checked each time the procedure is called. In the following example, the sub procedure Second takes one Long argument:

```
Declare Sub Second Lib "MyLIB" (X&)
```

If the name of the procedure in the DLL does not conform to Actuate Basic naming conventions, you must use an Alias clause to specify it as it appears in the DLL. From that point on, Actuate Basic uses <alias name>.

# Parameters Sub

Keyword indicating that the procedure does not return a value.

# Function

Keyword indicating that the procedure returns a value and therefore can be used in an expression.

# <proc>

The name of the sub or function you are declaring. Must be the same as its name in the DLL where it is defined. For function procedures, the data type of the procedure determines the data type it returns. Unless you use an As clause, <proc> can include a type-declaration character indicating the data type returned by the procedure.

The following conditions apply to <proc>:

- Must follow standard variable naming conventions.
- Cannot be the name of any other procedure.
- Cannot include a type-declaration character if you use the As clause.

# Lib

Keyword before the name of the DLL, <lib name>, that contains the procedure being declared.

# <lib name>

String literal of the DOS filename of the DLL that contains <proc> or <alias name>. If not in the current directory on the current drive, <lib name> must specify the full path name.

### Alias

Keyword indicating the sub or function as it is named in the DLL. The following conditions apply to Alias:

- Must be used when the DLL procedure name is the same as an Actuate Basic reserved word.
- Must be used when a DLL procedure has the same name as a global variable, constant, or any other procedure in the same scope.
- Must be used if any characters in the DLL procedure name are not allowed in Actuate Basic names.

#### <alias name>

String literal for the alternative name of <proc> as it appears within <lib name>.

#### <arg list>

List of variables representing arguments that are passed to the sub or function procedure when it is called. <arg list> has the following syntax:

[**ByVal**] <arg variable> [ **As** <arg data type>] [,[**ByVal**] <arg variable> [ **As** <arg data type>] ] . . .

Arguments may be passed by value (using the keyword ByVal) or by reference.

#### ByVal

Keyword indicating that the argument is passed by value rather than by reference. Table 6-4 summarizes the behavior of ByVal.

<arg variable=""></arg>	Behavior of ByVal
Numeric expression	Converts <arg variable=""> to the data type indicated by its type-declaration character, if any, or else by its associated <b>As</b> clause.</arg>
	Passes the converted value to the DLL procedure.
Null-terminated string	Passes the address of the string data to the DLL procedure.

Table 6-4 ByVal behavior

Default: String descriptor is sent to the called DLL procedure.

ByVal cannot be used with an array, user-defined type, or class variable.

#### <arg variable>

Any valid Actuate Basic variable name. <arg variable> must conform to standard variable naming conventions. If <arg variable> includes a type-declaration character, you cannot use the As <arg data type> clause.

# As <arg data type>

Clause that declares the data type of <arg variable> as Any, CPointer, Currency, Date, Double, Integer, Long, Single, String, Variant, a user-defined type, or any class. The following conditions apply to As <arg data type>:

- If you use this clause, do not use a type-declaration character as part of <arg variable>.
- Use the Any data type in an As clause only to override type checking for that argument.

The default type isVariant.

# As <function data type>

Clause that declares the data type of the value returned by a function procedure, and not valid with declarations of sub procedures. The argument <function data type> can be Integer, Long, Single, Double, Currency, String (variable-length only), or Variant.

**Example** The following example shows a Declare statement that substitutes a shorter name, WinDir, for the full name of a Windows function that retrieves the Windows directory, GetWindowsDirectory. The function takes two arguments that are declared here by value and returns an integer.

```
Declare Function WinDir Lib "Kernel"
+ Alias "GetWindowsDirectory" (
+ ByVal lpBuffer As String, ByVal nSize As Integer) As Integer
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Call statement

# **Declare...End Declare statement**

Declares global constants, global variables, and user-defined variable types.

Syntax Declare

<declaration statements>

End Declare

```
Parameters <declaration statements>
```

Any number of valid Actuate Basic global variable, global constant, or userdefined data type declaration statements.

**Tips** To declare global constants, global variables, and user-defined variable types in Actuate Basic, you must bookend your declarations with Declare and End Declare, as shown in the example.

- Do not confuse this multi-line Declare... End Declare statement with the single-line Declare statement that supports accessing functions in external dynamic link library files (DLLs).
- You can use as few or as many Declare... End Declare blocks as you need. It is good programming practice, however, to position all such statements in one group at or near the beginning of your code.

```
Example The following code in a user Basic file contains several different types of declarations within a single Declare... End Declare block:
```

```
Declare
  ' Global dynamic array:
  Global CarModelsCount()
   ' Global fixed array with 366 elements:
  Global Birthdays (366)
   ' Global Integer variable:
  Global PrintFlag as Integer
   ' Global constant:
  Global Const PI = 3.14159265359
   ' User-defined (non-intrinsic) variable type:
  Type FullName
     FirstName As String
     MiddleName As String
     Lastname As String
     ExactAge As Double
  End Type
   ' A declaration based on the new non-instrinsic type above.
   ' Like an alias:
  Type PenName As FullName
   ' Two similar user-defined variable types, but based on
   ' intrinsic types this time:
  Type CustomerCount As Integer
  Type LiquidMeasure As Double
   ' An enum declaration
  Enum DatabaseType
     ODBC
     Oracle
     DB2
  End Enum
End Declare
```

See also Const statement

Declare statement Enum...End Enum statement Global statement Type...As statement Type...End Type statement

# **Dim statement**

Declares variables and allocates memory. Dim is an abbreviation for the Basic term dimension.

- Syntax Dim <varname> [([<subscripts>])] [As [{Volatile|Transient|Persistent}] <type>] [, <varname> [([<subscripts>])] [As [{Volatile|Transient|Persistent}] <type>]]...
- **Description** When you dimension variables with Dim, they are initialized as shown in Table 6-5.

Туре	Initialized as
Numeric	0
Variant	Empty
Variable-length strings	Zero-length strings
CPointer	Null
User-defined	Separate variables
Object reference	Nothing (before <b>Set</b> )

 Table 6-5
 Initialization of variables with Dim

To declare variables that are available to all procedures within the module, use Global. To declare variables that are available only in the procedure, use Dim at the procedure level. You cannot redimension an array variable with ReDim if you explicitly specified its size using Dim.

# Parameters <varname>

A name you create for the new variable.

#### <subscripts>

Describes array dimensions according to the following syntax:

[<lower> To] <upper>[,[<lower> To]<upper>]...

Range from <lower> to <upper> can be from -2147483648 to 2147483647, inclusive. If you do not supply subscript values between parentheses, Actuate Basic declares a dynamic array.

The following statements are equivalent if you do not use Option Base:

Dim P(7,5) Dim P(0 TO 7, 0 TO 5) Dim P(7, 0 TO 5)

The following example declares a dynamic array:

Dim TestArray()

### [{VolatilelTransient|Persistent}]

These keywords are optional. When omitted, the variable is Persistent by default.

You probably do not want to use these keywords in your own code, and Actuate does not support their use in the Variables dialog box. They are documented here to help you understand AFC source code more thoroughly.

**Persistent**—The variable is to be written to the report object instance (.roi) file subject to standard ROI compression.

**Transient**—The variable is used only in the Factory and is not written to the ROI. When the object is read from the ROI, Transient variables are set to their default values. Marking variables as Transient results in a smaller ROI and improves object locality.

**Volatile**—Like a Persistent variable, the variable is written to the ROI. However, unlike Persistent, Volatile variables are not subject to ROI compression. Marking variables as Volatile improves object locality for objects with variables that change frequently during a report run, at the cost of taking additional space in the ROI.

#### As <type>

Specifies a data type or class for the variable. If you specify a class, <varname> can hold a reference to an instance of that class or descendant classes. Otherwise, if you do not specify As <type>, <varname> is of type Variant.

- **Tips** To avoid assigning incorrect variable types, use the As clause to declare variable types.
  - Use the Option Strict statement to enforce variable typing.
  - To declare dynamic arrays, use Dim with empty parentheses after the variable name. Later, to define the number of dimensions and elements in the array, use ReDim.

```
Dim TestArray()
```

 To declare the data type of a variable without using As <type>, append the data type declaration character to the variable's name. For example, if you want the variable Income to contain only Currency data, declare it as:

Dim Income@

If you want the variable to contain only Double data, declare it as:

Dim Income#

For more information about data type declaration characters, see "Using a type-declaration character" in Chapter 2, "Understanding variables and data types."

To set the value of an instance handle to an instance of a class, use Set.

**Example** The following code manipulates objects created using various Dim statements:

```
Sub Start()
  Dim i As Integer
  Dim myControl As AcControl
  Dim Obj1 As AcLabelControl
  Dim Obj2 As AcLabelControl, Msg As String
  Super::Start( )
  ' Assignment
  Set myControl = me
  i = 103
  ' Standard instantiation
  Set Obj1 = New AcLabelControl
  ' Dynamic instantiation
  Set Obj2 = NewInstance ( "AcLabelControl" )
  Obj2.BackgroundColor = Red
  Msg = "The numeric value for the background color "
     & "of Obj2 is: " & Obj2.BackgroundColor
+
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Option Strict statement ReDim statement Set statement

# Do...Loop statement

Repeats a block of instructions while a specified condition is True or until a specified condition becomes True.

Syntax Syntax 1

Do [{ While | Until } <expression to reevaluate>]

<statements>

[Exit Do]

<statements>

Loop

Syntax 2

Do

<statements>

[Exit Do]

<statements>

Loop [{ While | Until } <expression to reevaluate>]

**Description** If the condition to evaluate is placed at the bottom of the loop (after Loop), the loop executes at least once. If the condition to evaluate is placed after Do, the loop does not execute at all if the condition is initially False.

# Parameters While

Keyword indicating that every time Actuate Basic begins or repeats the instructions between Do and Loop, Actuate Basic must determine whether <expression to reevaluate> is True or False. The following conditions apply to <expression to reevaluate>:

- If <expression to reevaluate> is True, the program executes the loop.
- If <expression to reevaluate> is False, or has become False during the last trip through the loop, Actuate Basic skips the intervening statements and passes control to the statement following Loop.

# Until

Keyword indicating that every time Actuate Basic begins or repeats the instructions between Do and Loop, Actuate Basic must determine whether <expression to reevaluate> is True or False. The following conditions apply to <expression to reevaluate>:

- If <expression to reevaluate> is False, the program executes the loop.
- If <expression to reevaluate> is True, or has become True during the last trip through the loop, Actuate Basic skips the intervening statements and passes control to the statement following Loop.

The following example assumes Counter starts at 1 and increments by 1 each time through the loop, the following statement fragments are equivalent. In either case, the associated loop is executed nine times.

Do While Counter < 10 Loop Until Counter = 10

If Counter is instead incremented by 2 each time, the first associated loop is executed five (1, 3, 5, 7, 9) times. The second loop, however, never stops executing because Counter increments from 9 to 11 and so never exactly equals 10. A minor

correction can address this issue. In the following example, Counter is incremented by 2 each time and the loop executes only five times:

Do Until Counter >= 10

### Exit Do

A keyword indicating that the program must immediately interrupt the loop at the point at which the keyword occurs, and pass control to the statement following Loop.

#### <expression to reevaluate>

Any valid numeric or string expression that evaluates to True or False.

#### <statements>

Zero or more valid Actuate Basic statements.

- **Tips** To stop a loop from continuing based upon whether a given expression is True or False, use Exit Do within a conditional structure like If Then.
  - Exit Do does not automatically exit all nested loops. When Do...Loop statements are nested, an Exit Do in a loop transfers control to the loop that is logically nested one level above the current one. For example, if you want to provide users with a way to back out of a complicated structure, map out a logical backtracking path in the form of several Exit Do statements, one at each level to which they are backtracking.
  - Evaluate Boolean variables using the keywords True or False.
- **Example** The following example asks the user whether to display Demo 1 or Demo 2 first. In Demo 1, the program prompts the user for a number within a certain range. If that number falls outside the range, the Demo repeats the prompt. In Demo 2, the program prints the numbers from 1 to 11.

```
Sub Start()
  Dim DemoNum As Integer, Counter As Integer
  Dim FirstShown As Integer
  Dim SecondShown As Integer, BothShown As Integer
  Dim Msg As String
  Super::Start( )
  ' Initialize flags and counters before the top of the loop
  ' Initially set Counter to 1
  Counter = 1
  ' Assume Demo 1 has not been shown
  FirstShown = False
  ' Assume Demo 2 has not been shown
  SecondShown = False
  ' Assume neither have been shown
  BothShown = False
  ' Start with Demo 1
  DemoNum = 1
```

```
' The outer loop will execute until the exit condition
  ' BothShown is true
  Do Until BothShown
     ' First inner loop is executed if the user typed 1
     Do While DemoNum = 1
        ' This is the first demo
       Msg = "Random number is " + CStr(Int(10 * Rnd + 1))
        ShowFactoryStatus( Msg )
        ' Remember Demo 1 has been shown
       DemoNum = 2
        ' User is ready for Demo 2
       FirstShown = True
        ' Exit first inner loop
        Exit Do
     Loop
     ' The second inner loop is executed when the user types 2
     ' but only if user hasn't seen the second demo. In other
     ' words, SecondShown is false.
     Do While DemoNum = 2 And Not SecondShown
        ' This nested loop will execute 11 times and return to
        ' 2nd inner loop
       Do Until Counter = 12
          ShowFactoryStatus( CStr( Counter ))
          Counter = Counter + 1
       Loop
       DemoNum = 1
        ' Exit condition for second inner loop
        SecondShown = True
     Loop
     ' Set up final exit condition from outer loop
     If FirstShown and SecondShown then BothShown = True
  Loop
  ShowFactoryStatus ( "Both demos have been shown. Goodbye!" )
End Sub
```

# See also Exit statement If...Then...Else statement While...Wend statement

# End statement

Signals the completion of a multiline If...Then...Else statement, a Select Case statement, a block of Function or Sub statements, a Declare section, a Type or Class definition, or terminates the entire program.

Syntax End [{ Class | Declare | Function | If | Select | Sub | Type }]

### Parameters End

Keyword that terminates the entire program.

### End Class

Keyword that signifies the completion of a Class statement. End Class must terminate every Class statement.

#### End Declare

Keyword that completes a multiple-line Declare statement. End Declare must terminate a Declare statement that is in a multiline or block format.

#### End Function

Keyword that completes a Function definition. End Function must terminate every Function statement. Actuate Basic automatically supplies End Function when you type a Function statement.

#### End If

Keyword that completes a multiple-line If...Then...Else statement. End If must terminate every If...Then...Else statement that is in multiline or block format.

#### End Select

Keyword that completes a Select Case statement. End Select must terminate every Select Case statement.

#### End Sub

Keyword that completes a Sub procedure. End Sub must terminate every Sub statement. Actuate Basic automatically supplies End Sub when you type a Sub statement.

### End Type

Keyword that completes a Type statement. End Type must terminate every Type statement.

**Example** The following example uses a Select Case block to return the name of the day of the week for the current day. To use this example, remove the End statement and run the example. The example generates an error message, Select Case without End Select.

```
Sub Start()
Dim DayNumber As Integer, DayName As String
```

```
Super::Start( )
  DayNumber = Weekday( Date )
  Select Case DayNumber
     Case 1
       DayName = "Sunday"
     Case 2
       DayName = "Monday"
     Case 3
       DayName = "Tuesday"
     Case 4
       DayName = "Wednesday"
     Case 5
       DayName = "Thursday"
     Case 6
       DayName = "Friday"
     Case 7
       DayName = "Saturday"
  End Select
  ShowFactoryStatus ( DayName )
End Sub
```

See also Class statement Declare statement

Function...End Function statement Function...End Function statement If...Then...Else statement Select Case statement Stop statement Sub...End Sub statement Type...End Type statement

# Enum...End Enum statement

Declares an enumerated type. The Enum...End Enum statement must be placed within a Declare...End Declare block.

Syntax Enum <user-defined enumeration type>

```
<name 1> [= value 1]
<name 2> [= value 2]
...
<name N> [= value N]
```

End Enum

```
Parameters <user-defined enumeration type>
```

The name of the type. This name is used to declare enum variables.

### <name 1>...<name N>

The names of the values an enum variable of the declared type can take.

### <value 1>...<value N>

Optional explicit values for the names. If not set, the first value of the name list is set to zero, and the value of each following name is increased by one.

# **Example** The following code creates and uses an enumerated data type to determine branches in a section of code.

To create an enum, you place the enum statement in a Declare statement block, and save it as an Actuate Basic source (.bas) library file.

```
Declare
Enum DatabaseType
ODBC
Oracle
DB2
End Enum
End Declare
```

The enum is then used as any other global type, as shown in the following code:

```
Sub Start()
  Dim DataSource as DatabaseType
  ' Place code to determine database type here
  ' Assign value to enumeration variable, for example:
  DataSource = DB2
  ' Use the enum variable to branch
  Select Case DataSource
     Case ODBC
       ' Code specific to ODBC database
     Case Oracle
       ' Code specific to Oracle database
     Case DB2
       ' Code specific to DB2 database
  End Select
  Super::Start()
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Declare...End Declare statement

# **Environ, Environ\$ functions**

Returns the setting of a specified environment variable.

Syntax Environ\$(<environment variable name>)

Environ[\$](<environment variable name>)

or

Environ\$()

Environ[\$]()

# Parameters <environment variable name>

String expression that specifies the name of an environment variable about which you want to know the settings. An environment variable is an internal variable an operating system uses to hold information about the computer system in which your program is running. These variables contain information about the current drive and path, and special information needed by various programs. Although each computer system has a different set of environment variables with different values, few common ones exist, such as PATH.

<environment variable name> must not contain a space.

### 

Numeric expression that specifies the variable's entry number. This number corresponds to its position in the environment table, where the first line in the table is line 1. The number must be greater than zero. The following example returns the contents of the CLASSPATH environment variable:

PathToJavaClasses = Environ("CLASSPATH")

# Returns Environ: Variant Environ\$: String

- If is greater than the number of lines in the table, Environ[\$] returns a Empty string.
- If you call Environ[\$] using <environment variable name>, the function returns only the content of the variable, not its name.
- If you call Environ[\$] using , the function returns both the content of the variable in that position and its name.
- **Example** The following Windows-based code looks for the PATH environment variable in the user's system. If it finds the variable, it displays its entry number and the length of the string it contains.

```
Sub Start( )
  Dim EnvString As String, LineNumber As Integer
  Dim Msg As String, Pathlen As Integer
```

```
Super::Start( )
  ' Initialize line index to 1
  LineNumber = 1
  Do
     ' Get environment variable
     EnvString = Environ( LineNumber )
     ' Is it the PATH entry?
     If UCase$( Left( EnvString, 5 ) ) = "PATH=" Then
        ' If so, get its length
        Pathlen = Len( Environ( "PATH" ) )
       Msg = "The PATH entry occurs at position " & LineNumber
          & ". Its length is " & Pathlen & "."
+
       Exit Do
     Else
        ' Not the PATH entry, so increment line counter
       LineNumber = LineNumber + 1
     End If
  Loop Until EnvString = ""
  If Pathlen > 0 Then
     ShowFactoryStatus ( Msg )
  Else
     ShowFactoryStatus ( "No PATH environment variable." )
  End If
End Sub
```

```
See also Command, Command$ functions
```

# **EOF** function

Returns a value that indicates whether the end of a file being input has been reached.

Syntax EOF(<open file number>)

# Parameters <open file number>

Numeric expression that is the file descriptor used in the previously issued Open statement to open the target file. <open file number> must be the number of a currently open file, and must also refer to a disk file.

For example, the following code opens an existing disk file and then uses EOF repeatedly to test whether or not the end of the file has yet been reached:

Open "c:\Vb\Tests\test.fil" For Input As #1 Do While Not EOF(1)'Test for end of file. Input #1, FileData 'Read line of data. Loop

**Returns** Integer (0 for False or 1 for True)

EOF behaves differently depending upon the mode of access you use for the file:

- For sequential files, EOF returns True if the end of the file has been reached. Otherwise, EOF returns False.
- For random or binary files, EOF returns True if the last executed Get statement was unable to read an entire record. Otherwise, EOF returns False.
- **Tip** Use EOF when you handle sequential files to avoid an error generated when you try to access a file's contents past the end of a file.
- **Example** The following example creates a test file on disk that contains random numbers. The procedure opens the test file, reports the last value in the test file, then deletes the sample file from the disk. This example overrides Start to generate the test file, report the value, and prompt the user to delete the test file. To use this example, paste the procedure, MakeDataFile, after the End Function of the Start() procedure or save it in an Actuate code module (.bas) file.

```
Sub Start()
  Dim TempVar As Integer, Msg As String
  Super::Start( )
  ' Generate sample file
  MakeDataFile
  ' Open sample for input
  Open "Test.fil" For Input As #1
  ' Check for end of file
  Do While Not EOF(1)
     ' Read data
     Input #1, TempVar
  Loop
  ' Close test file
  Close #1
  Msg = "The last value in the test file was "
     & TempVar & "."
+
  ShowFactoryStatus ( Msg )
  Msg = "Now deleting test file."
  ShowFactoryStatus ( Msg )
  ' Delete test file
  Kill "Test.fil"
End Sub
'Here is the procedure that generates the test file
Sub MakeDataFile()
  Dim I As Integer
```

```
' Open file for output
Open "Test.fil" For Output As #1
' Generate random values
For I = 0 To 250
        Print #1, Int(711 * Rnd)
Next I
' Close test file
Close #1
End Sub
```

See also Loc function LOF function Open statement

# **Erase statement**

De-allocates memory reserved for dynamic arrays, or reinitializes the elements in a fixed array.

Syntax Erase <array name>... [, <array name> ]

**Description** Erase behaves differently depending upon whether <array name> refers to a fixed or to a dynamic array:

- When Erase refers to a fixed array, Erase reinitializes the contents of the array but does not free up memory.
- When Erase refers to a dynamic array, Erase recovers the memory previously allocated to the array.

Erase reinitializes fixed arrays as shown in Table 6-6.

	-
Type of fixed array	Array element set to
Numeric	0 (Zero)
String (variable-length)	Empty
Variant	Empty
User-Defined	The value as shown here for each component type, taken separately
Object	Nothing
CPointer	Null

Table 6-6How Erase initializes arrays

#### Parameters <array name>

Any valid array name. Can be a variable name or expression. An expression cannot contain parentheses. For example, the following statements are valid:

Erase BigArray Erase my.Array

The following statement is invalid:

Erase my.Array(1)

- **Tips** If you erase a dynamic array, you can use Dim or ReDim to create a new dynamic array with the same name as the old one.
  - You do not need to erase a dynamic array before you redimension it with ReDim.

**Example** The following example creates an array, fills it with values, then erases it:

```
Sub Start()
  ' Declare variables
  Dim I As Integer, J As Integer, Total As Long, Msg
  ' Create 2-D integer array
  Dim Matrix(50, 50) As Integer
  Super::Start( )
  For I = 1 To 50
     For J = 1 To 50
        ' Put some values into array
       Matrix(I, J) = J
     Next J
  Next I
  ' Erase array
  Erase Matrix
  ' Initialize total counter
  Total = 0
  For I = 1 To 50
     For J = 1 To 50
        ' Sum elements after Erase to make sure all are zero
        Total = Total + Matrix(I, J)
     Next J
  Next I
  Msq = "An array has been created, filled, and erased. When "
     & "the Erase statement was executed, zeros "
+
     & "replaced the previous contents of each element."
     & " The total of all elements is now " & Total
     & ", showing that all elements have been cleared."
  ' Display message
  ShowFactoryStatus ( Msg )
End Sub
```

See also Dim statement ReDim statement

# **Erl function**

Returns the line number for the most recent run-time error.

### Syntax Erl

Returns Integer

- When it runs an application, Actuate Basic uses Erl to report the number of the line at which the most recent error occurred, or of the line most closely preceding it.
- When Erl is 0, it means one of three things:
  - No run-time error has occurred.
  - An error could have occurred, but there was no line number just before the point at which the error occurred, or your program contains no line numbers.
  - Actuate Basic has reset Erl to 0 because it executed Resume or On Error, or because it executed Exit Sub or Exit Function from within an error handling routine.
- Erl returns only a line number, not a line label.
- **Tips** To preserve the value of Erl before it gets reset to 0, immediately assign it to a variable.
  - To set the value of Erl indirectly, simulate an error condition using an Error statement.
- **Example** In the following example, the program generates an error because it attempts to divide a number by zero. It then displays a message indicating the line number of the error.

```
Sub Start( )
Dim XVar, YVar, ZVar
Super::Start( )
' Set up error handler
On Error GoTo ErrorHandler
YVar = 1
' Now cause division by zero error:
XVar = YVar / ZVar
Exit Sub
```

' Error handler starts here
ErrorHandler:
 ShowFactoryStatus( "Error occurred at program line " & Erl )
 Resume Next
End Sub
For information about using the code examples see "Ilsing the code example

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Err statement Error statement On Error statement Resume statement

# **Err function**

Returns the code number for the most recent run-time error.

- Syntax Err
- Returns Integer
  - Returns an Integer between 0 and 2,147,483,647.
  - When Actuate Basic runs an application, it uses Err to report the occurrence of a run-time error and its error code.
  - When Err is 0, it means no run-time error has occurred.
  - Actuate Basic sets or resets Err to 0 each time it executes Resume or On Error or when it executes Exit Sub or Exit Function from within an error handling routine.
  - **Tips** To preserve the value of Err before it is reset to 0, immediately assign it to a variable.
    - To set the value of Err directly, use an Err statement. To set it indirectly, simulate an error condition using an Error statement.
- **Example** In the following example, the program generates an error because it attempts to divide a number by zero. It then displays the code number of the error.

```
Sub Start( )
Dim XVar, YVar, ZVar
Super::Start( )
' Set up error handler
On Error GoTo ErrorHandler
YVar = 1
' Now cause division by zero error:
XVar = YVar / ZVar
Exit Sub
```

```
' Error handler starts here
ErrorHandler:
   ShowFactoryStatus( "Error number " & Err & " occurred" )
   Resume Next
End Sub
```

See also Erl function Err statement Error statement On Error statement Resume statement Error, Error\$ functions Error statement

# **Err statement**

Sets Err to a given value.

Syntax Err = <error code>

#### Parameters <error code>

Numeric expression between 0 and 2,147,483,647 indicating the code of the most recent run-time error.

- **Description** When it runs an application, Actuate Basic uses Err to record the occurrence of a run-time error and its error code.
  - When Err is 0, it means no run-time error has occurred.
  - Actuate Basic sets or resets Err to 0 whenever it executes Resume or On Error, or whenever it executes Exit Sub or Exit Function from within an error handling routine.
  - **Tips** To define and use your own error code, use a value for <error code> that is greater than any of the standard Actuate Basic error codes. You can work down from error code 2,147,483,647 to find an available user-defined error number. Write a routine that assigns that number to Err whenever certain error conditions that you define are true. Also write an error handling routine that translates the error number into an understandable message for the user.
    - There are many user-defined codes among the standard ones.
    - The Error statement can set Err to any value, because it simulates any run-time error.
    - To preserve the value of Err before it gets reset to 0, immediately assign it to a variable.

**Example** The following example prompts the user for a filename, attempts to open the file to verify its existence, then handles a range of errors that might result:

```
Sub Start()
  Dim FileName As String
  Super::Start( )
  ' Set up error handler
  On Error Resume Next
  FileName = "ThisFileShouldNotExist.txt"
  Do
     ' Clear any error
     Err = 0
     ' Attempt to open file
     Open FileName For Input As #1
     ' Handle error, if any
     Select Case Err
        ' No error = success!
        Case 0
          ShowFactoryStatus( UCase$( FileName ) & " found." )
        ' I/O errors
        Case 14
          ShowFactoryStatus( "File not found." )
          ' Don't try again = failed!
          Exit Do
        ' Handle all other cases
        Case Else
          ShowFactoryStatus( "Error: Sorry! Cannot continue!" )
           ' Don't try again = failed!
          Exit Do
     End Select
  Loop Until Err = 0
   ' Clean up
  Close #1
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Erl function Err function Error, Error\$ functions Error statement

# **Error, Error\$ functions**

Returns the error message for the most recent run-time error, or for a specified error number.

Syntax Error[(<error code>)]

Error\$[(<error code>)]

# Parameters <error code> Numeric expression between 1 and 2,147,483,647 inclusive that is the error number to be displayed as a message string by Error[\$]. If <error code> is not defined by Actuate Basic, Error[\$] returns the message string: User-defined error.

- **Returns** Error: Variant Error\$: String
  - With <error code>, the Error[\$] function returns the message for the specified error number. In cases where context-sensitive information is inserted in the message at run time, the context-sensitive string is replaced with a space character. For example, in the following message, a space is inserted where the array element index number would normally appear:

```
Control array element ' ' doesn't exist.
```

- Without <error code>, the Error[\$] function returns the message for the most recent run-time error. If no error has occurred, the Error[\$] function returns an empty string.
- **Example** The following example shows error messages for user input number errors:

```
Sub Start()
  ' Demo of Error$ function and Error statement
  Dim Msg As String
  Dim UserError As Integer
  Super::Start( )
  ' Set up error handler
  On Error GoTo ErrorHandler
  ' Get a random number in the range 1 to 256
  UserError = 255 \times Rnd + 1
  ' Simulate given error
  Error UserError
  Exit Sub
  ' Error handler starts here
ErrorHandler:
  Msg = "The message for error number "
     & Err & " is:" & Error( Err )
  ShowFactoryStatus ( Msg )
  Resume Next
End Sub
```
See also Erl function Err function Error statement

### Error statement

Simulates the occurrence of an error. Also, generates a user-defined error.

Syntax Error <error code>[,<message> [,<message>...]]

**Description** By setting the value of Err to <error code>, this statement simulates the occurrence of a specified error. It then passes control to whatever error-handling routine you have enabled, if any, so that in a debugging session you can verify whether or not that error-handler works.

If you have not enabled an error-handler when your program executes this statement, Actuate Basic stops your program and displays its own internal error message if it has one corresponding to <error code>, or User-defined error if it does not.

#### Parameters <error code>

Integer between 1 and 2,147,483,647 that is the error you wish to artificially generate.

For example, the following statement causes Actuate Basic to stop your program and display the message Division by zero, whether or not your program has actually attempted to divide anything by zero:

Error 8

### <message>

You can pass any number of arguments in order to replace standard error messages. All extra messages are converted to strings and concatenated together. Then, the concatenated message is displayed instead of the standard error message.

- **Tips** To define your own error code, use a value for <error code> that is greater than any of the standard Actuate Basic error codes. You can work down from error code 2,147,483,647 to find an available user-defined error number.
  - You will find many user-defined codes among the standard error codes Actuate Basic uses.
- **Example** The following example prompts the user for an error number, simulates the appropriate error, the passes control to an error handler. The error handler then retrieves the corresponding message.

```
Sub Start()
   ' Demo of Error$ function and Error statement
  Dim Msg As String, NL As String
  Dim UserError As Integer
  Super::Start( )
  ' Set up error handler
  On Error GoTo ErrorHandler
  ' Get a random number in the range 1 to 256
  UserError = 255 \times Rnd + 1
  ' Simulate given error
  Error UserError
  Exit Sub
  ' Error handler starts here
ErrorHandler:
  Msg = "The message for error number "
     & Err & " is:" & Error(Err)
  ShowFactoryStatus ( Msg )
  Resume Next
End Sub
```

See also Erl function Err function Err statement On Error statement Resume statement

# **Exit statement**

Terminates execution of the instructions specified in a Do Loop or For Next structure, or in a Function or Sub procedure.

Syntax Exit { Do | For | Function | Sub }

### Parameters Exit Do

Keyword used to terminate a Do...Loop statement.

- When it encounters Exit Do, Actuate Basic transfers control to the statement following the Loop keyword.
- If it encounters Exit Do within a nested loop, the program transfers control to the loop that is nested one level above the one in which it occurs.
- Use Exit Do only within a Do Loop statement.

### Exit For

Keyword used to terminate a For...Next statement.

- When it encounters Exit For, Actuate Basic transfers control to the statement following the Next keyword.
- If it encounters Exit For within a nested loop, the program transfers control to the loop that is nested one level above the one in which it occurs.
- Use Exit For only within a For Next statement.

### **Exit Function**

Keyword used to terminate a Function statement. When it encounters Exit Function, Actuate Basic transfers control to the statement following the one that called the current Function statement.

#### Exit Sub

Keyword used to terminate a Sub statement. When it encounters Exit Sub, Actuate Basic transfers control to the statement following the one that called the current Sub statement.

- **Tips** To stop a loop from continuing or repeating based upon whether or not a given expression is true, use Exit Do within a conditional structure like an If...Then...Else statement.
  - Do not confuse Exit with End. Use End to define the completion of a structure.
- **Example** The following example shows how to use nested statements with Exit. When the generated random number matches one of those specified in the first Case clause, Actuate Basic exits the For...Next statement and the Do...Loop statement in two separate steps via both Exit For and Exit Do. When the random number matches one of those in the second Case clause, Actuate Basic exits both statements in one step, using Exit Do alone. A message reports in which Case clause the loops were terminated.

```
Sub Start( )
Dim I As Integer, Num As Integer, Msg As String
Dim BumpOut As Integer
Super::Start( )
' Set a flag for ultimate exit
BumpOut = False
' Generate random numbers
Randomize Timer
' Set up an infinite loop
Do
' If we exited FOR, let's also exit DO
If BumpOut Then Exit Do
```

```
For I = 1 To 1000
     ' Generate random number
     Num = Int(Rnd * 100)
     Select Case Num
        Case 7, 11, 13, 25
           ' This case causes an immediate exit from the For
          ' loop and will cause an exit from the Do loop on the
           ' next iteration
          ' Set flag for exit from Do next time
          BumpOut = True
          Exit For
        Case 29, 35, 87
          ' This case causes an immediate exit from both loops
           ' but leaves BumpOut set to false so we can tell
           ' where the exit occurred
          ' Directly exit in one step
          Exit Do
      End Select
     Next I
  Loop
  Msg = "Exited because of " & Num
  ' We exited in the first Case clause
  If BumpOut Then
     Msg = Msg & " (1st case: Two-step exit)."
  Else
     Msg = Msg & " (2nd case: One-step exit)."
  End If
  ShowFactoryStatus ( Msg )
End Sub
```

See also Do...Loop statement End statement For...Next statement Function...End Function statement Sub...End Sub statement

# **Exp function**

Raises e to the specified power.

Syntax Exp(<power>)

### Parameters <power>

Number, numeric expression, or Variant of VarType 8 (String) that specifies the exponent. The following rules apply to <power>:

- <power> cannot be greater than 709.782712893.
- If <power > is a String, it is parsed according to the formatting rules of the current run-time locale.

### Returns Double

If <power> evaluates to Null, Exp returns Null.

- **Tip** Exp is the inverse of Log.
- **Example** The following example generates a trigonometric angle expressed in radians, then uses the Exp function to calculate the hyperbolic sin of that angle:

```
Sub Start()
Dim Angle As Double, HyperSin As Double, Pi As Double
Dim Msg As String
Super::Start()
Pi = 3.14159265358979
' The angle in radians
Angle = Rnd * Pi
'Calculate hyperbolic sine
HyperSin = ( Exp( Angle ) - Exp( -1 * Angle ) ) / 2
Msg = "The hyperbolic sine of " & Angle & " is " & HyperSin
ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Log function

### ExtendSearchPath function

Adds directories to search for included images and files specified in the FindFile function.

- Syntax ExtendSearchPath(<directory>)
- **Description** Call ExtendSearchPath to specify additional directories to search when locating images and files specified in the FindFile function. Call this function once for each directory to be added to the search path.

You can call ExtendSearchPath at factory time, view time, or both. If you call ExtendSearchPath at factory time, the directory is added to the search path only when locating files included at factory time. If you call ExtendSearchPath at view

time, the directory is added to the search path only when locating files included at view time.

- Parameters <directory> A directory to add to the search path.
  - **Example** The following example adds C:\Documents and M:\Documents to the complete search path at factory time:

```
Sub Start()
Super::Start()
ExtendSearchPath( "C:\Documents\" )
ExtendSearchPath( "M:\Documents\" )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also FindFile function

### **FileAttr function**

Returns the mode of an open file or its DOS file handle.

**Syntax** FileAttr(<file number>, <info type>)

### Parameters <file number>

Numeric expression that is the number you used as a file descriptor when you opened the target file. <file number> must evaluate to the number of a currently open file.

### <info type>

Numeric expression that specifies whether to return the open mode of the file (1), or the DOS file handle (2). <info type> must evaluate or round to 1 or 2.

In the following code example, Actuate Basic opens a file, then stores values returned by FileAttr in corresponding variables:

```
Open "TEST" For Input As #1
OpenMode = FileAttr(1,1)
DOSHandle = FileAttr(1,2)
```

### Returns Integer

 When <info type> is 1, FileAttr returns a number that defines the mode under which the file was opened. Table 6-7 lists possible values for FileAttr when <info type> is 1, and the corresponding modes.

	······································	
Return value	Mode for which file was opened	
1	Input	
2	Output	
4	Random	
8	Append	
32	Binary	

 Table 6-7
 Values for FileAttr and the corresponding modes

- When <info type> is 2, FileAttr returns the file handle that DOS has assigned to the open file.
- **Example** In the following example, Actuate Basic creates a test file by opening it for Append. Then it displays a message indicating the DOS handle and open mode of the file.

```
Sub Start()
  Dim FileNum As Integer, DOSHandle As Long
  Dim OpenMode As String, Msg As String
  Super::Start( )
  ' Get next available file number
  FileNum = FreeFile
  ' Create sample file
  Open "TESTFILE" For Append As FileNum
  ' Get file DOS handle
  DOSHandle = FileAttr(FileNum, 2)
  ' Determine OpenMode
  Select Case FileAttr(FileNum, 1)
     Case 1: OpenMode = "Input"
     Case 2: OpenMode = "Output"
     Case 4: OpenMode = "Random"
     Case 8: OpenMode = "Append"
     Case 32:OpenMode = "Binary"
  End Select
  ' Close test file
  Close FileNum
  Msg = "The file assigned DOS file handle " & DOSHandle
   & " was opened for " & OpenMode & "."
+
  ShowFactoryStatus ( Msg )
  ' Delete test file
  Kill "TESTFILE"
End Sub
```

See also GetAttr function Open statement SetAttr statement

# FileCopy statement

Copies a file. Similar to the DOS command Copy.

Syntax FileCopy <source>, <destination>

### Parameters <source>

String expression that specifies the file to copy. Can include optional drive and path information. <source> cannot include wildcard characters and must refer to an existing file.

Default path: Current default drive and directory.

### <destination>

String expression that specifies where <source> is to be copied. Can include optional drive and path information. <destination> must conform to platform's file naming conventions and cannot include wildcard characters. The default path is the current default drive and directory.

<source> or <destination> can optionally specify full path information, in which case either has the following syntax:

[<drive:>] [ \ ]<directory>[\<directory>]...(Windows)

[/]<directory>[/<directory>]...(UNIX)

### <drive:>

Character, followed by a colon. Specifies the drive (Windows only).

### <directory>

String expression that specifies the name of a directory or subdirectory.

The following example assumes a file named Temp.fil exists on a Windows system, and that it is located on the current drive and in the current directory. The following code copies it, places the new copy on drive D in the subdirectory \Files\Archives, and names that copy Temp.arc, while leaving Temp.fil alone where it is:

```
NewName = "D:\Files\Archives\Temp.arc"
FileCopy "Temp.fil", NewName
```

**Example** The following example copies a file from the Actuate install directory:

```
Sub Start()
  Dim SourceFile As String, DestFile As String, Msg As String
  Super::Start( )
  On Error GoTo ErrHandler
  SourceFile = "C:\Program Files\Actuate11\readme.rtf"
  DestFile = "C:\actuate readme.rtf"
  ' Copy file to destination
  FileCopy SourceFile, DestFile
  Msg = "File readme.rtf copied to C:\actuate readme.rtf"
  ShowFactoryStatus ( Msg )
  Exit Function
  ErrHandler:
  Select Case Err
  ' Path not found
  Case 48
     ShowFactoryStatus( SourceFile & " not found." )
  Case 40
     ShowFactoryStatus( "Permission denied, or drive not
  available." )
  case Else
     Msg = "An error occurred. Try specifying the full path "
        & "name of the source or destination files, "
+
        & "or of both."
+
  End Select
  ShowFactoryStatus ( Msg )
End Sub
```

See also Kill statement Name statement

# FileDateTime function

Returns a string that indicates the date and time a specified file was created or last modified. The date is formatted according to the formatting rules of the current run-time locale.

Syntax FileDateTime(<file spec>)

### Parameters <file spec>

String expression that specifies the name of a valid file. Can include optional drive and path information. The file and any path must exist. Wildcard characters are not allowed in <file spec>. The default path is the current default drive and directory.

<file spec> can optionally specify full path information, in which case it has the following syntax:

[<drive:>] [ \ ]<directory>[\<directory>]... (Windows)

[ / ]<directory>[/<directory>]... (UNIX)

### <drive:>

Character, followed by a colon, that specifies the drive on which the file is located (Windows only).

### <directory>

String expression that specifies the name of a directory or subdirectory that constitutes a branch in the full path specification of <file spec>.

The following example statement determines when the file Mydata.mdb on a Windows system in the subdirectory C:\Access was last modified, and converts the returned string to a date variant:

LastSaved = CVDate(FileDateTime("C:\Access\Mydata.mdb"))

Returns String

**Example** The following example displays the date and time a file was created or last modified and its size in bytes:

```
Sub Start()
  Dim Msq As String, TimeStamp As String, UserFile As String
  Super::Start( )
  On Error Goto ErrHandler
  ' Get file name
  UserFile = "C:\Program Files\Actuate11\readme.rtf"
  ' Get file date/time
  TimeStamp = FileDateTime( UserFile )
  Msg = UCase$( UserFile ) & " created or last modified on "
     & Format( TimeStamp, "dddd, mmmm dd, yyyy" )
+
   & " at " & Format( TimeStamp, "h:nn AM/PM" )
+
+
   & ". Its size is " & FileLen( UserFile )
     & " bytes."
+
  ShowFactoryStatus ( Msg )
  Exit Sub
ErrHandler:
  Msg = "Sorry! An error occurred. "
   & "Please change this method code "
     & "to use a different file name."
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples" earlier in this chapter.

See also FileLen function

FileTimeStamp function GetAttr function LOF function

# **FileExists function**

Determines whether or not a given file or directory exists.

Syntax FileExists(<filename or dirname> [As String]) [As Boolean]

### Parameters <filename or dirname>

String expression enclosed in double quotation marks that is the name of the file, directory, or subdirectory the existence of which you wish to determine. The default path is the current drive and directory.

<filename or dirname> can optionally specify a full path, in which case it has the following syntax:

[<drive:>][\]<directory>[\<directory>]...(Windows)

[/]<directory>[/<directory>]...(UNIX)

### <drive:>

Character, followed by a colon, that specifies the drive (Windows only).

### <directory>

String expression that specifies the name of a directory or subdirectory.

### Returns Boolean

True if the file or directory exists, False otherwise.

- **Tip** It is good programming practice to evaluate Boolean variables by using the keywords True or False, instead of by inspecting their content for a nonzero (True) or zero (False) numeric value.
- **Example** Assuming a file called Readme.txt exists in the current directory, the following statement assigns the value True to the variable FileAlreadyInstalled:

FileAlreadyInstalled = FileExists( "Readme.txt" )

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also FileDateTime function GetAttr function Kill statement Name statement Open statement

# **FileLen function**

Returns the length of a given file in bytes.

Syntax FileLen(<file spec>)

### Parameters <file spec>

String expression that specifies the name of a valid file. Can include optional drive and path information. The default path is the current default drive and directory. The following conditions apply to <file spec>:

- Must be an unambiguous specification. Wildcard characters are not allowed.
- Indicated file or files must exist.
- Path, if specified, must exist.

<file spec> can optionally specify full path information, in which case it has the following syntax:

[<drive:>] [ \ ]<directory>[\<directory>]... (Windows)

```
[ / ]<directory>[/<directory>]... (UNIX)
```

### <drive:>

Character, followed by a colon that specifies the drive on which the file is located (Windows only).

### <directory>

String expression that specifies the name of a directory or subdirectory that constitutes a branch in the full path specification of <file spec>.

For example, the following statement determines the length of the file Mydata.mdb on a Windows system in the subdirectory C:\Access, and assigns the value to a variable:

SizeOfFile = FileLen("C:\Access\Mydata.mdb")

### Returns Integer

If the file specified in <file spec> is open when you call FileLen, the value FileLen returns represents the length of the file before it was opened.

**Example** The following example displays the date and time a file was created or last modified and its size in bytes:

```
Sub Start( )
Dim Msg As String, TimeStamp As String, UserFile As String
Super::Start( )
On Error Goto ErrHandler
' Get file name
UserFile = "C:\Program Files\Actuate11\readme.rtf"
```

```
' Get file date/time
  TimeStamp = FileDateTime( UserFile )
  Msq = UCase$( UserFile ) & " created or last modified on "
     & Format( TimeStamp, "dddd, mmmm dd, yyyy" )
+
     & " at " & Format( TimeStamp, "h:nn AM/PM" )
+
   & ". Its size is " & FileLen( UserFile )
     & " bytes."
  ShowFactoryStatus ( Msg )
  Exit Sub
ErrHandler:
  Msg = "Sorry! An error occurred. "
     & "Please change this method code "
+
     & "to use a different file name."
  ShowFactoryStatus ( Msg )
End Sub
```

See also FileDateTime function GetAttr function LOF function

# **FileTimeStamp function**

Returns the date and time the specified file was created or last modified.

Syntax FileTimeStamp(<file spec>)

### Parameters <file spec>

String expression that specifies the name of a valid file. Can include optional drive and path information. The default path is the current default drive and directory. The following conditions apply to <file spec>:

- <file spec> must be an unambiguous specification. Wildcard characters are not allowed.
- Indicated file or files must exist.
- Path, if specified, must exist.

<file spec> can optionally specify full path information, in which case it has the following syntax:

[<drive:>] [ \ ]<directory>[\<directory>]... (Windows)

[ / ]<directory>[/<directory>]... (UNIX)

### <drive:>

Character, followed by a colon, that specifies the drive on which the file is located (Windows only).

### <directory>

String expression that specifies the name of a directory or subdirectory that constitutes a branch in the full path specification of <file spec>.

For example, the following statement determines when the file Mydata.mdb on a Windows system in the subdirectory C:\Access was last modified, and converts the returned string to a date variant:

```
LastSaved = CVDate(FileTimeStamp("C:\Access\Mydata.mdb"))
```

### Returns Date

**Example** The following example displays the date and time a file was created or last modified and its size in bytes:

```
Sub Start()
  Dim Msg As String, TimeStamp As String, UserFile As String
  Super::Start( )
  On Error Goto ErrHandler
  ' Get file name and date/time
  UserFile = "C:\Program Files\Actuate11\readme.rtf"
  TimeStamp = FileTimeStamp( UserFile )
  Msg = UCase$( UserFile ) & " created or last modified on "
     & Format( TimeStamp, "dddd, mmmm dd, yyyy" )
+
     & " at " & Format( TimeStamp, "h:nn AM/PM" )
+
    & ". Its size is " & FileLen( UserFile )
+
     & " bytes."
+
  ShowFactoryStatus ( Msg )
  Exit Sub
ErrHandler:
  Msq = "Sorry! An error occurred. "
   & "Please change this method code "
     & "to use a different file name."
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples" earlier in this chapter.

See also FileDateTime function FileLen function GetAttr function

# **FindFile function**

Resolves a relative file name by searching for the file through the existing search path.

- Syntax FindFile(<filename>)
- **Description** Call FindFile to resolve a relative file name by searching for the file through the existing search path.

You can call FindFile at factory time, view time, or both. If you call FindFile at factory time, Actuate searches for the specified file only at run time. In this case, Actuate searches the following:

1 If the referring report object design (.rod) file is open, the design search path.

For information about the design search path, see *Accessing Data using e.Report Designer Professional*.

- **2** The search path specified in the ExtendSearchPath function when ExtendSearchPath is called at factory time.
- **3** Global search path.

For information about the global search path, see *Accessing Data using e.Report Designer Professional*.

**4** Configuration file search path.

For information about the configuration file search path, see *Accessing Data using e.Report Designer Professional.* 

If you call FindFile at view time, Actuate searches for the specified file only at view time. In this case, Actuate searches the following:

- 1 The search path specified in the ExtendSearchPath function when ExtendSearchPath is called at view time.
- **2** Global search path.

For information about the global search path, see *Accessing Data using e.Report Designer Professional.* 

**3** Configuration file search path.

For information about the configuration file search path, see *Accessing Data* using e.Report Designer Professional.

### Parameters <filename>

The relative file path.

- **Returns** If the file is found, the absolute path to the file.
  - If the file is not found, an empty string.

**Example** The following example sets a file name in a relative path to search for, and calls ExtendSearchPath to add C:\Program Files\Actuate11\ and M:\Documents\ to the search path:

```
Sub Start()
  Dim RelativeFile as String, FullFileName as String
  Dim Msg As String
  Super::Start( )
  ' Add search paths
  ExtendSearchPath( "C:\Program Files\Actuate11\" )
  ' Add search paths
  ExtendSearchPath( "M:\Documents\" )
  ' Set file name
  RelativeFile = "readme.rtf"
  ' Find full name
  FullFileName = FindFile( RelativeFile )
  ' Verify full file name
  If FullFileName = "" Then
     Msg = "File not found!"
  Else
     Msg = "Full file name is " & FullFileName
  End If
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples" earlier in this chapter.

```
See also ExtendSearchPath function
```

# **Fix function**

Removes the fractional part of a numeric expression and returns the integer that remains.

Syntax Fix(<number to truncate>)

### Parameters <number to truncate>

A numeric expression from which the fractional part is removed so that only the integer component is returned.

The returned data type depends on the data type of <number to truncate> as shown in Table 6-8.

Data type of <number to="" truncate=""></number>	Data type returned
Variant of VarType 8 (String) that can convert to a number	Variant of VarType 5 (Double). If a String, it is parsed according to the formatting rules of the current run-time locale. For example:
	Fix("123,456") returns 123.456 on a French locale
	Fix ("123,456") returns 123456.00 on an English locale
Any other data type	Same data type as <number to="" truncate=""></number>
Null	Null

# Table 6-8 Relationship of data type of <number to truncate> to the data type returned

#### **Returns** Integer

Int and Fix are functionally similar but not identical. For negative values, Int returns the first negative integer less than or equal to <number to round>. Fix returns the first negative integer greater than or equal to <number to round>. CInt is also similar, but not identical, to both Int and Fix.

Using some sample values as arguments, Table 6-9 shows the differences between Int, Fix, and CInt.

_	Value	Int (Value)	Fix (Value)	CInt (Value)
-	3.7	3	3	4
	3.2	3	3	3
	3	3	3	3
	-3	-3	-3	-3
	-3.2	-4	-3	-3
	-3.7	-4	-3	-4

 Table 6-9
 Differences of values between Int, Fix, and CInt

**Tip** Fix is equivalent to Sgn (<number to round>) \* Int (Abs (<number to round>)).

**Example** The following example prompts the user for a number and displays the return values from Int, Fix, and CInt:

```
Sub Start( )
Dim Num As Double, Msg As String
Super::Start( )
```

```
' Get a random number between 1 and 256
Num = 255 * Rnd + 1
ShowFactoryStatus( "Int(" & Num & ") = " & Int( Num ) )
ShowFactoryStatus( "Fix(" & Num & ") = " & Fix( Num ) )
ShowFactoryStatus( "CInt(" & Num & ") = " & CInt( Num ) )
End Sub
```

See also CInt function Int function

### Format, Format\$ functions

Formats a numeric expression, date variant, or string according to the specified pattern and locale.

**Syntax** Format(<exprs to format>)

Format(<exprs to format>, <format pattern>)

Format\$(<exprs to format>, <format pattern>, <locale>)

#### Parameters <exprs to format>

Expression to display according to the specified <format pattern>.

Format[\$] works differently depending on whether <exprs to format> is numeric, date, or string data.Format[\$] parses numeric values specified as a String according to the current run-time locale before applying <format pattern>. For example, in the French run-time locale, the following string:

Format( "1234,56", "Currency", "pt\_PT")

results in the following output:

€ 1.234

This is because Format [\$] parses the string according to the French locale, which uses the comma (,) as a decimal separator then uses the specified locale, Portugal, for formatting and output.

#### <format pattern>

A keyword that has been predefined in Actuate Basic or a string of format characters that specifies how the expression displays. The <locale> argument, if specified, determines <format pattern>. The following conditions apply to <format pattern>:

If <format pattern> is omitted or is zero-length, and <exprs to format> is a
numeric expression, Format[\$] does the same thing as Str\$. Positive numbers

converted to strings using Format[\$] lack a leading space, whereas those converted using Str\$ retain a leading space.

- Do not mix different types format expressions, such as numeric, date/time or string in a single <format pattern> argument.
- Numeric values specified as an Integer or Double must use a period (.) as a decimal separator and cannot include a thousands separator. For example, 32434.28 is valid, but 3,2434.28 and 3434,28 are invalid.
- Numeric values specified as a String must use the user-specified locale format for decimal and thousands separator.
- Enclose <format pattern> within quotes.

### <locale>

String expression that specifies the locale to use for determining <format pattern> and the output format. If the locale is not specified, Null, or invalid, Format[\$] uses the current run-time locale.

### Returns Format: Variant

Format<sup>\$</sup>: String

If any parameter evaluates to Null, Format[\$] returns Null.

# Using Format[\$] with numeric data

Format numeric data using one of the following methods:

- Use the format keywords that have been predefined in Actuate Basic.
- Create your own user-defined formats using standard characters or symbols that have special functional meanings when used in a Format[\$] expression.

Table 6-10 lists the predefined numeric format keywords that you can use with Format[\$] examples and their results.

Keyword	Description	Example/Result
General	Returns as is, without	Format(3434.2899,
number	rounding or applying any formats.	"General number")
		Returns 3434.2899
		(continues)

 Table 6-10
 Predefined numeric format keywords

Keyword	Description	Example/Result
Currency	Returns as a string with thousands separators and rounded to the nearest hundredth if needed. Two digits to the right of the decimal separator. Encloses negative numbers in parentheses.	<pre>Format(3434.2899, "Currency") Returns \$3,434.29 on en_US run-time locale</pre>
	If <locale> is specified, the currency format depends on the specified locale. If <locale> is not specified, the currency format of the current run-time locale is used.</locale></locale>	
Fixed	Returns at least one digit to the left and two digits to the right of the desired congretor.	Format(3434.2899, "Fixed") Returns 3434.29
	Rounds to the nearest hundredth.	Format(.3122, "Fixed") Returns 0.31
Standard	Returns with thousands separators and rounded to the nearest hundredth, if needed. At least two digits to the right of the decimal separator.	Format(3434.2899, "Standard") Returns 3,434.29
Percent	Returns * 100 with a percent sign to the right. Two digits to the right of the decimal	Format(0.2899, "Percent") Returns 28.99%
	separator. Rounds to the nearest hundredth.	Format(0.28999, "Percent") Returns 29.00%
	The return type is a string. Do not use in a method for formatting AcDoubleControl and AcIntegerControl controls.	
Scientific	Returns in standard scientific notation, appropriately	Format(3434.2899, "Scientific")
	rounded. The return type is a string. Do not use in a method for formatting AcDoubleControl and AcIntegerControl controls.	Returns 3.43E+03

 Table 6-10
 Predefined numeric format keywords (continued)

Keyword	Description	Example/Result	
Yes/No	Returns No if zero, Yes otherwise.	Format(3434.2899, "Yes /No") Returns Yes	
True/False	Returns False if zero, True otherwise.	Format(3434.2899, "True /False")	
		Returns True	
On/Off	Returns Off if zero, On otherwise.	Format(3434.2899, "On /Off") Returns On	

 Table 6-10
 Predefined numeric format keywords (continued)

### **User-defined numeric formats**

A format expression for numbers can have up to four sections, separated by semicolons. You can use double quotation marks (") for adding literal strings and backslashes  $(\)$  to add literal characters.

Table 6-11 shows the symbols or characters you can use to create your own user-defined formats, and the effect of each symbol on the resulting format string.

Symbol	Description	Examples/Results
0	Zero-digit placeholder. If <exprs td="" to<=""><td>Format(3434.2899,</td></exprs>	Format(3434.2899,
	format> has fewer digits than	"000.000")
	<format pattern="">, the empty digits are filled with zeros. If <exprs td="" to<=""><td>Returns 3434.290</td></exprs></format>	Returns 3434.290
	format> has more digits to the right	Format(129.557,
	of the decimal than <format< td=""><td>"0000.00")</td></format<>	"0000.00")
	pattern>, the result is rounded to the number of places in <format pattern&gt;; if it has more digits to the left of the decimal, the extra digits are displayed without modification.</format 	Returns 0129.56
#	Null digit placeholder. If <exprs td="" to<=""><td>Format(434.2899,</td></exprs>	Format(434.2899,
	format> has fewer digits than the	"#,##0.0")
	format pattern, empty digits become Null, not filled with blanks.	Returns 434.3
	Therefore, the resulting string can	Format(434.2899,
	be shorter than the original format	"000###0.00")
	pattern.	Returns 000434.29
		(continues)

 Table 6-11
 Symbols and characters for creating custom numeric formats

Symbol	Description	Examples/Results
%	Percentage placeholder. Multiplies	Format(.75, "##%")
	<exprs format="" to=""> by 100 and appends a % character.</exprs>	Returns 75%
		Format(.75, "0##.000%")
		Returns 075.000%
	Decimal placeholder. Indicates	Format(3434.28, "0.0")
	where to place the decimal point. If <locale> is specified, the decimal</locale>	Returns 3434.3
	separator character of the specified	Format(3434.28, "0.00")
	current run-time locale is used.	Returns 3434.28
,	Thousands separator. If <locale> is specified, the thousands separator character of the specified locale is used. Otherwise, the current run- time locale is used.</locale>	Format(3434.2899, "#,##0.00")
		Returns 3,434.29
E- or e-	Returns <exprs format="" to=""> in scientific notation. You must place a digit placeholder (0 or #) to the immediate left of the symbol. Place an appropriate number of digit placeholders to the right of the</exprs>	Format(1500000,
		"O.OE-")
		Returns 1.5E6
		Format(1500000,
		"#0.0E-")
	symbol to display the exponent. A minus sign is displayed if the result has a negative exponent.	Returns 15.0E5
E+ or e+	Same as E The plus sign causes the polarity of the exponent to display	Format(1500000,
		"O.OE+")
	whether positive (+) or negative (-).	Returns 1.5E+6
		Format(15000, "#0.0e+")
		Returns 15.e+3

# Table 6-11Symbols and characters for creating custom numeric<br/>formats (continued)

Symbol	Description	Examples/Results
\$	Currency symbol. Displays the currency symbol based on the specified locale.	<pre>Format(total,     "(\$)#,##0.00","de_DE"     or</pre>
		Format(total, "€#,##0.00","de_DE")
		Returns € 8.790,00
		Format(total, "(\$)#,##0.00", "en_US")
		Returns \$ 8,790.00
-, (, ), [space], +	Literal characters that do not affect the format of the expression. [space] is Chr\$(32).	Format\$(5551212, "##0-0000")
		Returns 555-1212
		Format\$(1111111111, "(###) ##0-0000")
		Returns (111) 111-1111
		Format\$(150,"+##0")
		Returns +150

Table 6-11Symbols and characters for creating custom numeric<br/>formats (continued)

### Format[\$] with Format sections

Unless <format pattern> specifies one of the predefined format keywords, a format expression for numbers can have up to four sections, separated by semicolons. If you use semicolons with nothing between them, the missing section is printed using the format of the positive value.

The examples in Table 6-12 assume settings in the user's Control Panel are for the United States.

Table 6-12	Working with format sections in numeric formats
------------	---

Use	Description	Example/Result
1 section	<format pattern=""> applies to</format>	Format\$(4.15, "\$#,##0.00")
all values.	all values.	Returns \$4.15
		Format\$(-4.15, "\$#,##0.00")
		Returns -\$4.15
		(continues)

Use	Description	Example/Result
2 sections	(1) Applies to positive values and zero.	Format\$(-7, "\$#,##0.00;\$(#,##0.00)")
	(2) Applies to negative values.	Returns \$(7.00)
		Format\$(-7, "\$#,##0.00; \L\o\s\s:#,##0.00")
		Returns Loss: 7.00
3 sections	<ol> <li>(1) Applies to positive values.</li> <li>(2) Applies to negative values.</li> <li>(3) Applies to zero.</li> </ol>	<pre>Format\$(0, "#,###;(#,###);     \Z\e\r\o!")</pre>
		Returns Zero!
4 sections	<ol> <li>(1) Applies to positive values.</li> <li>(2) Applies to negative values.</li> <li>(3) Applies to zero.</li> </ol>	<pre>Format\$(-25, "#;(#);\Z\e\r\     o;\N\i\l")</pre>
		Returns (25)
	(4) Applies to Null.	Format\$(25-25, "#;(#); \Z\e\r\o;\N\i\l")
		Returns Zero
		<pre>Format\$(Null, "#;(#);     \Z\e\r\o;\N\i\l")</pre>
		Returns Null

 Table 6-12
 Working with format sections in numeric formats (continued)

### Using Format[\$] for date data

Numbers can be used to represent date and time information. You can format date and time serial numbers using either date-and-time formats or numeric formats because date/time serial numbers are stored as floating point values.

To format date and time, use one of the following methods:

- Use the format keywords that have been predefined in Actuate Basic.
- Create your own user-defined formats using standard characters or symbols that have special functional meanings when used in a format expression.

If <locale> is specified, symbols are formatted according to formatting rules of the specified locale. Otherwise, symbols are formatted according to the formatting rules of the current run-time locale.

Table 6-13 shows the predefined date-and-time format keywords you can use and the effects of each on the resulting format pattern.

Keyword	Description	Example/Result
"General date"	Returns a date and time.	<pre>Format\$(30267.83681, "General     date")</pre>
		Returns 11/12/82 8:05:00 PM
"Long date"	Returns a Long Date as defined in the user's Control Panel.	Format\$(30267.83681, "Long date")
		Returns Friday, November 12, 1982
"Medium date"	Returns a date with the month name abbreviated to 3 letters.	Format\$(30267.83681, "Medium date")
		Returns 12-Nov-82
"Short date"	Returns a Short Date as defined in the user's Control Panel.	Format\$(30267.83681, "Short date")
		Returns 11-12-82
"Long time"	Returns a Long Time as defined in the user's Control Panel. Includes hours, minutes, and seconds.	Format\$(0.83681, "Long time")
		Returns 8:05:00 PM
"Medium time"	Returns hours and minutes in 12-hour format, including the A.M./P.M. designation.	<pre>Format\$(0.83681, "Medium    time")</pre>
		Returns 8:05 PM
"Short time"	Returns hours and minutes in 24-hour format.	Format\$(0.83681, "Short time")
		Returns 20:05

 Table 6-13
 Predefined data-and-time format keywords

### **User-defined date formats**

You can use double quotation marks (") for adding literal strings and backslashes (\) to add literal characters. Table 6-14 shows the symbols or characters that you can use to create your own user-defined date formats, and the effect of each symbol on the resulting format string.

In all cases, assume DateVar = CVDate("11/12/82") or DateVar = CVDate("9/9/82") to demonstrate use of leading zeros, unless otherwise noted.

If <locale> is specified, the results are formatted according to the format of the specified locale. Otherwise, the results are formatted according to the formatting rules of the current run-time locale.

Note that the symbol for minute is n, not m, which is the symbol for month.

Symbol	Description	Example/Result
/,-	Date separator	Format\$(DateVar, "mm/dd/yy")
		Returns 11/12/82
		Format\$(DateVar, "mm-dd-yy")
		Returns 09-09-82
d	Day of the month without	<pre>Format\$(DateVar, "/d/")</pre>
	leading zero (1-31)	Returns /12/
		Format\$(DateVar, "/d/")
		Returns /9/
dd	Day of the month with leading	Format\$(DateVar, "/dd/")
	zero if needed (01-31)	Returns /12/
		Format\$(DateVar, "/dd/")
		Returns /09/
ddd	Three-letter abbreviation for day	Format\$(DateVar, "ddd")
	of the week	Returns Fri
dddd	Full name of day of the week	Format\$(DateVar, "dddd, mm/dd")
		Returns Friday, 11/12
ddddd	Short Date string, as defined in the user's Control Panel	Format\$(DateVar, "ddddd")
		Returns 11/12/82
ddddd	Long Date string, as defined in the user's Control Panel	Format\$(DateVar, "dddddd")
		Returns November 12, 1982
W	Day of the week as a number (Sunday = 1, Saturday = 7)	Format\$(DateVar, "w")
		Returns 6
ww	Week of the year as a number (1-	Format\$(DateVar, "ww")
	53). The week begins on a Sunday.	Returns 46

 Table 6-14
 Symbols to use to create custom date-and-time formats

0,11001	Description	Example/Result
www	Custom week of the year as a	Format\$(DateVar, "www")
	number. The date you specify determines the day on which the week begins.	Returns 46
m	Number of the month without	Format\$(DateVar, "m")
	leading zero (1-12)	Returns 11
mm	Number of the month with	Format\$(DateVar, "mm")
	leading zero (01-12)	Returns 11
		<pre>Format\$(DateVar, "mm")</pre>
		Returns 09
mmm	Three-letter abbreviation for	Format\$(DateVar, "mmm")
	month name	Returns Nov
mmmm	Full name of the month	Format\$(DateVar, "mmmm")
		Returns November
q	Number of the quarter (1-4)	Format\$(DateVar, "q")
		Returns 4
		"Quarter" & Format\$(DateVar, "q")
		Returns Quarter 4
qq	Number of the year half (1-2)	Format\$(DateVar, "qq")
		Returns 2
у	Number of the day of the year	Format\$(DateVar, "y")
	(1-366)	Returns 316
		Val(Format\$(DateVar, "y") + 19)
		Returns 335
уу	Last two digits of the year (00-99)	Format\$(DateVar, "yy")
		Returns 82
уууу	All four digits of the year (100-9999)	Formats (DateVar "MARAN")
		Returns 1982
C	Date variant as ddddd ttttt	Formatš(DateVar. "c")
~		Returns 11/12/82
q qq y yy yyyy с	Number of the quarter (1-4) Number of the year half (1-2) Number of the day of the year (1-366) Last two digits of the year (00-99) All four digits of the year (100-9999) Date variant as ddddd ttttt	Returns November Format\$(DateVar, "q") Returns 4 "Quarter" & Format\$(DateVar, "q") Returns Quarter 4 Format\$(DateVar, "qq") Returns 2 Format\$(DateVar, "qq") Returns 316 Val(Format\$(DateVar, "y") + 19) Returns 335 Format\$(DateVar, "yy") Returns 82 Format\$(DateVar, "yyyy") Returns 1982 Format\$(DateVar, "c") Returns 11/12/82

 Table 6-14
 Symbols to use to create custom date-and-time formats

# Using Format[\$] with time data

The fractional part of a number that represents a date represents a time of day on that date.

For example, the CVDate function converts numbers to date variants. CVDate(30632) is 11/12/83. No time of day is returned, because 30632 has no fractional part. By contrast, CVDate(30632.83) returns 11/12/83 7:55:12 P.M., because 30632.83 does have a fractional part, and the fractional part, .83, corresponds to about 7:55 P.M. (0.83 hours \* 24 = 19.92 hours from midnight, or 7:55:12 P.M.).

Table 6-15 assumes DateVar = "11/12/82 20:05:07". Format[\$] returns times returned by Format[\$] in 24-hour format unless you use one of the A.M./P.M. format symbols.

Note that the symbol for minute is n, not m, which is the symbol for month.

Symbol	Description	Example/Result
:	Time separator	<pre>Format\$(DateVar, "hh:nn:ss")</pre>
		Returns 20:05:00
h	Hour without leading zero (0-23)	Format\$(DateVar, "h:nn AM/PM")
		Returns 8:05 PM
hh	Hour with leading zero (00-23)	Format\$(DateVar, "hh:nn AM/PM")
		Returns 08:05 PM
n	Minute without leading zero (0-59)	Format\$(DateVar, "h:n AM/PM")
		Returns 8:5 PM
nn	Minute with leading zero (00-59)	Format\$(DateVar, "hh:nn AM/PM")
		Returns 08:05 PM
S	Second without leading zero (0-59)	Format\$(DateVar, "h:n:s AM/PM")
		Returns 8:5:7 PM
SS	Second with leading zero (00-59)	Format\$(DateVar, "h:nn:ss AM/PM")
		Returns 8:05:07 PM

 Table 6-15
 Symbols for using time data with Format[\$]

Symbol	Description	Example/Result
AM/ PM or am/pm	Designation AM/am for any hour before noon and PM/pm for any hour after. Case- sensitive.	Format\$(DateVar, "hh:nn:ss am/pm") Returns 08:05:07 pm
A/P or a/p	Designation A/a for any hour before noon and P/p for any hour after. Case-sensitive.	Format\$(DateVar, "hh:nn:ss A/P") Returns 08:05:07 P
AMPM	Uses formats set by 11:59 and 23:59 as defined in the user's Control Panel. Default is AM/PM.	Format\$(DateVar, "hh:nn AMPM") Returns 08:05 PM
ttttt	Uses format set by Time as defined in the user's Control Panel. Default is h:nn:ss.	Format\$(DateVar, "h:nn ttttt") Returns 8:05 PM

 Table 6-15
 Symbols for using time data with Format[\$]

### Using Format[\$] with string data

You can format strings with Format[\$].

Table 6-16 shows the symbols or characters you can use to create your own userdefined formats, and the effect of each symbol on the resulting format string.

Symbol	Description	Example/Result
@	Character placeholder. Returns a character, or a space if there is no	Format\$("5101212111", "(@@@)@@@-@@@@@")
	character in the corresponding position. Placeholders fill from	Returns (510) 121-2111
	right to left unless the format pattern uses a ! character.	Format\$("1212111", "(@@@)@@@-@@@@@")
		Returns ( ) 121-2111
&	Character placeholder that does not return a space if the format	Format\$("61767899999", "(&&&)&&&-&&&&")
pattern contains i than there are cha source value. Pla right to left unles pattern uses a ! cl	pattern contains more placeholders than there are characters in the data	Returns (617) 789-9999
	source value. Placeholders fill from right to left unless the format pattern uses a ! character.	Format\$("6789999","(&&&) &&&-&&&&")
		Returns () 678-9999
		(continues)

 Table 6-16
 Symbols for using string data with Format[\$]

Symbol	Description	Example/Result
<	Lowercase conversion. Converts an expression to lowercase.	<pre>Format\$   ("Smith, John","&lt;")</pre>
		Returns smith, john
>	Uppercase conversion. Converts an expression to uppercase.	Format\$ ("Smith, John",">")
		Returns SMITH, JOHN
!	Specifies that placeholders fill from left to right.	Format\$("5551212", "!(&&&)&&&-&&&& + ext")
		Returns (555) 121-2 + ext
		Format\$("5551212", "!(@@@)@@@-@@@@@ + ext")
		Returns (555)121-2 + ext

 Table 6-16
 Symbols for using string data with Format[\$] (continued)

**Example** The following example displays a random number in various formats:

```
Sub Start()
  Dim FormatData As Integer, Msg as String
  Super::Start( )
  ' Generate random number to format
  FormatData = Int( 100000 * rnd )
  ' Display number formatted in various ways
  Msg = "Format result for format value #0: "
     & Format( FormatData, "#0" )
+
  ShowFactoryStatus ( Msg )
  Msg = "Format result for format value 000000: "
     & Format( FormatData, "000000" )
+
  ShowFactoryStatus ( Msq )
  Msg = "Format result for format value #, ##0: "
     & Format( FormatData, "#,##0" )
+
  ShowFactoryStatus ( Msq )
  Msg = "Format result for format value #,##0.00: "
     & Format( FormatData, "#,##0.00" )
+
  ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also CVDate function DateValue function

ParseDate function Str, Str\$ functions

# For...Next statement

Repeats a block of instructions a specified number of times.

**Syntax** For <counter variable> = <start> **To** <end> [ Step <step size> ]

[<statement block>]

[Exit For]

[<statement block>]

Next [<counter variable >]

### Parameters <counter variable>

Numeric variable that keeps track of the number of times the set of statements between For and Next has been executed.

### <start>

Numeric expression indicating the first number at which <counter variable> is to begin counting. The initial value of <counter variable>.

### <end>

Numeric expression indicating the number at which <counter variable> is to stop counting; the final value of <counter variable>. As soon as <counter variable> stops execution, Actuate Basic skips everything between the For and Next keywords, and transfers control to the statement following Next.

### Step

Keyword that specifies what number to add to <counter variable> every time after the loop finishes executing. The sum of <counter variable> and <step size> returns the next, new value for <counter variable>.

### <step size>

The number the program adds to <counter variable>, at the end of every trip through the loop, in order to determine the next value of <counter variable>.

Default: 1

### <statement block>

Any valid Actuate Basic statements.

### Exit For

Keyword that causes Actuate Basic to terminate the loop immediately, and to transfer control to the statement following Next. Exit For statement can be placed anywhere in the loop any number of times, although only one will be executed.

### Next

Keyword that defines the completion of a For...Next statement, and causes the program to add <step size> to <counter variable>.

Default: If you omit <counter variable> on the Next line, Actuate Basic adds the value of <step size> to the <counter variable> associated with the most recent For statement.

**Description** First, Actuate Basic sets <counter variable> to <start>.

 Table 6-17 summarizes what Actuate Basic does next, which depends upon the sign of <step size>.

<step size=""></step>	Loop executes when	Stops execution when
Positive	<counter variable=""> &lt;= <end></end></counter>	<counter variable=""> &gt; <end></end></counter>
Negative	<counter variable=""> &gt;= <end></end></counter>	<counter variable=""> &lt; <end></end></counter>

Table 6-17 Execution of For...Next statements with <step size>

- If Actuate Basic executes the loop, it adds <step size> to <counter variable> at the line starting with Next.
- Then Actuate Basic compares this new value of <counter variable> to <end>.
- If the conditions described in the table are satisfied, Actuate Basic executes the loop again.
- Otherwise, Actuate Basic skips everything between the For and Next keywords and transfers control to the statement immediately following Next.
- **Rule** If a For...Next loop is opened by its For clause and closed by its corresponding Next clause, nested loops must be closed in the reverse order as shown in the following example:

```
For I = 1 To 10

For J = 1 To 10

For K = 1 To 10

L = I + J + K

Next K

Next J

Next I
```

When you nest For...Next statements, you must use a different counter variable for each block. Traditionally, the letters I, J, K, and so forth, declared as Integers, are used as counter variables.

**Tips** To avoid confusion when debugging, do not change the value of <counter variable> from inside the loop.

- To make your program more efficient, dimension <counter variable> as an Integer.
- To evaluate a certain condition and then determine whether or not a loop continues, use Exit For within a conditional structure like If...Then...Else.
- To ascertain or set the contents of a multidimensional array, use nested For...Next loops.
- **Example** The following example uses two nested loops to put seven lines of the uppercase alphabet into a message:

```
Sub Start( )
Dim I As Integer, Rep As Integer
Super::Start( )
' Perform seven repetitions
For Rep = 7 To 1 Step -1
' Convert alpha to numeric values
For I = Asc("A") To Asc("Z")
' Convert back, append letters to string
ShowFactoryStatus( Chr$( I ) )
Next I
' Add newline after each repetition
Next Rep
End Sub
```

See also Do...Loop statement Exit statement While...Wend statement

# **FreeFile function**

Returns the next unused system file number.

- Syntax FreeFile
- Returns Integer

When you open a file, you must supply a number by which to reference it throughout the program. This number must not already be in use by another currently open file. FreeFile eliminates the need for you to keep track of which file numbers are currently in use by finding the next available number.

**Tip** The number that FreeFile returns does not change until you actually open a file with it. So to be sure FreeFile is always current, immediately open a file with the number it gives you, instead of waiting until later to do so.

**Example** The following example assigns the return value of FreeFile to the variable FileNum%, then uses FileNum to open the file, Test.dat:

FileNum% = FreeFile
Open "Test.dat" for Output As FileNum%

The following example creates a test file and opens it to determine an unused file number:

```
Sub Start()
  Dim FileNumber As Integer, Msg As String
  Super::Start( )
  ' Open a file
  Open "Test1.fil" For Output As #1
  ' Get unused file number
  FileNumber = FreeFile
  ' Create a test file
  Open "Test2.fil" For Output As FileNumber
  ' Close all files
  Close
  Msq = "The file number FreeFile returned was: " & FileNumber
  ShowFactoryStatus ( Msg )
  Msg = "The files will now be deleted."
  ShowFactoryStatus ( Msg )
  ' Delete files from disk
  Kill "Test1.fil"
  Kill "Test2.fil"
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Open statement

# **Function...End Function statement**

Declares and defines the name, code, and arguments that constitute an Actuate Basic function, and returns a value to the calling expression or statement.

Syntax [Static] Function <name of function> [(<list of arguments>)] [As <data type>]

[<statements>]

[Exit Function]

[<statements>]

<name of function> = <return value>

End Function

### Parameters Static

Keyword that instructs Actuate Basic to preserve the values of the procedure's local variables between calls.

Rules:

- You cannot use Static to affect variables declared outside the procedure.
- You should avoid using Static in recursive procedures.

### <name of function>

The name you assign to the procedure.

Rules for <name of function>:

- Subject to the same constraints as variable names, as well as to the following additional constraints:
  - Can include a type declaration character.
  - Cannot be the same as any other name recognized at the global or module level, such as that of:
    - A procedure in a declared dynamic-link library (DLL)
    - A variable
    - A constant
- Can be overloaded. That is, you can define another function or procedure that has the same name, as long as the respective arguments are unique. For example, the following are both permissible in the same program:

```
Sub Potato(intTomato As Integer)
...
End Sub
Sub Potato(dblTomato As Double, strEggplant As String)
...
End Sub
```

### <list of arguments>

The variable name or names that the function uses internally to store and manipulate the values (arguments) that you pass to it from the calling statement.

Each of the <variable name> components of the <list of arguments> corresponds to a value or variable that is in the same relative position in the calling statement. It does not matter what you call the variable that is passed to the function; the function will process the value it receives as an argument. The <statements> in the procedure process arguments it receives to produce the single value that is returned to the calling procedure.

In the following example, the function Square expects to be supplied with a single argument, a numeric expression, which it squares:

```
Function Square(X)
   Square = X ^ 2
End Function
```

The following statements will pass the value 4 to the function Square and receive back the value 16 after Square has completed. The statement will then continue; it will multiply the value it received from the function Square times 2 and print the resulting value.

```
MyNumber = 4
Print #1, Square(MyNumber) * 2
```

Here are the steps in the process:

- 1 First, the value 4 is assigned to the variable MyNumber.
- **2** The value of MyNumber is passed to the function Square. Square assigns the value it receives from the calling statement to the variable X.
- **3** Square processes X and assigns the resulting value to the name of the function itself—Square—and returns control to the statement that called it.
- **4** The calling statement continues processing the new value of MyNumber (16); it multiplies it by 2, and prints the result (32).

### <statements>

One or more valid Actuate Basic statements. These statements constitute the body of the function procedure.

### <return value>

The value the function returns to the calling statement after the function has completed. Usually, <return value> is a variable that contains the final result of all the changes the argument or arguments underwent in the course of the execution of the <statements>.

### **Exit Function**

Keyword that signals Actuate Basic to terminate the function procedure and transfer control to the statement following the one that called the procedure. You can use multiple Exit Function statements, anywhere in a function procedure, but only one is executed.

list of arguments> has the following syntax:

[**ByVal**] <variable name> [**As** <data type>] [,[**ByVal**] <variable name> [**As** <data type>]...

### ByVal

Keyword that instructs Actuate Basic to pass the argument to the procedure by value rather than by reference, so that whatever change the function makes to the argument variable has no affect on its original value in the calling procedure.

Rule for ByVal: Cannot be used with a variable of user-defined type, object type, or with an array variable.
#### <variable name>

Name of the variable to pass as an argument. If the function changes the value of <variable name> internally, then it also changes its value externally.

Rules for <variable name>:

 You must use ByVal if you do not want the function's changes to an argument variable to affect the variable's value in the calling statement.
 Example:

Function SquareRoot (ByVal UserNum)

• For array variables, use the parentheses but omit the number of dimensions. Example:

Function SalesTax (MyArray())

 If <variable name> is undeclared within the function but has the same name as another procedure, a Global or module-level constant or variable, or an object, Actuate Basic assumes your function refers to that external name. This can cause name conflicts. To avoid such conflicts, explicitly declare each <variable name> within the function.

#### As <data type>

Clause that declares the data type of <variable name>. You can also specify the data type of <returned value> by appending this clause after the <list of arguments>.

Rule for As <data type>: <data type> can specify any valid Actuate Basic or userdefined data type except fixed-length String.

- **Examples** Function SalesTax(Customer As String, Amount As Currency) Function SalesTax(ByVal Customer As String, Amount)
- **Description** Like a Sub statement, a Function can take arguments, execute a series of statements, and change the values of its arguments. However, unlike a Sub statement, a Function can also return a value directly and can be used in an expression.

You can use the same name for two or more different procedures, as long as you make sure the procedures take a different number of arguments, or that the arguments are of different data types. For example, you can write two different square root functions—one that operates on integers, and another that operates on doubles. Their respective first lines might look like the following:

Function SquareRoot(intParam As Integer) As Integer Function SquareRoot(dblParam As Double) As Double

Actuate Basic will know which procedure you mean by the type of value you pass when you call the procedure. For instance, if you write a call to SquareRoot(5), the compiler will choose the first SquareRoot function. But if you call SquareRoot(5.1234567), or SquareRoot(5.000), it will execute the second one.

Rules:

- You cannot define a Function statement from within another Function statement.
- You cannot use GoTo to enter or exit a Function statement.
- **Tips** Functions can be recursive. Actuate Basic sets a run-time stack limit of 200. A report using recursion with a large number of iterations might exceed this limit.
  - To avoid possible name conflicts between <variable name> and the names of other variables, constants, procedures, or objects, explicitly declare all variables within the function with Dim, ReDim, or Static. That way, you can use a common variable name like SaleDate within a function procedure, even if there is a global variable named SaleDate. Name conflicts can arise only when you use a variable in a procedure that you did not explicitly declare within the procedure.
  - To distinguish between a variable and a function with the same name in your code, use the function's name with parentheses and the variable's name without parentheses. Where the function takes no parameters, append an empty pair of parentheses. For example:

```
' Assign global variable Homonym to CopyVar1
CopyVar1 = Homonym
' Assign return value of function Homonym to CalculatedVar2
CalculatedVar2 = Homonym()
```

- To evaluate a condition and then determine whether or not the function procedure should continue, use Exit Function within a conditional structure like If...Then...Else.
- To guard against the wrong data type being passed to or returned from a function, use As <data type> clauses.
- **Example** The following example generates a number, then calls a user-defined function that returns the number's square root.

To use this example, paste the function SquareRoot after the End Sub of the procedure or save it in your Actuate code module (.bas) file.

```
Sub Start( )
Dim Msg As String, NumRoot
Dim UserNum As Double
Super::Start( )
UserNum = 255 * Rnd + 1
' Call the SquareRoot function here to process UserNum
NumRoot = SquareRoot(UserNum)
' NumRoot has now been set to the square root of UserNum
Msg = "The square root of " & UserNum
```

```
Select Case NumRoot
     Case 0: Msg = Msg & " is zero."
     Case -1: Msg = Msg & " is an imaginary number."
     Case Else: Msg = Msg & " is " & NumRoot
  End Select
  ShowFactoryStatus ( Msg )
End Sub
' This function finds the square root of a number. It takes one
' input argument X (data type Double) and returns a value that
' is also data type Double.
Function SquareRoot (X As Double) As Double
  ' Sqn returns 1 if a number is positive, -1 if it's negative,
  ' and 0 if it's zero. This fact is important here since we
  ' only want to operate on a number greater than zero
  ' Evaluate sign of argument
  Select Case Sqn(X)
     Case 1
        SquareRoot = Sqr(X)
        ' OK if positive, so exit
       Exit Function
     Case 0
        SquareRoot = 0
     Case -1
        SquareRoot = -1
  End Select
End Function
```

### See also Dim statement Static statement Sub...End Sub statement

# **FV** function

Returns the future value of an annuity based on periodic, constant payments, and on an unvarying interest rate.

Syntax FV(<rate per period>,<number pay periods>,<each pmt>,<present value>, <when due>)

#### **Parameters** <rate per period> Numeric expression that specifies the interest rate that accrues per period.

Rule for <rate per period>: Must be given in the same units of measure as <number pay periods>. For instance, if <number pay periods> is expressed in months, then <rate per period> must be expressed as a monthly rate.

#### <number pay periods>

Numeric expression that specifies the total number of payment periods in the annuity.

Rule for <number pay periods>: Must be given in the same units of measure as <rate per period>. For instance, if <rate per period> is expressed as a monthly rate, then <number pay periods> must be expressed in months.

#### <each pmt>

Numeric expression that specifies the amount of each payment.

Rule for <each pmt>: Must be given in the same units of measure as <rate per period>. For instance, if <rate per period> is expressed in months, then <each pmt> must be expressed as a monthly payment.

#### <present value>

Numeric expression that specifies the value today of a future payment, or stream of payments.

Example for cyresent value>: If you put \$23.94 in the bank today and leave it there for 15 years at an interest rate of 10% compounded annually, you end up with about \$100. The present value of \$100 is approximately \$23.94.

#### <when due>

Numeric expression that specifies whether each payment is made at the beginning (1), or at the end (0) of each period.

Default 0

Rule for <when due>: Must be 0 or 1.

**Examples** The following example assumes you deposit \$10,000 in a savings account for your daughter when she is born. If the account pays 5.7% compounded daily, how much will she have for college in 18 years? The answer, \$27,896.60, is assigned to the variable TotalValue.

TotalValue = FV(0.057/365, 18\*365, 0, -10000, 1)

The following example is almost the same as the previous one. In this one, however, assume that the interest is compounded monthly instead of daily, and that you have decided to make an additional monthly deposit of \$55 into the account. The future value assigned to TotalValue in this case is \$48,575.82.

TotalValue = FV(0.057/12, 18\*12, -55, -10000, 1)

#### Returns Double

An annuity is a series of cash payments, constant in value, made over a period of time. An annuity can be an investment, such as a monthly savings plan, or a loan,

such as a home mortgage. The future value of an annuity is the cash balance you want after you have made your final payment.

- **Examples** You set up a savings plan with a goal of having \$75,000 in 18 years to pay for your child's education. For this plan, the future value is \$75,000.
  - You take out a loan for \$11,000. The future value is \$0.00, as it is for any typical loan.
  - - You must express cash paid out, such as deposits to savings, using negative numbers, and cash received, such as dividend checks, using positive numbers.
- **Example** The following example defines an amount to save or invest each month, the annual percentage rate (APR) of the interest, the total number of payments, and prompts for when during the month payments will be made. Then it tells the user what the future value of such a savings plan or investment will be.

To use the following example, paste the Declare section into a source code (.bas) library file:

```
Declare
  Global Const ENDPERIOD = 0
  Global Const BEGINPERIOD = 1
End Declare
Sub Start()
  Dim EachPmt As Double, APR As Double
  Dim TotalPmts As Double, PayWhen As Integer
  Dim PresentVal As Double, FutureVal As Double
  Dim Msg as String, Fmt As String
  Super::Start( )
  ' Specify money format
  Fmt = "$###, ###, ##0.00"
  ' Amount to save each month
  EachPmt = 2000
  ' The annual percentage rate for the interest
  APR = 0.0325
  ' The number of months to save
  TotalPmts = 60
   ' Assume payment at month's end. Change to BEGINPERIOD
   ' for payment at month beginning
  PayWhen = ENDPERIOD
   ' Amount in the savings account now
  PresentVal = 12000
  ' Now do the real work
  FutureVal =
     FV(APR / 12, TotalPmts, -EachPmt, -PresentVal, PayWhen)
+
```

' Format the resulting information for the user Msg = "Starting with " & Format( PresentVal, Fmt ) + & " and saving " & Format( EachPmt, Fmt ) + & " every month at an interest rate of " + & Format( APR, "##.00%" ) & " for a period of " + & TotalPmts & " months will give you " + & Format( FutureVal, Fmt ) & "." ShowFactoryStatus( Msg ) End Sub

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also IPmt function NPer function Pmt function PV function Rate function

### Get statement

Reads data from a disk file into a variable.

Syntax Get [#] <open file number>, [<record number>], <receiving variable>

### Parameters <open file number>

Numeric expression that is the number you assigned as a descriptor for the target file when you opened it.

#### <record number>

Numeric expression between 1 and 2,147,483,647:

- For files opened in Random mode, <record number> is the number of the record to be read.
- For files opened in Binary mode, it is the byte position at which reading starts, where the first byte in a file is at position 1, the second byte at position 2, and so on.

Default: The next record or byte—that is, the one following the last Get or Put statement, or the one pointed to by the last Seek function.

Rule for <record number>: If you omit <record number> you must still include the delimiting commas that would have been on either side of it had you included it.

### <receiving variable>

String expression that specifies the variable that is to receive input from the open file.

Rules for <receiving variable>:

- Cannot be an object variable, user-defined data type structure, handle to a class, CPointer, or Java object.
- Cannot be an array variable that refers to an entire array, although you can use a variable that denotes a single element of an array.
- **Description** You can use the Get statement to read from files opened only in random or binary mode. You cannot use the Get statement to read from files opened in input mode. The behavior of Get varies depending upon the mode in which the file was opened.

### For files opened in random mode

If the length of the data you read is less than the length specified in the Len clause of the Open statement, the next Get skips the data you did not read and reads subsequent records starting at the record-length boundary, specified by Len.

**Example** The first Get stores bytes 1-4 of the disk file Myfile.txt into the variable HoldingVar1. The second Get skips to the record-length boundary (20) and starts counting from there, which means it stores bytes 21-25 into HoldingVar2. The intervening bytes (5-20) are ignored.

```
Dim HoldingVar1 As String
Dim HoldingVar2 As String
HoldingVar1 = "1234"
HoldingVar2 = "1234"
Open "Myfile.txt" for Random As #1 Len = 20
Get #1, , HoldingVar1
Get #1, , HoldingVar2
```

Table 6-18 summarizes how Get behaves depending on the data type of <receiving variable>. Open here refers to the particular statement that opened the file in question.

	-		
<receiving variable&gt;</receiving 	G	et reads, in order	Rules
Variable-length string	1 2	2-byte descriptor containing string length Data that goes into <receiving variable=""></receiving>	Record length specified in the Len clause of Open must be at least 2 bytes greater than the actual length of the string.
			(continues)

Table 6-18Execution of Get statements with <receiving variable> data<br/>types

<receiving variable&gt;</receiving 	Ge	et reads, in order	Rules
Numeric Variant (Variant Types 0-7)	3	2-byte descriptor identifying VarType of the Variant	Len clause length must be at least 2 bytes greater than the actual number of bytes
	4	<receiving variable=""></receiving>	required to store the variable.
String Variant (Variant Type 8)	5	2-byte VarType descriptor	Len clause length must be at least 4 bytes greater than the
	6	2 bytes indicating the string length	actual length of the string.
	7	Data that goes into <receiving variable=""></receiving>	
Any other type of variable	8	Data that goes into <receiving variable=""></receiving>	Len clause length must be greater than or equal to the length of the data.

**Table 6-18**Execution of Get statements with <receiving variable> data<br/>types (continued)

### For files opened in binary mode

All the Random descriptions and rules above apply except that the Len clause in the Open statement has no effect.

**Example** The first statement stores the first 4 bytes of the disk file Myfile.txt into the variable HoldingVar1. The second stores the next 5 bytes into HoldingVar2. No data is ignored.

```
Dim HoldingVar1 As String
Dim HoldingVar2 As String
Open "Myfile.txt" for Binary As #1
Get #1, , HoldingVar1
Get #1, , HoldingVar2
```

For variable-length strings that are not elements of user-defined types, Get does not expect a 2-byte descriptor to tell it how many bytes to read; it reads the number of bytes equal to the number of characters already in the string.

**Example** The following statements reads 12 bytes from file #1:

```
VariLen$ = String$(12, "*")
Get #1, , VariLen$
```

- **Tip** To avoid corrupting or misreading data, be sure your record lengths and variable lengths match the lengths of the data you want to read or write.
- **Example** The following example prompts the user for three customer names. It writes each name to a test file and then reads the names back.

```
Sub Start()
  Dim CustName As String
  Dim I As Integer, Max As Integer, Msg As String
  Super::Start( )
  ' Create a sample data file
  Open "Testfile.dat" For Random As #1 Len = 50
  ' Put records into file on disk
  Put #1, 1, "Customer 1"
  Put #1, 2, "Customer Two"
  Put #1, 3, "Third Customer"
  Close #1
   ' Close the file we've written and open again for reading
  Open "Testfile.dat" For Random As #1 Len = 50
  ' Calc total # of records
  Max = LOF(1) \setminus 50 + 1
  ' Read file backwards
  For I = Max To 1 Step -1
     ' Seek statement used
     Seek #1, I
     ' Get record at that position
     Get #1, , CustName
     Msg = "Record #" & ( Seek2( 1 ) - 1 ) & " contains: "
        & CustName
+
     ShowFactoryStatus ( Msg )
  Next I
  ' Close test file
  Close #1
  Msg = "Now removing file from disk."
  ShowFactoryStatus ( Msg )
  ' Delete file from disk
  Kill "Testfile.dat"
End Function
```

See also Open statement Put statement Seek statement Type...End Type statement

## **GetAFCROXVersion function**

Returns either the integer or the decimal component of the version number of the AFC library (Afcnnnn.rox).

Syntax GetAFCROXVersion(<numpart>)

### Parameters <numpart>

Number or numeric expression that tells GetAFCROXVersion whether you want to know the integer part of the version number, or the decimal part.

Rule: Must be either 0 or 1.

Returns Integer

If <numpart> is 0, GetAFCROXVersion returns the integer (major) component of the version number.

If <numpart> is 1, GetAFCROXVersion returns the decimal (minor) component of the version number.

**Example** The following example determines both parts of the version number of the AFC library and puts the parts together, interpolating the decimal point where it belongs:

```
Dim AFCMajor As Integer, AFCMinor As Integer
Dim AFCTotal as String
AFCMajor = GetAFCROXVersion(0)
AFCMinor = GetAFCROXVersion(1)
' Put the two parts together and prefix some identifying text:
AFCTotal = "AFC Version: " & (AFCMajor) & "." & (AFCMinor)
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also GetAppContext function GetFactoryVersion function GetOSUserName function GetROXVersion function GetServerName function GetServerUserName function

# GetAppContext function

Indicates which Actuate application is currently running.

- Syntax GetAppContext
- Returns AppContext

You interpret the value that GetAppContext returns depending on whether GetAppContext is called from a client or a server machine at run time.

On a client, GetAppContext returns the context of the Actuate application being used—for example, e.Report Designer Professional.

On a server, GetAppContext returns the context of server, not the context of Management Console or e.Report Designer Professional.

Table 6-19 shows how Actuate maps the return value from GetAppContext to the corresponding Actuate application. Other values refer to unsupported products.

 Table 6-19
 Relationships of return values from GetAppContext to Actuate applications

GetAppContext value	Application
DWBContext	e.Report Designer Professional (ERDPro)
ServerContext	Server
UnknownContext	An unknown context

**Example** The following example determines which Actuate application is running at the time you call GetAppContext:

```
Dim ApplicationContext As AppContext, MyValue As String
         MyValue = "Application Context = "
         ApplicationContext = GetAppContext
         Select Case ApplicationContext
            Case ServerContext
               MyValue = MyValue & "Server"
            Case DWBContext
               MyValue = MyValue & "e.Report Designer Professional"
            Case Else
               MyValue = MyValue & "unknown context"
         End Select
         For information about using the code examples, see "Using the code examples,"
         earlier in this chapter.
         GetAFCROXVersion function
See also
         GetFactoryVersion function
```

GetFactoryVersion function GetOSUserName function GetROXVersion function GetServerName function GetServerUserName function

# **GetAttr function**

Determines the attributes of a file, directory, or volume.

**Syntax** GetAttr(<file dir or vol name>)

#### Parameters <file dir or vol name>

String expression that specifies the file, directory, or volume for which you wish to determine what attributes are set.

Default path: Current drive and directory.

Rules:

- Must refer to a valid file, directory, or volume.
- Reference must be unambiguous.

<file dir or vol name> can optionally specify a full path, in which case it has the following syntax:

```
[<drive:>][ \ ]<directory>[\<directory>]...(Windows)
```

[/]<directory>[/<directory>]...(UNIX)

#### <drive:>

Character, followed by a colon, that specifies the drive (Windows only).

#### <directory>

String expression that specifies the name of a directory or subdirectory.

**Example** Assuming Testfile.dat has both the system and read-only attributes set, the following statement assigns the value 5 (4 + 1) to the variable ATTRIBS:

```
ATTRIBS = GetAttr("Testfile.dat")
```

- **Returns** Integer
  - The operating system assigns one or more attributes to each file, directory, and volume. These attributes indicate whether the item is a normal, read-only, hidden, or system file; a volume label; a directory; whether it has been modified since the last backup; or some combination of these.
  - If the file has multiple attributes, the return value will be the bitwise AND of all the values that apply. For example, a return value of 3 indicates a file is both read-only and hidden (1 + 2).
  - Table 6-20 shows certain constant names that are defined in Header.bas and their corresponding return values and attributes. The return value of GetAttr is the sum of one or more return values. The return values not supported under UNIX are 0.

Return value	Constant name	File attribute	<b>Operating system</b>
0	ATTR_NORMAL	Normal	Windows, UNIX
1	ATTR_READONLY	Read-only	Windows, UNIX
2 or 0	ATTR_HIDDEN	Hidden	Windows

 Table 6-20
 Header.bas constants with their return values and attributes

Return value	Constant name	File attribute	<b>Operating system</b>
4 or 0	ATTR_SYSTEM	System file	Windows
8 or 0	ATTR_VOLUME	Volume label	Windows
16	ATTR_DIRECTORY	Directory label	Windows, UNIX
32 or 0	ATTR_ARCHIVE	Changed since last backup	Windows

 Table 6-20
 Header.bas constants with their return values and attributes

**Tip** To determine—without having to do any arithmetic—whether a file has a particular, single attribute, use the BAnd operator to perform a bitwise test. If this statement evaluates to nonzero you know that, whatever other attributes it has, the file is read-only.

GetAttr("TESTFILE.DAT") BAnd ATTR READONLY

**Example** The following Windows example displays a message indicating the attributes of a given file:

```
Sub Start()
  Dim Attr As Integer, FName As String, Msg As String
  Super::Start( )
  On Error Goto ErrorHandler
  ' Set file name to check
  FName = "C:\Program Files\Actuate11\readme.rtf"
  ' Determine the file's attributes
  Attr = GetAttr(FName)
  Msg = "The value returned by GetAttr is: " & Attr
  ShowFactoryStatus ( Msg )
  ' Disregard the Archive attribute
  If Attr > 7 And Attr <> 16 Then
     If Attr BAnd ATTR ARCHIVE Then
       Attr = Attr BAnd BNot ATTR ARCHIVE
     Else
       Attr = Attr BOr ATTR ARCHIVE
     End If
  End If
  ' Correlate some attribute codes and code sums to messages
  Select Case Attr
     Case ATTR NORMAL
       Msg = "Normal"
     Case ATTR READONLY
       Msq = "Read-only"
     Case ATTR_HIDDEN
       Msg = "Hidden"
     Case ATTR HIDDEN + ATTR READONLY
       Msg = "Hidden and Read-only"
```

```
Case ATTR SYSTEM
       Msg = "System"
     Case ATTR READONLY + ATTR SYSTEM
       Msg = "Read-only and System"
     Case ATTR HIDDEN + ATTR SYSTEM
       Msg = "Hidden and System"
     Case ATTR READONLY + ATTR HIDDEN + ATTR SYSTEM
       Msg = "Read-only, Hidden, and System"
     Case ATTR DIRECTORY
       Msq = "Directory"
  End Select
  ShowFactoryStatus( UCase$(FName) & " is a " & Msg & " file.")
  Exit Function
  ErrorHandler:
     Msg = "Sorry, an error occurred. Please change the name or
  path "
     Msq = Msq & "of the file in this method code and try again."
     ShowFactoryStatus ( Msg )
End Sub
```

See also FileDateTime function FileLen function SetAttr statement

# **GetClassID** function

Returns the ID number that Actuate assigns automatically to all classes. Objects of the same class have the same ID number.

Syntax GetClassID(<object reference>, <offset>)

### Parameters <object reference>

The name of the variable that refers to the object for which you want the ID.

#### <offset>

An integer that indicates the level of the class hierarchy for which you want the ID. Must be 0 or a negative number. Specify 0 to return only the class ID, -1 to return the ID of the classes' superclass, and so on.

**Returns** Integer that is the class ID.

Null if <offset> is a positive number or a negative number higher than the class hierarchy.

- **Tips** Use GetClassID when you want to check if an object is of a given type without the overhead of a string compare.
  - To find out an object's class, use GetClassName.
  - To find out if an object is an instance of a specified class or is an instance of a subclass of a specified class, use IsKindOf.
- **Example** The following example displays the Class Name and Class ID of the current component, creates two new label components, then displays the Class Name and Class ID of each of the new components:

```
Sub Start()
  Dim ClassIDVar, ClassNameVar
  Dim MyLabel1 As AcLabelControl, MyLabel2 As AcLabelControl
  Dim Msg As String
  Super::Start( )
  ClassIDVar = GetClassID(Me, 0)
  ClassNameVar = GetClassName(Me, 0)
  Msq = "My Class ID is: " & ClassIDVar
  ShowFactoryStatus ( Msg )
  Msg = "My Class Name is: " & ClassNameVar
  ShowFactoryStatus ( Msg )
  Set MyLabel1 = New AcLabelControl
  Set MyLabel2 = New AcLabelControl
  Msg = "The Class Name for MyLabel1 is: "
     & GetClassName (MyLabel1)
+
  ShowFactoryStatus ( Msg )
  Msq = "Its Class ID is: "
     & GetClassID(MyLabel1)
+
  ShowFactoryStatus ( Msg )
  Msg = "The Class Name for MyLabel2 is: "
     & GetClassName(MyLabel2)
+
  ShowFactoryStatus ( Msg )
  Msg = "Its Class ID is: " & GetClassID(MyLabel2)
  ShowFactoryStatus ( Msg )
End Sub
```

See also GetClassName function IsKindOf function

### GetClassName function

Returns the name of the object's class.

Syntax GetClassName(<object reference>, <offset>)

### Parameters <object reference>

The name of the variable that refers to the object for which you want the class name.

### <offset>

An integer that indicates the level of the class hierarchy for which you want the class name. Must be 0 or a negative number. Specify 0 to return only the class name, -1 to return the name of the classes' superclass, and so on.

### Returns String

The name of the class. If the class is nested, the name is the fully qualified name, for example, MyReport::MyFrame::MyControl.

Null if <offset> is a positive number or a negative number higher than the class hierarchy.

- **Tips** Use GetClassName to test for a specific class before performing an action.
  - To find out if an object is an instance of a specified class or is an instance of a subclass of a specified class, use IsKindOf.
- **Example** The following example displays the Class Name and Class ID of the current component, creates two new label components, then displays the Class Name and Class ID of each of the new components:

```
Sub Start()
  Dim ClassIDVar, ClassNameVar
  Dim MyLabel1 As AcLabelControl, MyLabel2 As AcLabelControl
  Dim Msg As String
  Super::Start( )
  ClassIDVar = GetClassID( Me, 0 )
  ClassNameVar = GetClassName( Me, 0 )
  Msg = "My Class ID is: " & ClassIDVar
  ShowFactoryStatus ( Msg )
  Msg = "My Class Name is: " & ClassNameVar
  ShowFactoryStatus ( Msg )
  Set MyLabel1 = New AcLabelControl
  Set MyLabel2 = New AcLabelControl
  Msg = "The Class Name for MyLabel1 is: "
     & GetClassName(MyLabel1)
+
  ShowFactoryStatus ( Msg )
  Msg = "Its Class ID is: "
+ & GetClassID(MyLabel1)
  ShowFactoryStatus ( Msg )
  Msg = "The Class Name for MyLabel2 is: "
     & GetClassName(MyLabel2)
+
```

```
ShowFactoryStatus( Msg )
Msg = "Its Class ID is: " & GetClassID(MyLabel2)
ShowFactoryStatus( Msg )
End Sub
```

```
See also GetClassID function
IsKindOf function
```

## GetClipboardText function

Returns a text string from the operating environment Clipboard.

Syntax GetClipboardText(<format>)

### Parameters <format>

One of the Clipboard formats Actuate Basic recognizes as shown in Table 6-21.

Table 6-21	Clipboard formats
------------	-------------------

Symbolic constant	Value	Clipboard format
CF_LINK	&HBF00	DDE conversation information
CF_TEXT	1	Text

If <format> is omitted, CF\_TEXT is assumed.

### **Returns** String

If there is no text string on the Clipboard that matches the expected format, a zero-length string ("") is returned.

**Example** The following example places the current date on the Clipboard, then displays the contents of the Clipboard:

```
Sub Start( )
  Dim Msg As String
  Super::Start( )
  On Error Resume Next
  Msg = "The Clipboard contains: " & GetClipboardText
  ShowFactoryStatus( Msg )
  Msg = "Placing today's date on the clipboard."
  ShowFactoryStatus( Msg )
  ClearClipboard
```

```
SetClipboardText(Format$(Date, "dddd, mm/dd/yyyy"))
Msg = GetClipboardText
ShowFactoryStatus( "The Clipboard now contains: " & Msg )
End Function
```

See also ClearClipboard function SetClipboardText function

# **GetDisplayHeight function**

Returns the height required to display a given text string without truncating it, in twips.

Can be used to dynamically adjust the height of a container when you have a large amount of text that might otherwise overflow.

- Syntax GetDisplayHeight (<context>, <width>, <theText>, <theFont>, <format>)
- **Description** Different devices display or print fonts slightly differently. This means, for example, that a visual box filled with text may look correct on the screen, but incorrect in a printed report because the target device truncates some of the text.

GetDisplayHeight supports calculating exactly how tall the formatted text will be when a given display or printer device renders it.

### Parameters <context>

String expression. Specifies whether to return the height for display, or for the default printer. Valid values are display and default\_printer.

### <width>

Integer expression. Specifies the width of the control displaying the text, in twip. <width> is used to determine the height of the control to display all the text for this control. GetDisplayHeight assumes that at least one character fits on each line. Width must be a positive Integer.

### <theText>

String expression. The text whose height you want to determine. The string must not be empty.

### <theFont>

AcFont data type.

### <format>

AcTextPlacement data type.

### Returns Integer

• The height of the formatted text for the specified device, in twips.

 When negative, an error has occurred. Table 6-22 shows the meaning of the error codes.

Error code	Description
-2	Wrong number of parameters passed
-3	<context> is not a string, or represents a nonsupported context</context>
-4	<width> is not a positive integer</width>
-5	<thetext> is not a string, or it is an empty string</thetext>
-6	<acfont.facename> is an empty string</acfont.facename>
-7	<acfont.size> is not a positive integer</acfont.size>
-8	<acfont.bold> internal error</acfont.bold>
-9	<acfont.italic> internal error</acfont.italic>
-10	<acfont.underline> internal error</acfont.underline>
-11	<acfont.strikethrough> internal error</acfont.strikethrough>
-12	<actextplacement.multiline> internal error</actextplacement.multiline>
-13	<actextplacement.wordwrap> is not 0, 2, or 3</actextplacement.wordwrap>
-14	Default printer is selected, but no default printer has been set
-15	Implementation failed to get information on the device
-16	Operation caused an exception
-17	<font.script> is not a string or empty</font.script>

 Table 6-22
 Error codes for GetDisplayHeight

**Example** The following example determines display height based on given text parameters:

```
Sub Start( )
Dim myFont As AcFont
Dim myTextPlacement As AcTextPlacement
Dim theText as String
Dim Msg As String
Super::Start( )
' Specify font characteristics and text to measure
myFont.FaceName = "Arial"
myFont.Size = 14
myFont.Script = " "
theText = "Text that the display height is being queried for."
Msg = "The required height for the text: " & theText
ShowFactoryStatus( Msg )
```

```
Msg = "in a control of width 1 inch is "
   & GetDisplayHeight(
^{+}
           "display", OneInch, theText, myFont, myTextPlacement )
+
  ShowFactoryStatus ( Msq )
  Msg = "in a control of width 2 inches is "
     & GetDisplayHeight(
+
           "display", OneInch*2, theText, myFont,
+
          myTextPlacement )
+
  ShowFactoryStatus ( Msq )
  Msg = "in a control of width 3 inches is "
     & GetDisplayHeight(
+
           "display", OneInch*3, theText, myFont,
+
          myTextPlacement )
+
  ShowFactoryStatus ( Msg )
End Sub
```

# **GetFactoryVersion function**

Returns either the integer or the decimal component of the version number of the Factory that the report object executable (.rox) file is using.

Syntax GetFactoryVersion(<numpart>)

### Parameters <numpart>

Number or numeric expression that tells GetFactoryVersion whether you want to know the integer part of the Factory's version number, or the decimal part.

```
Rule: Must be either 0 or 1.
```

#### Returns Integer

- If <numpart> is 0, GetFactoryVersion returns the integer (major) component of the Factory's version number.
- If <numpart> is 1, GetFactoryVersion returns the decimal (minor) component of the Factory's version number.
- **Example** The following example determines both parts of the version number of the Factory that the ROX is using and puts the parts together, interpolating the decimal point where it belongs:

```
Dim FactVerMajor As Integer, FactVerMinor As Integer
Dim FactVerTotal As String
FactVerMajor = GetFactoryVersion(0)
FactVerMinor = GetFactoryVersion(1)
```

' Put the two parts together and prefix some identifying text: FactVerTotal = "Factory Version: " & (FactVerMajor) & "." & + (FactVerMinor)

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also GetAppContext function GetAFCROXVersion function GetOSUserName function GetROXVersion function GetServerName function GetServerUserName function

### GetFontAverageCharWidth function

Returns the average width (in twips) of a character in the specified font.

- **Syntax** GetFontAverageCharWidth(<font>)
- Parameters <font> AcFont data type.
  - **Returns** Integer specifying the average width of characters in twips of the font (generally defined as the width of the letter *x*). This value does not include the overhang required for bold or italic characters.
  - **Example** The following example overrides the Start method of a dynamic text control. It uses the average character width to compute the initial size of the control.

```
Sub Start( )
Dim width As Double
Dim fontHeight As Double
' Get the label size
width = GetFontAverageCharWidth( Font ) * 20
fontHeight = GetFontDisplayHeight( Font )
Size.width = width
Size.height = fontHeight
Super::Start( )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also GetFontDisplayHeight function GetTextWidth function

# GetFontDisplayHeight function

Size.width = width Size.height = fontHeight

Super::Start( )

earlier in this chapter.

GetDisplayHeight function

GetFontAverageCharWidth function

Returns the height (in twips) needed to display a character in the specified font.

```
Syntax GetFontDisplayHeight(<font>)
```

```
Parameters <font>
AcFont data type.
Returns Integer representing the height (in twips) needed to display a character in the
specified font. A font's height is the sum of its ascent, descent, and leading area.
Example The following example overrides the Start method of a dynamic text control. It
uses the font display height to compute the initial size of the control.
Sub Start()
Dim width As Double
Dim fontHeight As Double
' Get the label size
width = GetFontAverageCharWidth(Font) * 20
fontHeight = GetFontDisplayHeight(Font)
```

# **GetHeadline function**

End Sub

See also

Returns the headline associated with the completed request for a report.

For information about using the code examples, see "Using the code examples,"

Syntax	GetHeadline
Returns	String
	The current text of the headline.
Description	Headlines are associated with the completed request for a report.
	By default, the headline is set to the value of the Headline parameter that appears in the Output Options section of the Requester. Use GetHeadline to programmatically determine the headline text.
	To change the headline use SetHeadline, do not change the Headline parameter itself.

**Example** The following example overrides Start() on the report root. This code displays the current headline, sets a new headline, then displays the new headline.

```
Sub Start( )
Super::Start( )
' Getting the current headline.
ShowFactoryStatus( "Original headline is: " & GetHeadline )
' Setting and displaying the new headline
SetHeadline( "New Headline" )
ShowFactoryStatus( "New headline is: " & GetHeadline )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also SetHeadline statement

## **GetJavaException function**

Accesses the last Java exception thrown by the application.

- Syntax GetJavaException
- **Returns** A Basic object which is a handle to the Java Exception object if the method succeeded.

An undefined handle if the call failed.

**Example** The following example triggers a Java exception and then displays the error:

```
Sub Start()
  Dim javaObj As Object
  Dim Msg As String
  Super::Start( )
  On Error Goto HandleError
  ' Trigger a java exception
  Set javaObj = CreateJavaObject( "JavaClassName" )
  javaObj.TryToDoSomething()
HandleError:
  ' Retrieve error code
  Dim errorCode As Integer
  errorCode = Err
  ' If it is a java error, display error string
  if errorCode = E JAVAEXCEPTIONOCCURRED Then
     Dim exceptionObj As Object
     Set exceptionObj =GetJavaException
```

```
Msg = "Java exception: " & exceptionObj.toString()
    ShowFactoryStatus( Msg )
    End If
End Sub
```

## GetLocaleAttribute function

Retrieves locale attributes. You can use this information, for example, in formatting functions and functions that parse strings.

Syntax GetLocaleAttribute (<locale>, <attrib>)

#### Parameters <locale>

The name of the locale for which to retrieve attributes.

#### <attrib>

The enum value of the attribute to retrieve:

- AC\_LOCALE\_CURRENCY
- AC\_LOCALE\_CURRENCY\_FORMAT
- AC\_LOCALE\_CURRENCY\_RADIX
- AC\_LOCALE\_CURRENCY\_THOUSAND\_SEPARATOR
- AC\_LOCALE\_DATE\_LONG
- AC\_LOCALE\_DATE\_SEPARATOR
- AC\_LOCALE\_DATE\_SHORT
- AC\_LOCALE\_MONTHS\_LONG
- AC\_LOCALE\_MONTHS\_SHORT
- AC\_LOCALE\_NUM\_RADIX
- AC\_LOCALE\_NUM\_THOUSAND\_SEPARATOR
- AC\_LOCALE\_TIME\_AM\_STRING
- AC LOCALE TIME FORMAT
- AC\_LOCALE\_TIME\_PM\_STRING
- AC LOCALE TIME SEPARATOR
- AC\_LOCALE\_WEEKDAYS\_LONG
- AC\_LOCALE\_WEEKDAYS\_SHORT

#### **Returns** String

**Example** The following example returns the names of valid days of the week:

```
'Parse the numeric string "1.001,99" specified in French
locale
sep1000 = GetLocaleAttribute("fr",
+ AC_LOCALE_NUM_THOUSAND_SEPARATOR)
radix = GetLocaleAttribute("fr", AC_LOCALE_NUM_RADIX)
ParseNumeric("1.001,99", sep1000, radix)
```

```
' Get the names of valid days of week in short format
' for the current run-time locale
weekNames = GetLocaleAttributes(GetLocaleName(),
+ AC_LOCALE_ WEEKDAYS_SHORT)
' In the English locale, the following string is returned
' " Mon,Tue,Wed,Thu,Fri,Sat,Sun"
```

## GetLocaleName function

Retrieves the name of the current run-time locale.

Syntax GetLocaleName

**Description** This function retrieves the name of the current run-time locale.

Returns String

**Example** The following example displays the locale name:

```
Sub Start()
Super::Start()
' Display message containing locale name
ShowFactoryStatus( "The current locale name is: " &
+ GetLocaleName )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

# GetObjectIdString function

Returns a unique identifying string for an object within an ROI.

Syntax GetObjectIdString(<object>)

Parameters <object>

The handle or reference that refers to the object of which you want information. It can be an array reference, expression, or variable.

**Returns** String representing a unique identifier for the object.

### **GetOSUserName function**

Returns the operating system (login) name of the user. For server, returns the login name of the account running the server.

Syntax	GetOSUserName
Returns	String
	The login name of the user, or the login name of the account running the server.
Example	The following example retrieves the user's login name and puts it in a variable along with some identifying text:
	Dim OSUserName As String OSUserName = "OS User Name is: " & ( GetOSUserName )
	For information about using the code examples, see "Using the code examples," earlier in this chapter.
See also	GetAppContext function GetAFCROXVersion function GetFactoryVersion function GetROXVersion function GetServerName function

# GetPId function

Returns the process ID of the Factory.

GetServerUserName function

l
l

**Returns** Integer identifying the process ID of the Factory.

**Example** The following example uses the process ID to generate a unique temporary file name. Basic factories are not multithreaded, allowing for unique file names per process.

```
Sub Finish( )
Dim fileName As String
Dim pId As Integer
Static fileSequence As Integer
' Determine process id
pId = GetPId
' Generate file name. Use a static variable to sequence
' through filenames.
fileSequence = fileSequence + 1
```

```
FileName = "TEST" & pId & fileSequence & ".DAT"
ShowFactoryStatus(
+ "File name " & fileName & " has been generated." )
End Sub
```

## GetReportContext function

Returns the value of the global ReportContext variable, which specifies whether the report is in the factory, viewing, or printing stage of the life cycle.

Syntax GetReportContext

Returns ReportContext

One of the following values defined in Header.bas:

UnknownReportContext FactoryReportContext ViewerReportContext PrintReportContext

### **Description** Use GetReportContext in conjunction with GetAppContext.

Table 6-23 gives some examples of how the GetReportContext and GetAppContext functions work together to determine which Actuate application is being used and what the report context is at the time.

Table 6-23	Examples of how GetReportContent and GetAppContext determine
	the Actuate application in use and the report context

If GetAppContext returns this	And GetReportContext returns this	You are in the
ServerContext	ViewerReportContext	View process
ServerContext	FactoryReportContext	Factory server
ServerContext	PrintReportContext	Factory server

**Example** The following example determines whether the report is running in the View process:

InViewServer = ((GetReportContext = ViewerReportContext)
+ And (GetAppContext = ServerContext))

For information about using the code examples, see "Using the code examples," earlier in this chapter.

**See also** GetAppContext function

# GetReportScalingFactor function

Returns the scaling factor.

GetReportScalingFactor	
Use GetReportScalingFactor to retrieve the scaling factor and adjust the size of the browser scripting control's rectangle based on the current scaling factor.	
When the scaling factor changes, the DHTML converter scales controls on the report page, but does not scale or change the output of custom browser code. Make sure that the output of custom browser code can be viewed with different scaling factors, especially 75% and 100%.	
Double	
1.0 corresponds to a 100% zoom.	
The following example shows how to use GetReportScalingFactor to change the width and height of the HTML element produced by the custom browser code:	

```
Dim dhtmlCode As String
Dim zoom As Double
zoom = GetReportScalingFactor
dhtmlCode = "<SPACER TYPE=""block"" HEIGHT=" + Str$(150 * zoom)
dhtmlCode = dhtmlCode + " WIDTH=" + Str$(250 * zoom) + ">"
BrowserCode = dhtmlCode
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

# **GetROXVersion function**

Returns either the integer or the decimal component of the version number of the report object executable (.rox) file.

Syntax GetROXVersion(<numpart>)

### Parameters <numpart>

Number or numeric expression that tells GetROXVersion whether you want to know the integer part of the version number, or the decimal part.

Rule: Must be either 0 or 1.

### Returns Integer

- If <numpart> is 0, GetROXVersion returns the integer (major) component of the version number.
- If <numpart> is 1, GetROXVersion returns the decimal (minor) component of the version number.

**Example** The following example determines both parts of the version number of the ROX and puts the parts together, interpolating the decimal point where it belongs:

```
Dim ROXMajor As Integer, ROXMinor As Integer
Dim ROXTotal As Srting
ROXMajor = GetROXVersion(0)
ROXMinor = GetROXVersion(1)
' Put the two parts together and prefix some identifying text:
ROXTotal = "ROX Version: " & (ROXMajor) & "." & (ROXMinor)
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also GetAFCROXVersion function GetAppContext function GetFactoryVersion function GetOSUserName function GetServerName function GetServerUserName function

# **GetSearchFormats function**

Retrieves a list of supported DHTML search formats, places the supported formats into a string array, and returns the number of elements in the string array.

- Syntax GetSearchFormats(<formatArray>()) As Integer
- Parameters <formatArray>

String array to contain the list of supported search formats. The search formats determine how the DHTML viewing environment presents search results.

**Returns** The number of elements in the array.

The supported formats that GetSearchFormats returns depend upon the formats available in a particular viewing environment. For example, the following formats might be supported:

- XMLDisplay (DISPLAY)
- Download formats:
  - CSV
  - TSV
  - ANALYSIS

**Example** The following example retrieves a list of supported formats:

```
Function BrowserCode() As String
Dim fmtArray() As String
Dim count As Integer
Dim i As Integer
Dim DhtmlStr As String
DhtmlStr = "<SELECT NAME='search_format'>"
count = GetSearchPageFormats(fmtArray)
For i = 1 To count
DhtmlStr = DhtmlStr + "<OPTION VALUE=" + fmtArray(i) + ">"
DhtmlStr = DhtmlStr + "<OPTION VALUE=" + fmtArray(i) + ">"
DhtmlStr = DhtmlStr + fmtArray(i) + "</OPTION>"
Next i
DhtmlStr = DhtmlStr + "</SELECT>"
BrowserCode = dhtmlStr
End Function
```

See also IsSearchFormatSupported function

### GetServerName function

Returns the name of the machine on which iServer is running.

- Syntax GetServerName
- Returns String
- **Example** The following example displays in a control the name of the machine on which iServer is running:

Dim ServerName as String
ServerName = "Server Name: " & (GetServerName)

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also GetAFCROXVersion function GetAppContext function GetFactoryVersion function GetOSUserName function GetROXVersion function GetServerUserName function

# GetServerUserName function

Returns the login name of the user logged into iServer.

Syntax GetServerUserName

Returns String

**Example** The following example displays the login name of the user logged into iServer:

Sub Start()

Dim ServerUserName as String ServerUserName = "Server User Name: " & GetServerUserName ShowFactoryStatus ServerUserName End Sub

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also GetAFCROXVersion function GetAppContext function GetFactoryVersion function GetOSUserName function GetROXVersion function GetServerName function

# GetTextWidth function

Returns the width of a text string in twips.

Syntax	GetTextWidth( <text>, <font>)</font></text>	
Parameters	<text> The text string whose length is to be calculated.</text>	
	<font> AcFont data type.</font>	
Returns	The integer width of the text in twips.	
Example	<b>ple</b> The following example overrides the Start method of a dynamic text control. uses the text width to compute the initial size of the control.	
	Sub Start() Dim width As Double Dim fontHeight As Double Dim text As String	

```
' Get the label size
text = "Testing text"
width = GetTextWidth( text, Font )
fontHeight = GetFontDisplayHeight( Font )
Size.width = width
Size.height = fontHeight
Super::Start( )
End Sub
```

```
See also GetDisplayHeight function
GetFontAverageCharWidth function
```

## GetUserAgentString function

Returns the unparsed user agent string a web browser sends with every HTTP request.

Syntax GetUserAgentString

- Returns String
- **Example** The following example generates different output for Microsoft Internet Explorer and Mozilla Firefox:

```
Function GetText() As String
Dim str As String
str = GetUserAgentString
If Len(str) = 0 Then
GetText = "...Not in a web browser."
Else
If InStr(str, "MSIE") = 0 Then
'Not MSIE.
GetText = str + "...In Firefox"
Else
GetText = str + "...In MSIE"
End If
End If
End Function
```

# **GetValue function**

Finds the value of a variable in an object dynamically at run time.

Syntax GetValue(<object reference>, <variable name>l<index>)

#### Parameters <object reference>

AnyClass expression that specifies an object which has a variable whose value you want to find.

#### <variable name>

String expression that specifies the name of the variable whose value you want to find.

#### <index>

Integer expression that specifies the index of the variable whose value you want to find.

Indexing starts at 1. The index order puts all superclass variables before those defined in a subclass.

Within a given class, the index order of variables is the order in which the variables are defined in the Actuate Basic source. If e.Report Designer Professional generates the Basic source for a class, it lists the variables in alphabetical order.

- **Description** You typically use GetValue in conjunction with GetVariableCount, GetVariableName, GetValueType, and SetValue to work with variables in objects whose types you do not know in advance.
  - Returns Variant

The value of the specified variable if it exists. If the specified variable does not exist, GetValue returns Null. It is not an error if the variable does not exist.

**Tip** You cannot use the value GetValue returns to determine whether or not the specified variable exists by checking whether the returned value is Null. This is because GetValue will also return Null if the variable exists but has a value of Null. Instead, use GetValueType to check whether the variable exists.

If you know the class of the object, you should use the following technique to provide improved performance and compile-time error checking. You can access the variables in a class by declaring an object variable reference that points to an instance of that class, then access the variables through that object reference variable. For example, sometimes you are given an object reference variable that points to a base class, such as AcDataRow, but you are interested in a derived class, such as CustomerRow. You declare a temporary object reference variable of the derived class and set it equal to the reference to the base class as shown in the following example:

```
Sub OnRow(row as AcDataRow)
Dim custRow As CustomerRow
Set custRow = row
MyVariable = custRow.CustomerName
End Sub
```

**Example** The following example shows how to use GetVariableCount, GetVariableName, GetValueType, GetValue, and SetValue together:

```
' Simulate the behavior of the CopyInstance statement,
' but only copy integers > 0 whose names begin "Z "
Dim vCount As Integer
Dim vName As String
Dim vType As Integer
Dim vValue As Variant
Dim i As Integer
vCount = GetVariableCount( fromObject )
For i = 1 To vCount
  vName = GetVariableName( fromObject, i )
  vType = GetValueType( fromObject, vName )
  If (Left( vName, 2 ) = "Z ") And (vType = V INTEGER) Then
     vValue = GetValue( fromObject, vName )
     If (vValue > 0) Then
       SetValue( toObject, vName, vValue )
     End If
  End If
Next i
```

See also GetValueType function GetVariableCount function GetVariableName function SetValue function

## GetValueType function

Finds the data type of a variable in an object dynamically at run time.

**Syntax** GetValueType( <object reference>, <variable name>|<index>)

#### Parameters <object reference>

AnyClass expression that specifies an object which has a variable whose data type you want to find.

#### <variable name>

String expression that specifies the name of the variable whose data type you want to find.

#### <index>

Integer expression that specifies the index of the variable whose data type you want to find.

Indexing starts at 1. The index order puts all superclass variables before those defined in a subclass. Within a given class, the index order of variables is the order in which the variables are defined in the Actuate Basic source. If e.Report

Designer Professional generates the Basic source for a class, it lists the variables in alphabetical order.

- **Description** You typically use GetValueType in conjunction with GetVariableCount, GetVariableName, GetValue, and SetValue to work with variables in objects whose types you do not know in advance.
  - **Returns** Integer indicating the data type of the specified variable, if it exists. If the specified variable does not exist, GetValueType returns Null. It is not an error if the variable does not exist.

Table 6-24 shows the values VarType returns, the Variant data types they indicate, and the names of the corresponding symbolic constants stored in Header.bas.

GetValueType returns	Data type of variable	Symbolic constant
0	Empty	V_EMPTY
1	Null	V_NULL
2	Integer	V_INTEGER
3	Long	V_LONG
4	Single	V_SINGLE
5	Double	V_DOUBLE
6	Currency	V_CURRENCY
7	Date	V_DATE
8	String	V_STRING

Table 6-24VarType values with their Variant data types and symbolic<br/>constants

**Tip** Test the value GetValueType returns with the IsNull function to determine whether or not the specified variable exists. For example:

```
Dim valueType As Integer
valueType = GetValueType( anObject, "SomeVariable" )
Assert( Not IsNull( valueType ), "SomeVariable not found!" )
```

**Example** The following example shows how to use GetVariableCount, GetVariableName, GetValueType, GetValue, and SetValue together:

```
' Simulate the behavior of the CopyInstance statement,
' but only copy integers > 0 whose names begin "Z_"
Dim vCount As Integer
Dim vName As String
Dim vType As Integer
Dim vValue As Variant
Dim i As Integer
```

```
vCount = GetVariableCount( fromObject )
For i = 1 To vCount
  vName = GetVariableName( fromObject, i )
  vType = GetValueType( fromObject, vName )
  If (Left( vName, 2 ) = "Z_") And (vType = V_INTEGER) Then
      vValue = GetValue( fromObject, vName )
      If (vValue > 0) Then
         SetValue( toObject, vName, vValue )
      End If
   End If
   Next i
```

See also GetValue function GetVariableCount function GetVariableName function SetValue function

# **GetVariableCount function**

Returns the number of variables in an instance. Used in conjunction with GetVariableName.

Syntax GetVariableCount(<object reference>)

### Parameters <object reference>

Any class expression. An object variable that refers to the object that has the variables you want to count.

Indexing of variables starts at 1. The index order puts all superclass variables before those defined in a subclass.

Within a given class, the index order of variables is the order in which the variables are defined in the Actuate Basic source. If Actuate generates the Basic source, as is standard for a data row, then Actuate lists the variables in alphabetical order.

#### **Returns** Integer

- **Tip** Use GetVariableName and GetVariableCount to loop over all the variables in an instance, and get both the variables' names and values. You can write highly customized reports, or general purpose libraries using these functions.
- **Example 1** The following example shows how to use GetVariableCount, GetVariableName, GetValueType, GetValue, and SetValue together:
  - ' Simulate the behavior of the CopyInstance statement,
  - ' but only copy integers > 0 whose names begin "Z\_"
```
Dim vCount As Integer
Dim vName As String
Dim vType As Integer
Dim vValue As Variant
Dim i As Integer
vCount = GetVariableCount( fromObject )
For i = 1 To vCount
  vName = GetVariableName( fromObject, i )
  vType = GetValueType( fromObject, vName )
  If (Left( vName, 2 ) = "Z ") And (vType = V INTEGER) Then
     vValue = GetValue( fromObject, vName )
     If (vValue > 0) Then
        SetValue( toObject, vName, vValue )
     End If
  End If
Next i
```

**Example 2** The following example retrieves the count of members in a local class. In it, the New() method of a data row component has been overridden. The method instantiates the parent class and then subtracts the member count of the parent class from the member count of the current class.

```
Sub New()
  Super::New( )
  ' Count local members of a class by instantiating
  ' a parent class and then subtracting the count of
  ' those members from the count of the local members.
  Static localCount As Variant
  If IsEmpty( localCount ) Then
     localCount = 0
     Dim r As AcDataRow
     Set r = New AcDataRow
     Dim parentCount As Integer
     parentCount = GetVariableCount( r )
     Set r = New MyDataRow
     localCount = GetVariableCount( r ) - parentCount
     Set r = Nothing
  End If
End Sub
```

**Example 3** The following example retrieves the index of the first local variable in a data row class:

```
Sub New( )
Super::New( )
```

```
' Compute the index of the first local variable in a
' data row subclassed from AcDataRow.
Static firstRowVariableIndex As Integer
if (firstRowVariableIndex = 0) Then
   Dim r As AcDataRow
   Set r = New AcDataRow
   firstRowVariableIndex = GetVariableCount(r) + 1
   Set r = Nothing
   End If
End Sub
```

See also GetValue function GetValueType function GetVariableName function SetValue function

# **GetVariableName function**

Returns the name of the variable at a given index.

**Syntax** GetVariableName(<object reference>, <index>)

### Parameters <object reference>

Any class expression. An object variable that refers to the object that has the variable whose name you want to find.

### <index>

Integer that specifies the index of a variable.

Indexing starts at 1. The index order puts all superclass variables before those defined in a subclass.

Within a given class, the index order of variables is the order in which the variables are defined in the Actuate Basic source. If Actuate generates the Basic source, as is standard for a data row, then Actuate lists the variables in alphabetical order.

### Returns String

- **Tip** Use GetVariableName and GetVariableCount to loop over all the variables in an instance, and get both the variables' names and values. You can write highly customized reports or general purpose libraries using these functions.
- **Example** The following example shows how to use GetVariableCount, GetVariableName, GetValueType, GetValue, and SetValue together:

```
' Simulate the behavior of the CopyInstance statement,
' but only copy integers > 0 whose names begin "Z "
Dim vCount As Integer
Dim vName As String
Dim vType As Integer
Dim vValue As Variant
Dim i As Integer
vCount = GetVariableCount( fromObject )
For i = 1 To vCount
  vName = GetVariableName( fromObject, i )
  vType = GetValueType( fromObject, vName )
  If (Left( vName, 2 ) = "Z ") And (vType = V INTEGER) Then
     vValue = GetValue( fromObject, vName )
     If (vValue > 0) Then
       SetValue( toObject, vName, vValue )
     End If
  End If
Next i
```

See also GetValue function GetValueType function GetVariableCount function SetValue function

# GetViewPageFormats function

Retrieves a list of supported DHTML viewing formats, places the supported viewing formats into a string array, and returns the number of elements in the string array.

Syntax GetViewPageFormats(<formatArray>)

### Parameters <formatArray>

String array to contain the list of supported DHTML viewing formats. The viewing formats determine how the DHTML viewing environment presents a report.

**Returns** The number of elements in the array.

The supported viewing formats that GetViewPageFormats returns depend upon the viewing formats available in a particular viewing environment. For example, your DHTML viewing environment might support the following viewing formats:

- XMLDisplay
- XMLCompressedDisplay
- DHTML
- DHTMLLong
- DHTMLRaw
- PDF

The viewing formats are display formats. Formats such as CSS and XMLStyle might be supported, but are not display formats. The GetViewPageFormats function returns only display formats.

You can use GetViewPageFormats to dynamically build a view page URL such as ?ViewPage&format=PDF.

**Example** The following example retrieves a list of supported viewing formats:

```
Function BrowserCode() As String
  Dim formatArray() As String
  Dim count As Integer
  Dim i As Integer
  Dim DhtmlStr As String
  DhtmlStr = "<SELECT NAME='view format'>"
  count = GetViewPageFormats(formatArray)
  For i = 1 To count
     DhtmlStr = DhtmlStr + "<OPTION VALUE=" + formatArray(i) +
        ">"
+
     DhtmlStr = DhtmlStr + formatArray(i) + "</OPTION>"
  Next i
  DhtmlStr = DhtmlStr + "</SELECT>"
  BrowserCode = dhtmlStr
End Function
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also IsViewPageFormatSupported function

# **GetVolumeName function**

Retrieves the name of the Encyclopedia volume on which the file is stored.

Syntax GetVolumeName

- **Description** This function retrieves the name of the Encyclopedia volume on which the file is stored.
  - **Returns** If the file is stored in the Encyclopedia, the name of the Encyclopedia volume.

If the file is not stored in the Encyclopedia, an empty string.

**Example** The following example shows how to use GetVolumeName. To use this example, create an empty report named ShowVolumeName. Override the report's Start() method and paste the following code into the method editor:

```
Sub Start()
Super::Start()
ShowFactoryStatus("Volume is: " & GetVolumeName)
End Sub
```

Run this report from e.Report Designer Professional to produce the following output in the Output window:

```
Running...
Starting...
Generating ShowVolumeName.roi ...
Volume is:
Finishing...
Idle.
```

Publish the report to iServer. In this example, the Encyclopedia volume is named seamore. Run the report from Actuate Management Console or Actuate Information Console to produce the following output in Job Status messages:

```
Starting...
Generating rotp:/ShowVolumeName.roi;0 ...
Volume is: seamore
Finishing...
Headline:
Job completed.
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

**See also** ShowFactoryStatus statement

# Global statement

Declares and allocates memory for variables, and declares the variables to be available to all procedures in all modules.

Syntax Global <variable name> [([<subscripts>])] [As <type>] [, <variable name> [([<subscripts>])] [As <type>]]...

### Parameters <variable name>

A name you create for the new variable.

### <subscripts>

Each <subscripts> argument specifies the number of elements in a dimension. The number of <subscripts> arguments specifies the number of dimensions in the array.

Array dimensions using the following syntax:

[<lower>] To <upper>],[<lower>] To <upper>]...

Rules for <subscripts>:

- If you do not specify <subscripts> between parentheses, Actuate Basic declares a dynamic array.
- <lower> and <upper> can range from -32,768 to 32,767 inclusive.
- <lower> must always be less than <upper>.
- Use no more than 60 dimensions in a Global statement.
- Do not later use ReDim to declare more than 8 dimensions for a dynamic array you declared with Global.
- **Examples** The statements within the following Declare...End Declare block are equivalent if you do not use Option Base:

```
Declare
Global P(7,5)
Global P(0 to 7, 0 to 5)
Global P(7, 0 to 5)
End Declare
```

The following example declares a dynamic array:

Global TestArray()

# As <type>

Specifies a data type or class for the variable. If you specify a class, the variable can hold a reference to an instance of that class or descendant classes.

Rule for As <type>: If you do not specify As <type>, Actuate Basic uses the data type you declared with Dim or defaults to a Variant.

**Rule** Use Global within a Declare...End Declare block in your .bas source file.

- **Tips** To avoid assigning incorrect variable types, use the As clause to declare variables.
  - To create a dynamic array with more than eight dimensions, declare it with ReDim, not Global or Dim.

- Use global variables sparingly; declare variables in local scope if at all
  possible. Global variables are most useful when values must be shared or
  accessed by several tasks. Adding a lot of global variables makes debugging
  more complicated; in addition, variables with global scope must be kept in
  memory while the application is running.
- To avoid debugging problems, note that variables declared with Global are automatically initialized as shown in Table 6-25.

Туре	Initialized as
Numeric	0
Variant	Empty
Variable-length strings	Zero-length String
CPointer	Null
User-defined	Separate variables
Object or Class	Nothing (before Set)

 Table 6-25
 Initialization of variables that are declared with Global

To initialize one or more of your global variables to some default value (other than zero), write a procedure that initializes the variables, then call that procedure from the OnStartViewer() method of the report.

**Example** The following Declare statement uses Global to declare two variables, TestArray and PrintFlag. TestArray is a global dynamic array variable. PrintFlag is a global integer variable. Several parts of the application check PrintFlag to determine whether a print action has been completed.

To use the following example, paste the Declare block and Sub block at or near the beginning of your Actuate Basic code module (.bas) file:

```
Declare
   ' Global dynamic array
   Global TestArray()
   ' Global integer variable
   Global PrintFlag as Integer
End Declare
Sub LoadGlobalArray()
   Dim I As Integer, Size As Integer, Msg As String
   ' First check if the global array has already been printed.
   If PrintFlag Then
      Msg = "That information has already been printed."
      ShowFactoryStatus( Msg )
```

```
Else
  ' Otherwise, set up TestArray for printing
  Size = Int(10 * Rnd + 1)
  ReDim TestArray(Size)
  For I = 1 to Size
    TestArray(I) = Int(10 * Rnd + 1)
    Msg = TestArray(I)
    ShowFactoryStatus( Msg )
   Next I
    ' Set print flag to True to notify the whole application
    PrintFlag = True
  End If
End Sub
```

Paste the following line into the Start() method of a label or other component in the method editor:

```
LoadGlobalArray
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Const statement Dim statement GetAFCROXVersion function GetAppContext function GetFactoryVersion function GetOSUserName function GetROXVersion function GetServerUserName function Option Base statement ReDim statement Static statement Type...End Type statement

# GoTo statement

Branches to a specified line within a procedure unconditionally and without returning to the calling statement.

Syntax GoTo { <line label> | <line number> }

# Parameters <line label>

The name of any existing program label that marks the statement to execute next.

Rules for setting up <line label>:

Must begin with an alphabetic character and end with a colon.

- Must not be an Actuate Basic keyword.
- Must be unique to the module in which it appears.
- Must be the first non-blank characters on the line.
- Is not case-sensitive.

```
Sub Start( )
Super::Start( )
' Using Goto with a line label
On Error Goto MyErrorHandler
...
Exit Sub
' The line label
MyErrorHandler:
...
End Sub
```

#### <line number>

Example

The number of any program line that marks the statement to execute next.

Rules for setting up <line number>:

- Must consist entirely of decimal digits (0 through 9).
- Must not end with a colon.
- Must be unique to the module in which it appears.
- Must not be greater than 2,147,483,647 if you intend to use the Erl function.
- Can begin in any column, but must be the first non-blank character on the line.

```
Example Sub Start()
    500 Dim X, Y, Z
    505 Super::Start()
    ' Set up error handler
    510 On Error GoTo 550
    520 Y = 1
    'Now cause division by zero error:
    530 X = Y / Z
    540 Exit Sub
    ' Error handler
    550 ShowFactoryStatus( "Error occurred at line " & Erl )
    560 Resume Next
End Sub
```

- **Rule** GoTo can branch only to lines that exist within the procedure in which it appears.
- **Tips** Using GoTo in your code makes it difficult to understand and debug. To avoid confusion, use GoTo rarely. Instead, use more structured control statements,

such as Do...Loop, For...Next, If...Then...Else, and Select Case, or define Function and Sub procedures.

Evaluate Boolean variables by using the keywords True or False.

**Example** The following example uses GoTo to branch around a subroutine:

```
Sub Start()
  Dim UserNum as Integer, HalfNum As Double
  Super::Start( )
  HalfNum = 0
  UserNum = Int(255 * Rnd + 1)
  ' Branch around subroutine
  GoTo ThisPlusColon
  ShowFactoryStatus ( "This message never appears." )
Routine:
  ' GoTo detours around this subroutine
  HalfNum = UserNum / 2
ThisPlusColon:
  ShowFactoryStatus ( "UserNum is: " & UserNum
        & ". HalfNum is still: " & HalfNum )
+
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Erl function

Do...Loop statement For...Next statement Function...End Function statement If...Then...Else statement Select Case statement Sub...End Sub statement

# Hex, Hex\$ functions

Converts a numeric expression from decimal to hexadecimal notation, and from numeric to string.

Syntax Hex(<numeric expression>)

Hex\$(<numeric expression>)

# Parameters <numeric expression>

Numeric expression to be converted from decimal to hexadecimal notation.

**Examples** The following statements are equivalent. Each returns &H010, which is the full form of the hexadecimal equivalent of decimal number 16.

```
"&H" & Hex$(16)
"&H" & Hex$(2*8)
```

In the following example, the first statement returns decimal value 32, and the second statement returns decimal value 37:

```
("&H" & Hex$(16)) * 2
Hex$(16) * 2 + 17
```

# Returns Hex: Variant

Hex\$: String

- Hex[\$] rounds <numeric expression> to the nearest whole number before evaluating it.
- If <numeric expression> evaluates to Null, Hex[\$] returns Null.
- If the data type of <numeric expression> is Integer, Variant of VarType 2 (Integer), or Variant of VarType 0 (Empty), Hex[\$] returns up to four hexadecimal characters.
- If <numeric expression> is of any other numeric or Variant data type, Hex[\$] returns up to eight hexadecimal characters.
- **Tips** To represent a hexadecimal number directly, precede a number in the correct range with the radical prefix &H. The valid range for hex numbers is &H0 to &HFFFFFFFF. For example, use &H010 \* 2 to generate the decimal value 32, &H010 is hexadecimal notation for decimal value 16.
  - To generate the full hexadecimal representation of <numeric expression>, supply the radical prefix &H, because Hex[\$] does not return that component.
  - To ensure correct results when you assign the output of Hex to a variable, be sure the variable is of type Variant.
- **Example** The following example generates a decimal number, then uses Hex[\$] to convert that number to hexadecimal notation:

```
Sub Start( )
Dim Msg, Num
Super::Start( )
' Generate a number
Num = 255 * Rnd + 1
Msg = Num & " in decimal notation is &H"
+ & Hex$( Num ) & " in hexadecimal notation."
ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Oct, Oct\$ functions Val function

# **Hour function**

Returns the hour of the day as an integer from 0 (midnight) to 23 (11:00 P.M.), inclusive, based on a specified date expression.

Syntax Hour(<date expression>)

### Parameters <date expression>

Date expression, or any numeric or string expression that can be interpreted as a date, a time, or both a date and a time:

- Can be a string expression such as November 12, 1982 8:30 P.M., Nov. 12, 1982 08:30 PM, 11/12/82 8:30PM, 08:30pm, or any other string that can be interpreted as a date or both a date and a time in the valid range.
- Can be a date serial number such as 30267.854, which corresponds to November 12, 1982 at 8:30 P.M., or any other number that can be mapped to a date or both a date and a time in the valid range.
- For date serial numbers, the integer component to the left of the decimal represents the date itself while the fractional component to the right of the decimal represents the time of day on that date, where January 1, 1900, at precisely noon has the date serial number 2.5, and negative numbers represent dates prior to December 30, 1899 (0).

Default if no time specified: 0

Rules:

- If <date expression> includes a date, it must be a valid date, even though Hour does not return a date. A valid date is any date in the range January 1, 100 through December 31, 9999, inclusive, expressed in one of the standard date formats.
- If <date expression> includes a time, it must be in the range 00:00:00 (12:00:00 A.M.) through 23:59:59 (11:59:59 P.M.), in either 12- or 24-hour notation.
- If <date expression> is a numeric expression, it must be in the range -657,434 to +2,958,465, inclusive.
- If <date expression> is a variable containing a date serial number, the variable be explicitly declared as one of the numeric types.
- <date expression> is parsed according to the formatting rules of the current run-time locale.
- **Examples** The following statements are equivalent. Each assigns 14 to the variable UserHour.

UserHour = Hour("6/7/64 2:35pm") UserHour = Hour("14:35")

```
UserHour = Hour("June 7, 1964 2:35 PM")
UserHour = Hour("Jun 7, 1964") + 14
UserHour = Hour(23535.6076)
UserHour = Hour(0.6076)
```

# Returns Integer

 If <date expression> cannot be evaluated to a date, Hour returns Null. Example:

Hour("This is not a date.") returns Null

 If <date expression> fails to include all date components, such as day, month, and year, Hour returns Null.

Examples:

Hour("Nov 12, 1982 7:11 AM") returns 7, but Hour("Nov 1982 7:11 AM") returns Null

If <date expression> is Null, Hour returns Null.

The following assumes that the AC\_CENTURY\_BREAK property is set to the default value of 30. For information about using AC\_CENTURY\_BREAK, see *Accessing Data using e.Report Designer Professional.* 

- For abridged date expressions ending in 00 through 29 that do not specify the century, relevant date functions estimate the unabridged year by adding 2000. For example, while Year(#4/30/1910#) returns 1910, the abridged expression Year(#4/30/10#) returns 2010 (10 + 2000).
- For abridged date expressions ending in 30 through 99 that do not specify the century, relevant date functions estimate the unabridged year by adding 1900. For example, while Year(#11/12/2082#) returns 2082, the abridged expression Year(#11/12/82#) returns 1982 (82 + 1900).

Note that a date can contain an abridged month, such as Nov. 12, 1982 as opposed to November 12, 1982.

- **Tip** If you use a date expression in one locale, it might be misinterpreted in another locale. For instance, in the United States, 1/2/2005 means January 2nd, 2005, but in France, it means February 1st, 2005. To avoid such ambiguities, use DateSerial to specify all your dates.
- **Example** The following example displays the number of hours, minutes, and seconds remaining until midnight:

```
Sub Start( )
Dim HrDiff As Integer, MinDiff As Integer, SecDiff As Integer
Dim RightNow As Double, Midnight As Double
Dim TotalDiff As Double, TotalMinDiff As Double
Dim TotalSecDiff As Double, Msg As String
```

```
Super::Start( )
  Midnight = TimeValue( "23:59:59" )
  ' Get current time
  RightNow = Now
  ' Get diffs from midnight
  HrDiff = Hour( Midnight ) - Hour( RightNow )
  MinDiff = Minute( Midnight ) - Minute( RightNow )
  SecDiff = Second( Midnight ) - Second( RightNow ) + 1
  ' Restate seconds and minutes if necessary
  If SecDiff = 60 Then
     ' Add 1 to minute
     MinDiff = MinDiff + 1
     ' And set 0 seconds
     SecDiff = 0
  End If
  If MinDiff = 60 Then
     ' Add 1 to hour
     HrDiff = HrDiff + 1
     ' And set 0 minutes
     MinDiff = 0
  End If
  ' Now get totals
  TotalMinDiff = ( HrDiff * 60 ) + MinDiff
  TotalSecDiff = ( TotalMinDiff * 60 ) + SecDiff
  TotalDiff = TimeSerial( HrDiff, MinDiff, SecDiff )
  ' Prepare msg for display
  Msg = "There are a total of "
     & Format( TotalSecDiff, "#,##0" )
+
   & " seconds until midnight. That translates to "
+
    & HrDiff & " hours, "
+
   & MinDiff & " minutes, and "
+
   & SecDiff & " seconds. "
+
     & "In standard time notation, it becomes "
  ' Remember not to use "mm" for minutes! m is for month.
  Msg = Msg & Format(TotalDiff, "hh:nn:ss") & "."
  ShowFactoryStatus ( Msg )
End Sub
```

See also Day function Minute function Month function Now function Second function Time, Time\$ functions Weekday function Year function

# If...Then...Else statement

Executes an instruction or block of instructions based on a condition or series of conditions.

# Syntax Single-line If...Then...Else statement

If <condition> Then <then-statement> Else <else-statement>

# Block, multiple-line If...Then...Else statement

If <condition> Then

<then-statement>

[[Elself <condition> Then

<then-statement>]]

[Else

[<else-statement>]]]

End If

# Parameters <condition>

A numeric or string expression that can be evaluated and that determines whether to execute the associated <statements>. When <condition> evaluates to zero or Null, the condition is False. Otherwise, <condition> is True.

# <then-statement>, <else-statement>

One or more Actuate Basic expressions which can include nested If...Then...Else statements. The <then-statement> executes if the preceding <condition> is true. The <else-statement> executes if all previous <conditions> are False. Only one block of <statements> is executed in an If...Then...Else statement: the block of statements following the first <condition> found to be true, or, if no <condition> is found to be true, the <else-statement> block.

# lf

Keyword that begins every If...Then...Else statement and precedes the first <condition> to be tested.

# Then

Keyword that separates <condition> from <then-statement> in an If...Then...Else statement. When <condition> evaluates to True, the <then-statement> is

executed. When <condition> evaluates to False, the <then-statement> is ignored, and the program execution tests the next <condition> that it finds.

# Else

A keyword that tells the program what to do when all previous conditions are False.

# Elself

Keyword that begins an alternative <condition> to be evaluated when the previous <condition> is false. You can use multiple ElseIf clauses. (Tip: if you have several ElseIf clauses, consider using a Select Case statement instead.) Actuate Basic evaluates each ElseIf <condition> in turn, but only until the first true one is encountered. When an ElseIf <condition> evaluates to true, its <then-statements> are executed and control resumes after End If.

# End If

Keyword that signals the end of an If...Then...Else statement. A multi-line block If...Then...Else statement must be closed with End If. It is not required for a single-line If...Then...Else statement.

Rule: If, Else, ElseIf, and End If must be the first statement on a line. Else, ElseIf, and End If can be preceded by a line number or a line label.

- **Tip** It is good programming practice to evaluate Boolean variables by using the keywords True or False instead of by inspecting their content for a nonzero (True) or zero (False) numeric value.
- **Example** The following example generates a number between 1 and 10000. Then it returns the number of digits in the number.

```
Sub Start()
  Dim UserNum As Integer, NumDigits As Integer
  Dim Msg As String, Unit As String
  Super::Start( )
  UserNum = Int(10000 * Rnd + 1)
  If UserNum < 10 Then
  ' 1 digit
     NumDigits = 1
  ElseIf UserNum < 100 Then
     ' 2 digits
     NumDigits = 2
  ElseIf UserNum < 1000 Then
     ' 3 digits
     NumDigits = 3
  ElseIf UserNum < 10000 Then
     ' 4 digits
     NumDigits = 4
  Else
```

' Prepare msg if none of our conditions was met Msg = "Sorry, the number " & userNum & " is too high." End If ' Prepare a msg for display if at least one condition was met If UserNum < 10000 Then If NumDigits > 1 Then Unit = " digits." Else Unit = " digit." End If Msg = "The number " & userNum & " has " & NumDigits & Unit End If ' Display message ShowFactoryStatus( Msg ) End Sub

For information about using the code examples, see "Using the code examples," earlier in this chapter.

# See also End statement Exit statement Select Case statement

# **Ilf function**

Evaluates an expression and returns one of two supplied values; if the expression that is the first argument is true, IIf returns the second argument, if false, IIf returns the third.

All three arguments must be valid. That is, none can generate an error since all three are always evaluated whether <condition> is True or False.

Syntax IIf(<condition>, <value if true>, <value if false>)

# Parameters <condition>

Numeric or string expression to be evaluated.

# <value if true>

Numeric or string expression that specifies the value or expression to return if <condition> is true.

# <value if false>

Numeric or string expression that specifies the value or expression to return if <condition> is false.

**Example** The following statement assigns the double-precision value 112.335577 to the variable, ValidMsg, if the variable contains a value less than 5. It assigns the string

expression: Sorry, invalid response, if the value of UserAns is greater than or equal to 5.

ValidMsg = IIf(UserAns < 5, 112.335577, + "Sorry, invalid response")

- **Returns** Same type as that of <value if true> or <value if false>, depending on which is returned.
- **Example** The following example finds the day of the week and displays an appropriate message:

```
Sub Start( )
   Dim TodaysDay As Integer, ValidMsg As String
   Super::Start( )
   TodaysDay = Weekday(Now)
   ValidMsg =
+ IIf(TodaysDay = 1 Or TodaysDay = 7, "Weekend!", "Work day")
   ShowFactoryStatus( ValidMsg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

```
See also If...Then...Else statement
Select Case statement
```

# Input statement

Reads data from a sequential file and assigns the data to variables.

Syntax Input #<open file number>, <variable list>

**Description** The Input statement places each data item in the file into one or more of the variables in <variable list>. Data items in the file must appear in the same order as the variables in <variable list> and each variable in <variable list> must be of the same data type as the target data item.

The Input statement reads each data item and assigns it to a corresponding variable according to its data type. Table 6-26 summarizes the behavior of Input.

Fable 6-26	Behavior of Input
------------	-------------------

Variable in <variable list=""></variable>	Behavior of Input
Numeric	Assumes first nonspace character it encounters to be the start of a number. Assumes the first space, comma, or end- of-line it encounters to be the end of a number. Inputs blank lines and data of an invalid data type as zero.

Variable in <variable list=""></variable>	Behavior of Input				
String	Assumes the first nonspace character it encounters to be the start of a character string. If that character is a double quote ("), it ignores the mark itself and inputs all the succeeding characters until it encounters the next double quote, including spaces and commas. If string data are not delimited by double quotes, it assumes the first line, space, or end-of-line character to be the end of the string. Inputs blank lines as zero-length strings.				

Table 6-27 summarizes how Input determines what Variant type to assign to Variants in <variable list>.

Input data	Variant type assigned	
No data, but only a delimiting comma, or a blank line.	VarType 0 (Empty)	
The literal Null.	VarType 1 (Null)	
Valid number.	An appropriate numeric Variant (types 2 through 6)	
Date literal. A	VarType 7 (Date)	
date of the form yyyy-mm-dd hh:nn:ss).	Tip: When either the date or time portion is present, the other portion is optional.	
None of the above.	VarType 8 (String)	

 Table 6-27
 Types of input data and the corresponding Variant types

The following rules apply to Input:

- Input generates an error if it reaches the end of file while it is inputting a data item.
- Data items in a file should appear in the same order as the variables in <variable list> to which they correspond.

### Parameters <open file number>

Table 6-26

Rehavior of Input

Numeric expression that is the file number assigned by Open statement when the target file was opened.

Rules for <open file number>:

• The number sign (#) preceding the file number is required.

The file must be open.

### <variable list>

Comma-delimited list of the variables that act as containers for values read from the file.

Rules for <variable list>:

- Cannot contain array variables, but can contain variables that reference an element of an array. For instance, MyArray() refers to an entire array and is therefore not allowed, but MyArray(23) refers to a specific element in the array and is therefore a valid variable.
- Cannot be an object variable, user-defined data type structure, handle to a class, CPointer, or OLE object.
- **Example** The following code fragment opens an existing file named Test.fil, assigns data from it to a variable called FileData, and constructs a message:

```
Open "Test.fil" For Input As #1
Do While Not EOF(1)
Input #1, FileData
ShowFactoryStatus(FileData)
Loop
Close #1
Kill "Test.fil"
```

- **Tips** To prevent Input from encountering the end-of-file before it is finished processing data, use it within a loop that tests for the end-of-file condition. For more information, see EOF function.
  - Match data in the file with variables of the proper data type, that is:
    - Assign string data to string or Variant variables.
    - Assign numeric data to numeric or Variant variables.
    - Assign date data to date or Variant variables.
    - Assign uniquely Variant data—like Empty and Null values—only to Variant variables.
- **Example** The following example creates a test file and reads nonnumeric values from the test file into a variable. The example overrides Start() to generate the test file. To use this example, paste the procedure MakeSomeData after the End Function of the Start() procedure or save it in an Actuate code module (.bas) file.

```
Sub Start( )
Dim FileData As String
Super::Start( )
' Create test file
MakeSomeData
```

```
' Open to read test file
  Open "Test.fil" For Input As #1
  Do While Not EOF(1)
     ' Read a line of data
     Input #1, FileData
     ' Construct message
     ShowFactoryStatus (FileData)
  Loop
  ' Close file
  Close #1
  ' Delete test file
  Kill "Test.fil"
End Sub
Sub MakeSomeData()
  ' Create the test file used in Start
  ' Open to write test file
  Open "Test.fil" For Output As #1
  ' Write sample data
  Write #1, "This is the first line of the file."
  Write #1, "This is the second line."
  ' Close test file
  Close #1
End Sub
```

# See also Input, Input\$ functions Write statement

# Input, Input\$ functions

Returns a specified number of characters from a sequential file. The Input function can be used only with files opened in Input or Binary mode.

Syntax Input(<number of chars>,<file number>)

Input\$(<number of chars>, <file number>)

# Parameters <number of chars>

Numeric expression for the number of characters to read from the file. <number of chars> must be between 1 and 2,147,483,647.

For example, the following code fragment opens a file in the current directory named Myfile.txt, reads 51 characters from that file into a variable called ReadChunk, then closes the file:

```
Open "Myfile.txt" For Input As #1
          ReadChunk = Input$(51, #1)
          Close #1
          <file number>
          Integer that is the file number assigned by Open statement when the target file
          was opened. File <file number> must be open.
         Input: Variant
Returns
          Input<sup>$</sup>: String
          Returns all the characters it reads, including commas, carriage returns, linefeeds,
          quotation marks, and leading spaces.
Example
          The following example reads one character at a time from a file. When it
          encounters an ANSI character code 10 (linefeed), it displays the entire line of text
          in a message box for up to five lines.
          Sub Start()
            Dim Collector, UsersFile, DisplayMe
            Dim LineCount As Integer
             Super::Start( )
             ' Set file name
            UsersFile = "C:\Program Files\Actuate11\eRDPro\AcUninst.txt"
             If Len( UsersFile ) Then
                ' Open specified file
               Open UsersFile For Input As #1
               LineCount = 0
                ' read in to the end of file, or max of 5 lines
                ' whichever comes first.
               Do While Not EOF(1) And LineCount < 5
                  ' Obtain 1 character
                  Collector = Input$(1,1)
                  ' If not a linefeed, then collect it
                  If Collector <> Chr(10) Then
                     DisplayMe = DisplayMe & Collector
                   ' If it is a linefeed, then display it
                  Else
                     ShowFactoryStatus( DisplayMe )
                     ' Clear the line
                     DisplayMe = ""
                     LineCount = LineCount + 1
                  End If
                ' Loop if not EOF
               Loop
                ' Close the file
               Close #1
            End If
          End Sub
```

See also Input statement InputB, InputB\$ functions

# InputB, InputB\$ functions

Returns a specified number of bytes from a sequential file. This function can be used only with files opened in Input or Binary mode.

These functions are provided for backward compatibility with previous Actuate releases. Actuate uses UCS-2 as internal encoding, therefore passing strings consisting of characters from different code pages might produce unexpected results. For this reason, Actuate recommends you use the Input and Input\$ functions instead of the InputB and InputB\$ functions.

Syntax InputB(<number of bytes>,<file number>)

InputB\$(<number of bytes>, <file number>)

### Parameters <number of bytes>

Numeric expression for the number of bytes to read from the file. <number of bytes> must be between 1 and 2,147,483,647.

The following example opens a file in the current directory, Myfile.txt, reads 51 bytes from that file into a variable, ReadChunk, then closes the file:

Open "Myfile.txt" For Input As #1
ReadChunk = InputB\$(51, #1)
Close #1

### <file number>

Integer that is the file number assigned by Open statement when the target file was opened. File <file number> must be open.

# Returns InputB: Variant

InputB\$: String

Returns all the bytes read, including commas, carriage returns, linefeeds, quotation marks, and leading spaces.

See also Input statement Input, Input\$ functions

# **InStr function**

Returns the starting position of the occurrence of one string within another.

# Syntax Syntax 1

InStr(<string being searched>, <string to find>)

# Syntax 2

InStr(<start>, <string being searched>, <string to find>)

### Parameters <start>

Position (number of characters) within <string being searched> at which to begin searching. The following rules apply for <start>:

- If you do not supply a <start> position, Actuate Basic begins searching at 1, the first character.
- Must be a number or numeric expression.
- Must be between 1 and 2,147,483,647, inclusive.

### <string being searched>

String you are inspecting to locate <string to find>. <string being searched> must be a variable string, a literal string, a string constant, the return value of any function that returns a string, or a variant that can evaluate to a string.

### <string to find>

String you are seeking within <string being searched>. <string to find> must be a variable string, a literal string, a string constant, the return value of any function that returns a string, or a variant that can evaluate to a string.

# Returns Integer

- If <string to find> is found within <string being searched>, InStr returns the position of the first character at which the match was found.
- If <string to find> is not found within <string being searched>, InStr returns 0.
- If <string to find> is zero-length, InStr returns the value of <start>.
- If <string being searched> is zero-length, InStr returns 0.
- If <start> is greater than <string being searched>, InStr returns 0.
- If any parameter evaluates to Null, InStr returns Null.
- **Example** The following example returns the position of a randomly generated character from within the alphabet:

```
Sub Start( )
Dim Alphabet As String, Position As Integer
Dim Msg As String, Letter As String
Super::Start( )
Alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

' Randomly generate a letter to find in the alphabet Letter = Chr\$( Int( 26 \* Rnd + 65 ) ) ' Determine where in the alphabet the letter is positioned Position = InStr( Alphabet, Letter ) Msg = "The letter " & Letter & " is in position " & Position + & " in the alphabet." ShowFactoryStatus( Msg ) End Sub

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Left, Left\$ functions Len function Mid, Mid\$ functions Right, Right\$ functions

# InStrB function

Returns the starting byte of the occurrence of one string within another.

This function is provided for backward compatibility with previous Actuate releases. Actuate uses UCS-2 as internal encoding, therefore passing strings consisting of characters from different code pages might produce unexpected results. For this reason, Actuate recommends you use the InStr function instead of the InStrB function.

# Syntax Syntax 1

InStrB(<string being searched>, <string to find>)

# Syntax 2

InStrB(<start>, <string being searched>, <string to find>)

### Parameters <start>

Position (byte count) within <string being searched> at which to begin searching.

The following conditions apply to <start>:

- If you do not supply a <start> position, Actuate Basic begins searching at 1, the first byte.
- <start> must be a number or numeric expression.
- <start> must be between 1 and 2,147,483,647, inclusive.

### <string being searched>

String you are inspecting to locate <string to find>. <string being searched> must be a variable string, a literal string, a string constant, the return value of any function that returns a string, or a variant that can evaluate to a string.

### <string to find>

String you are seeking within <string being searched>. <string being searched> must be a variable string, a literal string, a string constant, the return value of any function that returns a string, or a variant that can evaluate to a string.

# Returns Integer

- If <string to find> is found within <string being searched>, InStrB returns the position of the first byte at which the match was found.
- If <string to find> is not found within <string being searched>, InStrB returns 0.
- If <string to find> is zero-length, InStrB returns the value of <start>.
- If <string being searched> is zero-length, InStrB returns 0.
- If <start> is greater than <string being searched>, InStrB returns 0.

See also Format, Format\$ functions LeftB, LeftB\$ functions LenB function MidB, MidB\$ functions

# Int function

Returns the largest integer that is less than or equal to a given numeric expression.

**Syntax** Int(<number to round>)

### Parameters <number to round>

Numeric expression from which the fractional part is removed so that an integer value is returned.

**Returns** If <number to round> is a Variant of type 8 (String) that can convert to a number, Int returns Variant of type 5 (Double). If <number to round> is a String, it is parsed according to the formatting rules of the current run-time locale. For example:

```
Fix("123,456") returns 123.456 on a French locale
Fix ("123,456") returns 123456.00 on an English locale
```

- If <number to round> is not a Variant of type 8 (String) that can convert to a number, Int returns the same data type as <number to round>.
- If <number to round> is Null, Int returns Null.

 Int and Fix are similar but not identical. For negative values, Int returns the first negative integer less than or equal to <number to round> while Fix returns the first negative integer greater than or equal to <number to round>. CInt is also similar to Int and Fix. Table 6-28 shows the differences between Int, Fix, and CInt using sample values as arguments.

-	Value	Int(Value)	Fix(Value)	CInt(Value)
_	3.7	3	3	4
	3.2	3	3	3
	3	3	3	3
	-3	-3	-3	-3
	-3.2	-4	-3	-3
	-3.7	-4	-3	-4

 Table 6-28
 Examples of differences between Int, Fix, and CInt

- **Tip** Fix is equivalent to Sgn(<number to round>) \* Int(Abs(<number to round>)).
- **Example** The following example prompts the user for a number and displays the values that Int, Fix, and CInt return:

```
Sub Start( )
Dim Num As Double, Msg As String
Super::Start( )
' Get a random number between 1 and 256
Num = 255 * Rnd + 1
Msg = "Int(" & Num & ") = " & Int(Num)
ShowFactoryStatus( Msg )
Msg = "Fix(" & Num & ") = " & Fix(Num)
ShowFactoryStatus( Msg )
Msg = "CInt(" & Num & ") = " & CInt(Num)
ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also CInt function Format, Format\$ functions RightB, RightB\$ functions

# **IPmt function**

Returns the interest payment for a given period of an annuity, based on periodic, constant payments, and on an unvarying interest rate.

Syntax IPmt(<rate per period>,<single period>, <number pay periods>, <present value>, <future value>, <when due>)

The following conditions apply to IPmt parameters:

- <rate per period> and <number pay period>> must be expressed in terms of the same units (weekly/weeks, monthly/months, yearly/years).
- You must express cash paid out, such as deposits to savings, using negative numbers, and cash received, such as dividend checks, using positive numbers.

#### Parameters <rate per period>

Numeric expression that specifies the interest rate that accrues per period. <rate per period> must be given in the same units of measure as <number pay periods>. For instance, if <number pay periods> is expressed in months, then <rate per period> must be expressed as a monthly rate.

#### <single period>

Numeric expression that specifies the particular period for which you want to determine how much of the payment for that period represents interest. <single period> must be in the range 1 through <number pay periods>.

### <number pay periods>

Numeric expression that specifies the total number of payment periods in the annuity. <number pay periods> must be given in the same units of measure as <rate per period>. For instance, if <rate per period> is expressed as a monthly rate, then <number pay periods> must be expressed in months.

#### <present value>

Numeric expression that specifies the value today of a future payment or stream of payments.

For example, if you put \$23.94 in the bank today and leave it there for 15 years at an interest rate of 10% compounded annually, you end up with about \$100. In this case, the present value of \$100 is approximately \$23.94.

#### <future value>

Numeric expression that specifies the cash balance you want after you have made your final payment.

- **Examples** You set up a savings plan with a goal of having \$75,000 in 18 years to pay for your child's education. For this plan, the future value is \$75,000.
  - You take out a loan for \$11,000. The future value is \$0.00, as it is for any typical loan.

#### <when due>

Numeric expression that specifies whether each payment is made at the beginning (1) or at the end (0) of each period. The default is 0, end of period. <when due> must be 0 or 1.

The following example assumes you are making monthly payments the first of each month on a loan of \$20,000, over 36 months, at an APR of 11.5%. How much of your 5th payment represents interest? The answer, \$171.83, is assigned to Interest5.

Interest5 = IPmt(.115/12, 5, 36, -20000, 0, 1)

### Returns Double

An annuity is a series of cash payments, constant in value, made over a period of time. An annuity can be an investment, such as a monthly savings plan, or a loan, such as a home mortgage. Each payment consists of two components, principal and interest. IPmt returns the interest component of the payment.

# **Example** The following example prompts the user for various particulars about a loan. It then returns the total amount of interest that will be paid on the loan.

To use this example, paste the first portion at or near the beginning of your Actuate Basic source code (.bas) file.

```
Declare
  Global Const ENDPERIOD = 0
  Global Const BEGINPERIOD = 1
End Declare
Sub Start()
  Dim FutVal As Double, PresVal As Double
  Dim APR As Double, TotPmts As Integer
  Dim PayWhen As Integer, Period As Integer, IntPmt As Double
  Dim SumInt As Double, Fmt As String, Payment As Double
  Dim Msg as String
  Super::Start( )
     ' Specify money format
  Fmt = "$###, ###, ##0.00"
  ' Normally 0 for a loan
  FutVal = 0
   ' Amount to borrow
  PresVal = 300000
   ' The annual percentage rate (APR) for your loan
  APR = 0.0625
   ' The number of monthly payments to make
  TotPmts = 240
   ' Assume payment at month end
  PayWhen = ENDPERIOD
```

```
' Do the computational work
  For Period = 1 To TotPmts
     IntPmt = IPmt( APR / 12, Period, TotPmts, -PresVal,
        FutVal, PayWhen )
+
     SumInt = SumInt + IntPmt
  Next Period
  ' Set up display for the user
  Msg = "You will pay a cumulative total of "
     & Format( SumInt, Fmt ) & " in interest for this "
+
     & "loan, which represents about "
+
     & Format( SumInt / PresVal, "##.0%" ) & " of the "
+
     & Format( PresVal, Fmt ) & " you are borrowing."
  ShowFactoryStatus ( Msg )
  Payment = Pmt( APR / 12, TotPmts, -PresVal, FutVal, PayWhen )
  Msg = "Your payments will be " & Format( Payment, Fmt )
     & " per month."
+
  ShowFactoryStatus ( Msg )
End Sub
```

See also FV function NPer function Pmt function PPmt function PV function Rate function

# **IRR** function

Returns the internal rate of return for a series of periodic cash flows, payments and receipts, in an existing array.

Syntax IRR(<casharray>(),<starting guess>)

The following conditions apply to IRR parameters:

- You must express cash paid out, such as deposits to savings, using negative numbers, and cash received, such as dividend checks, using positive numbers.
- <casharray>() must contain at least one negative and one positive number.
- In cases where you have both a positive cash flow (income) and a negative one (payment) for the same period, use the net flow for that period.

If no cash flow or net cash flow occurs for a particular period, you must type 0 (zero) as the value for that period.

# Parameters <casharray>()

Specifies the name of an existing array of Doubles representing cash flow values. The <casharray> array must contain at least one positive value (receipt) and one negative value (payment).

# <starting guess>

Numeric expression. Specifies the value you estimate IRR will return. In most cases, this is 0.1 (10 percent).

The following example assumes you have filled the array MyArray() with a series of cash flow values. The internal rate of return is assigned to the variable IRRValue.

```
IRRValue = IRR(MyArray(), .1)
```

# Returns Double

- The internal rate of return is the interest rate for an investment consisting of payments and receipts that occur at regular intervals. The cash flow for each period does not need to be constant, as it does for an annuity.
- IRR is closely related to the net present value function, NPV, because the rate of return calculated by IRR is the interest rate corresponding to a net present value of zero. IRR calculates by iteration. Starting with the value of <starting guess>, it repeats the calculation until the result is accurate to within 0.00001 percent. If it cannot determine a result after 20 iterations, the function fails.
- **Tips** Because IRR uses the order of values within the array to interpret the order of payments and receipts, be sure to type your payment and receipt values in the correct sequence.
  - If IRR fails, try a different value for <starting guess>.
- **Example** The following example returns the internal rate of return for a series of five cash flows contained in the array, CashValues(). The first array element is a negative cash flow that represents business start-up costs. The remaining four cash flows represent positive cash flows for the subsequent four years. The variable, Guess, holds the estimated internal rate of return. Option Base is assumed to be set to zero.

```
Sub Start( )
    ' Set up the array
    Static CashValues(5) As Double
    Dim Guess As Double, Fmt As String
    Dim ReturnRate As Double, Msg As String
    Super::Start()
    ' Start guess at 10%
    Guess = .1
```

```
' Define % format
Fmt = "#0.00"
' Business start-up costs
CashValues( 0 ) = -80000
' Now type positive cash flows reflecting
' income for four successive years:
CashValues( 1 ) = 23000: CashValues( 2 ) = 27000
CashValues( 3 ) = 31000: CashValues( 4 ) = 35000
' Calculate internal rate
ReturnRate = IRR( CashValues, Guess ) * 100
Msg = "The internal rate of return for the cash flows is: "
+ & Format( ReturnRate, Fmt ) & " percent."
ShowFactoryStatus( Msg )
End Sub
```

See also MIRR function NPV function Option Base statement Rate function

# **IsDate function**

Determines whether the given argument can be converted to a date.

Syntax IsDate(<exprs>)

# Parameters <exprs>

Variant expression to test to determine whether it is a date, can be evaluated as a date, or can be converted into a date.

The following conditions apply to <exprs>:

- The range of valid dates is January 1, 100, through December 31, 9999, inclusive.
- Numbers can be converted to a date.
- <exprs> is parsed according to the formatting rules of the current run-time locale.

# Returns Integer

- Returns 1 (True) if <exprs> can be converted to a date. Otherwise, returns 0 (False).
- If <exprs> fails to include all date components, such as day, month, and year, IsDate returns False.

- **Tips** To convert any number between -657,434 and 2,958,465 into a date, use CVDate. For example, use CVDate(34040.34) to return 3/12/93 7:55:12 A.M.
  - If you use a date expression in one locale, it might be misinterpreted in another locale. For instance, in the United States, 1/2/2005 means January 2nd, 2005, but in France, it means February 1st, 2005. To avoid such ambiguities, use DateSerial to specify all your dates.

**Examples** The following statements are equivalent. They all return True.

```
IsDate("11/12/82")
IsDate("November 12, 1982")
IsDate("Nov 12, 1982")
IsDate("N" & "ov " & "12, " & (1980 + 2))
IsDate(5)
IsDate(5.3)
```

The following example tests today's date and a number and, if possible, converts the number to a date:

```
Sub Start()
  Dim DateStr As String, Msg As String
  Dim Number As String
  Super::Start( )
  Number = CStr(4000000 * Rnd)
  DateStr = CStr(Now)
  ' Test the string
  If IsDate(DateStr) Then
     Msg = "Today's date is: "
     Msg = Msg & Format( CVDate( DateStr ), "dddddd" )
     ShowFactoryStatus ( Msg )
  End If
  ' Test the number
  If IsDate( Number ) Then
     Msg = "The number " & Number & " as a date is: " &
        Format( CVDate( Number ), "dddddd" )
+
     ShowFactoryStatus ( Msg )
  Else
     If IsNumeric (Number ) Then
        If Val(Number) >= -657434
       And Val( Number ) <= 2958465 Then
+
          Msg = "The number " & Number & " is not in "
             & "date format. However, it will be "
+
             & "interpreted as the date:"
+
             & Format ( CVDate ( Number ), "dddddd" )
+
          ShowFactoryStatus ( Msg )
```

+

```
Else
          Msg = "Sorry, the number " & Number &
             " cannot be converted to a date."
          ShowFactoryStatus( Msg )
        End If
     End if
  End If
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also **CVDate function** IsEmpty function IsNull function IsNumeric function VarType function

# IsEmpty function

Reports whether the given variable has been initialized.

```
Syntax
        IsEmpty(<exprs>)
```

#### Parameters <exprs>

Variant expression to test whether it has ever been assigned a value.

#### Returns Integer

- Returns 1 (True) if <exprs> contains the Empty value. Otherwise, returns 0 (False).
- Returns 0 (False) if <exprs> contains more than one Variant.
- Tip Empty and Null are different. A variable can contain the Null value only if you explicitly assign Null to the variable.
- Example The following example tests a variable to determine whether it has ever held a value. If not, it reports that the variable has not been initialized and asks the user whether to initialize it.

```
Sub Start()
  Dim MyVar, Msg As String
  Super::Start( )
  'Test MyVar
  If IsEmpty( MyVar ) Then
     ShowFactoryStatus( "MyVar is uninitialized." )
```

```
' Initialize MyVar
MyVar = 32.5
ShowFactoryStatus( "MyVar set to " & MyVar )
ShowFactoryStatus( "The value of IsEmpty( MyVar ) is now:"
+ & IsEmpty( MyVar ) )
Else
ShowFactoryStatus( "MyVar initialized to " & MyVar )
End If
End Sub
```

See also IsDate function IsNull function IsNumeric function VarType function

# IsKindOf function

Determines if a specified object is an instance of a specified class.

Syntax IsKindOf(<object>, <class>)

### Parameters <object>

The variable name that refers to the object about which you want information.

### <class>

The class to test for the membership of an instance of <object>.

### **Returns** Integer

- Returns 1 (True) if <object> is a member of <class>.
- Returns 0 (False) if <object> is not a member of <class>.
- **Tips** Use IsKindOf to test if an object is an instance, or an instance of a subclass, of a specified class.
  - To find out an object's class, use GetClassName.
- **Example** The following example tests if an object is either an instance of MyLabel or an instance of a subclass of MyLabel. If True, the example sets the object's background color to red.

```
Sub Start( )
  Dim MyLabel As AcLabelControl, Msg As String
  Super::Start( )
  Set MyLabel = New AcLabelControl
```

```
If IsKindOf(MyLabel, "AcLabelControl") Then
     Msg = "Yes, MyLabel is an AcLabelControl.
        & "We will now set its background color to red."
     ShowFactoryStatus( Msg )
     MyLabel.BackgroundColor = Red
  End If
End Sub
```

GetClassID function See also GetClassName function

# IsNull function

+

Determines whether the given Variant or Variant expression contains the Null value.

Syntax IsNull(<exprs>)

#### Parameters <exprs>

Variant expression to test. Any expression that contains a Null is itself Null and therefore False. That is why both the following expressions evaluate to False, no matter what value MyVar contains:

If MyVar = Null If MyVar <> Null

For example, assuming you have assigned the Null value to MyVar, the following statements are equivalent. Each returns 1 (True).

```
IsNull(MyVar)
IsNull(MyVar = Null)
IsNull(MyVar <> Null)
IsNull(MyVar + 5000)
```

#### Returns Integer

- Returns 1 (True) if <exprs> contains the Null value. Otherwise, returns 0 (False).
- The only way to determine whether a variable contains Null is to use IsNull.
- Tip Empty and Null are different. If a variable contains the Empty value, it means it has never yet been assigned any value. A variable can contain the Null value only if you have explicitly assigned Null to it.

Example The following example tests MyVar to determine whether it contains the Null value. The variable MyVar is first initialized to Null but the first time though the
loop it is changed to a zero-length string. The second time through the loop, MyVar is again set to Null and the loop exits.

```
Sub Start()
  Dim MyVar
  Super::Start( )
  ' Initialize variable as Null
  MyVar = Null
  Do
   ' Evaluate variable
     If IsNull( MyVar ) Then
        ' Report if Null
        ShowFactoryStatus( "MyVar is Null." )
        ' Assign zero-length string
        MyVar = ""
     Else
        ' Report if not Null
        ShowFactoryStatus( "MyVar is not Null." )
        ' Assign Null to variable
        MyVar = Null
     End If
   ' Loop until MyVar is Null
  Loop Until IsNull ( MyVar )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also IsDate function IsEmpty function IsNumeric function VarType function

### **IsNumeric function**

Tests if the type of a variable is or can be converted to Integer, Long, Single, Double, or Currency.

Syntax IsNumeric(<exprs>)

#### Parameters <exprs>

Variant expression to test to determine if it is, can be evaluated as, or can be converted to a numeric data type. For example, the following are equivalent. Each returns True.

```
IsNumeric("34040")
IsNumeric(34040)
IsNumeric(Val("343Haydn"))
```

If <exprs> is a String, it is parsed according to the formatting rules of the current run-time locale.

```
Returns Integer
```

- Returns 1 (True) if <exprs> can be converted to a number. Otherwise, returns 0 (False).
- Returns 1 (True) if <exprs> is Empty.
- **Tip** To extract a numeric value from a string that starts with numbers but also contains non-numeric characters, use Val.
- **Example** The following example inspects each of four elements in an array of mixed types and reports whether or not the content of the variable is numeric:

```
Sub Start()
  Dim UserVar(4), Msg, Slot As Integer
  Super::Start( )
  UserVar(1) = 12
  UserVar(2) = "twelve"
  UserVar(3) = "123four"
  UserVar(4) = Val("123four")
  For Slot = 1 to 4
     If IsNumeric( UserVar( Slot ) ) Then
       Msg = UserVar( Slot ) & " is Numeric."
     Else
       Msg = UserVar( Slot ) & " is not Numeric."
     End If
     ShowFactoryStatus ( Msg )
  Next Slot
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also IsDate function IsEmpty function IsNull function Val function VarType function

### **IsPersistent function**

Reports if an instance of an object is persistent or transient.

**Syntax** IsPersistent(<object>)

#### Parameters <object>

The variable name that refers to the object about which you want information.

Returns Integer

- Returns 1 (True) if <object> is persistent.
- Returns 0 (False) if <object> is transient.
- **Example** The following example determines if specific objects are either transient or persistent:

```
Sub Start()
  Dim myObj As AcLabelControl, Msg As String
  Super::Start( )
  ' Instance instantiation
  Set myObj = NewInstance( "AcLabelControl" )
  ' Determine persistent or transient nature of objects
  If ( IsPersistent( me ) ) Then
     Msg = "The current object is persistent. "
  Else
     Msg = "The current object is transient. "
  End If
  if ( IsPersistent ( myObj ) ) Then
     Msg = Msg & "myObj is persistent."
  Else
     Msg = Msg & "myObj is transient."
  End If
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also NewInstance function NewPersistentInstance function

### IsSearchFormatSupported function

Determines whether the specified search format is supported or not.

Syntax IsSearchFormatSupported(<format> As String) As Boolean

#### Parameters <format>

The format for which to determine support. The supported formats depend on the formats available in a particular environment. For example, the following search formats might be supported:

- XMLDisplay (DISPLAY)
- Download formats:
  - CSV

	- TSV	
	ANALYSIS	
Returns	True if the format is supported.	
	False if the format is not supported.	
Example	The following example determines whether the ANALYSIS format is support	
	Function CreateSearchURL(format As String) As String Dim URL As String Dim format As String Dim searchCriteria As String	
	<pre>SearchCriteria = "" format = "ANALYSIS" If Not IsSearchFormatSupported( format ) Then     format = "CSV" End If URL = "http://" format=" + format + searchCriteria CreateSearchURL = URL End Function</pre>	

See also GetSearchFormats function

## IsViewPageFormatSupported function

Determines whether the specified view format is supported.

Syntax IsViewPageFormatSupported(<format> As String) As Boolean

```
Parameters <format>
```

The DHTML viewing format for which to determine support. The supported formats depend on the formats available in a particular environment. For example, the following viewing formats might be supported:

- XMLDisplay (DISPLAY)
- DHTML
- PDF

**Returns** True if the format is supported.

False if the format is not supported.

**Example** The following example determines whether the PDF format is supported:

Function CreateSearchURL( format As String ) As String Dim URL As String Dim format As String

```
Dim searchCriteria As String
SearchCriteria = "..."
format = "PDF"
If Not IsViewPageFormatSupported( format ) Then
format = "DHTML"
End If
URL = "http://" format=" + format + searchCriteria
CreateSearchURL = URL
End Function
```

See also GetViewPageFormats function

## Kill statement

Deletes a specified file or files from a disk.

Syntax Kill <file spec>

**Description** Kill is similar to the operating system commands ERASE and DEL.

#### Parameters <file spec>

String expression that specifies the name or names of the file or files to be deleted. Wildcard characters can be used. Can include optional drive and path information. The default path is the current default drive and directory.

The following rules apply to <file spec>:

- Indicated file or files must exist.
- Path, if specified, must exist.
- Indicated file must not currently be open by Actuate Basic.

<file spec> may optionally specify full path information, in which case it has the following syntax:

[<drive:>] [ \ ]<directory>[\<directory>]...<file spec> (Windows)

[/]<directory>[/<directory>]...<file spec> (UNIX)

#### <drive:>

Character, followed by a colon, that specifies the drive where <file spec> is located (Windows only).

#### <directory>

String expression that specifies the name of a directory or subdirectory where <file spec> is located.

For example, the following statement deletes all files in the Windows directory C:\Discard that have an extension of .doc:

Kill "C:\Discard\\*.doc"

- **Tips** Be careful in how you use Kill, especially if you include wildcard characters in <file spec>. It is easy to delete files unintentionally, and Kill is irreversible.
  - To delete directories, use RmDir.
- **Example** The following example copies a file from the Actuate install directory to the root directory on the C: drive, and then deletes it. If the example cannot perform the operation, it issues an error message.

```
Sub Start()
  Dim SourceFile As String, DestFile As String, Msg As String
  Super::Start( )
   ' Set up error handler
  On Error GoTo Errhandler55
  ' Copy file from installation to be deleted
  SourceFile = "C:\Program Files\Actuate11\readme.rtf"
  DestFile = "C:\actuatekill readme.rtf"
  FileCopy SourceFile, DestFile
  Msq = "File C:\actuatekill readme.rtf is now on your system."
  ShowFactoryStatus ( Msg )
  ' Delete the file
  Kill DestFile
  Msq = "File C:\actuatekill readme.rtf has been deleted."
  ShowFactoryStatus ( Msg )
  Exit Sub
Errhandler55:
  Msg = "Error number " & Err & " occurred. " & Error$(Err)
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Name statement

### LBound function

Returns the smallest available subscript for the given dimension of an array.

Syntax LBound(<array name>[,<dimension>])

#### Parameters <array name>

Name of an array variable.

#### <dimension>

Numeric expression that specifies the array dimension for which you want to determine the lower bound.

The following rules apply to <dimension>:

- If you do not supply a <dimension>, Actuate Basic defaults to 1.
- Use 1 for the first dimension, 2 for the second, and so on.
- Must not be larger than the total number of dimensions in <array name>.
- Must not be zero, negative, or Null.

Table 6-29 shows the values LBound returns for an array with the following dimensions:

```
Dim MyArray(1 To 55, 0 To 27, -3 To 42)
```

**Table 6-29**Examples of LBound return values for an array

Statement	Returned value
LBound(MyArray, 1)	1
LBound(MyArray, 2)	0
LBound(MyArray, 3)	-3

#### Returns Integer

If <dimension> is not an Integer, LBound rounds <dimension> to the nearest Integer before evaluating it.

- **Tips** To determine the upper bound of an array, use UBound.
  - To determine the total number of elements in a given dynamic array dimension, take the value returned from UBound, subtract the value returned from LBound, and then add 1.
- **Example** The following example determines the lower bounds of an array of 3 dimensions. The use of Rnd simulates changes in lower bounds that the user can make at run time.

```
Sub Start()
Dim First As Integer, Sec As Integer, Third As Integer
Dim Msg As String
' Declare array
Dim MyArray()
Super::Start()
' Generate random dimensions between 3 and 16 for array size.
' First, second, and third dimension
First = Int(14 * Rnd + 3)
Sec = Int(14 * Rnd + 3)
Third = Int(14 * Rnd + 3)
```

```
' Set dimensions
ReDim MyArray(First To 30, Sec To 30 , Third To 30)
Msg = "MyArray has the following lower bounds: "
+ & Tab & "Dimension 1 -> " & LBound(MyArray, 1)
+ & Tab & "Dimension 2 -> " & LBound(MyArray, 2)
+ & Tab & "Dimension 3 -> " & LBound(MyArray, 3)
ShowFactoryStatus( Msg )
End Sub
```

See also Dim statement UBound function

## LCase, LCase\$ functions

Converts all uppercase alphabetic characters in a string to lowercase.

Syntax LCase(<string exprs>)

LCase\$(<string exprs>)

#### Parameters <string exprs>

String expression to convert to lowercase characters.

For example, the following statements are equivalent. Each returns 6 jane street, 2nd fl.

```
LCase$("6 JANE STREET, 2ND FL")
LCase$("6 Jane Street, 2nd Fl")
LCase$("6 jAn" & "E sTreeT, 2nD fL")
```

Returns LCase: Variant LCase\$: String

- If <string exprs> contains no uppercase alphabetic characters, LCase[\$] returns <string exprs> unchanged.
- LCase[\$] has no effect on non-alphabetic characters in <string exprs>.
- If <string exprs> evaluates to Null, LCase[\$] returns Null.
- **Tips** To ensure uniformity in the data you get from the user so that, for example, name strings like MacManus, Macmanus, or macMaNus are always treated in the same way, first convert strings using LCase[\$] or UCase[\$].
  - To convert alphabetic characters in <string exprs> to uppercase, use UCase[\$].

**Example** The following example renders the alphabet in lowercase:

```
Sub Start( )
Dim LowerCase As String, UpperCase As String, Msg As String
Super::Start( )
UpperCase = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
' Convert to lowercase
LowerCase = LCase$( UpperCase )
Msg = "LCase$ converts """ & UpperCase & """ to """
+ & LowerCase & """."
ShowFactoryStatus( Msg )
End Sub
```

See also UCase, UCase\$ functions

## Left, Left\$ functions

Returns a segment of a Variant or String, starting at the first character.

**Syntax** Left(<string exprs>, <length>)

Left\$(<string exprs>, <length>)

#### Parameters <string exprs>

Source string from which you are copying the first portion. Can be a variable string, a literal string, a string constant, the return value of any function that returns a String, or a Variant that can evaluate to a String.

#### <length>

Numeric expression that specifies how many characters to copy from the left of <string exprs>. <length> must be an Integer or expression of type Long, and must be between 0 and 2,147,483,647.

The following statements are equivalent. Both return Widget:

```
Left$("Widget",6)
Left$("Widget",99)
```

#### Returns Left: Variant

Left\$: String

- If <length> = 0, returns a zero-length string.
- If <length> is greater than or equal to the length of <string exprs>, returns an exact copy of <string exprs>.
- If any parameter evaluates to Null, Left[\$] returns Null.

**Tips** Use Len to find the number of characters in <string exprs>.

Use InStr to find the position of a specified character in <string exprs>.

```
Example The following example parses a string for a customer's first and last names:
```

```
Sub Start( )
  Dim FName As String, Msg As String, LName As String
  Dim SpacePos As Integer, Customer As String
  Super::Start( )
  Customer = "Manuel Barajas"
' Find the space
  SpacePos = InStr(1, Customer, " ")
  If SpacePos Then
     ' Get first and last name
     FName = Left$(Customer, SpacePos - 1)
     LName = Right$(Customer, Len(Customer) - SpacePos)
     Msg = "The first name is """ & FName & "." & """"
       & " The last name is """
+
       & LName & "." & """"
+
  Else
     Msg = Customer & " does not have a first and last name!"
  End If
  ' Display the message
ShowFactoryStatus ( Msg )
End Sub
```

See also Format, Format\$ functions Len function Mid, Mid\$ functions Right, Right\$ functions

## LeftB, LeftB\$ functions

Returns a segment of a Variant or String, starting at the first byte.

These functions are provided for backward compatibility with previous Actuate releases. Actuate uses UCS-2 as internal encoding, therefore passing strings consisting of characters from different code pages might produce unexpected results. For this reason, Actuate recommends you use the Left and Left\$ functions instead of the LeftB and LeftB\$ functions.

Syntax Left(<string exprs>, <length>)

Left\$(<string exprs>, <length>)

#### Parameters <string exprs>

Source string. Can be a variable string, a literal string, a string constant, the return value of any function that returns a String, or a Variant that can evaluate to a String.

#### <length>

Numeric expression that specifies how many bytes to copy from the left of <string exprs>. <length> must be an Integer or expression of type Long, and must be between 0 and 2,147,483,647.

#### Example:

The following statements are equivalent. Both return Widget.

```
LeftB$("Widget",6)
LeftB$("Widget",99)
```

#### Returns LeftB: Variant

LeftB\$: String

- If <length> = 0, the function returns zero-length string.
- If <length> is greater than or equal to the length of <string exprs>, the function returns an exact copy of <string exprs>.
- If any parameter evaluates to Null, LeftB[\$] returns Null.
- **Tips** Use LenB to find the number of bytes in <string exprs>.
  - Use InStrB to find the position of a specified byte in <string exprs>.

#### See also InStrB function

Left, Left\$ functions LenB function MidB, MidB\$ functions RightB, RightB\$ functions

## Len function

Returns the number of characters in a string expression.

Len( <string exprs="">)</string>
Len( <variable name="">)</variable>
<string exprs=""> String expression to test for character length.</string>

#### <variable name>

Variable to test for character length.

The following example returns 11:

Len("Smith, John")

The following example, which tests MyVar, a 4-byte Integer, returns 4:

Len(MyVar)

#### **Returns** Integer

- Regardless of its value, if <variable name> is of any numeric data type except Variant, Len returns the number of characters required to store any variable of that data type.
- Regardless of its Variant type, if <variable name> is a Variant, Len returns the number of characters required to store it as a String.
- If <string exprs> or <variable name> is Null, Len returns Null.
- **Tip** Use Len to determine record size when you perform file input/output with random-access files using user-defined variable types. Actuate Basic does not notice a type mismatch if you dimension a variable, such as UserInput, as a String, and then later test it as an Integer, such as if UserInput = 0.
- **Example** The following example returns the length of a name:

```
Sub Start( )
Dim AName As String, Msg as String
Super::Start( )
AName = "Wang Xiaokun"
Msg = "The name " & AName & " is " & Len(AName)
+ & " characters long, including the space(s)."
ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also LenB function Type...End Type statement VarType function

## LenB function

Returns the number of bytes in a string expression.

This function is provided for backward compatibility with previous Actuate releases. Actuate uses UCS-2 as internal encoding, therefore passing strings consisting of characters from different code pages might produce unexpected results. For this reason, Actuate recommends you use the Len function instead of the LenB function.

Syntax LenB(<string exprs>)

LenB(<variable name>)

#### Parameters <string exprs>

String expression to test for byte length.

<variable name>

Variable to test for byte length.

#### Returns Integer

- Regardless of its value, if <variable name> is of any numeric data type except Variant, LenB returns the number of bytes required to store any variable of any value that is of the same data type.
- Regardless of its Variant type, if <variable name> is Variant, LenB returns the number of bytes required to store it as a String.
- If <string exprs> or <variable name> is Null, LenB returns Null.
- **Tip** To determine record size when performing file input/output with random-access files using user-defined variable types, use LenB.

See also Len function Type...End Type statement VarType function

## Let statement

Assigns a value to a variable.

Syntax [Let] <variable> = <exprs>

- **Description** In some older versions of Basic, the Let keyword is required to begin a statement that assigns a value to a variable. In Actuate Basic, the use of Let is optional. The following conditions apply to the Let statement:
  - If <variable> is a user-defined type declared with Type, <exprs> must be of the same user-defined type.
  - The data types of <variable> and <exprs> must be compatible.
  - In addition to the data type compatibilities listed in Table 6-30, you can assign Null to numeric, String, Variant, or CPointer variables.

Table 6-30	Compatibility of data types for <variable> and <exprs></exprs></variable>	
Data type of	<variable></variable>	Valid data types for <exprs></exprs>
Numeric		Numeric Variant that converts to a numeric data type

#### Let statement

<exprs> (continued)</exprs>		
Data type of <variable></variable>	Valid data types for <exprs></exprs>	
String	String Non-Null Variant	
Variant	String Numeric Variant	
CPointer	CPointer	

# Table 6-30 Compatibility of data types for <variable> and <exprs> (continued)

#### Parameters <variable>

The name of any variable to store <exprs>.

#### <exprs>

String or numeric expression to assign to <variable>.

The following rules apply to <exprs>:

- If <exprs> is Variant, and <variable> is numeric, Actuate Basic converts <exprs> to a number, if possible, before assigning it to <variable>.
- If <exprs> is numeric, its value must fall into the range of the data type of <variable>.

For example, the following statements are equivalent:

```
Let N = 50
N = 50
```

The following example generates an error, because 2,147,483,647 is the upper limit of the range of the Integer data type variable to which it is being assigned:

```
N% = 2147483648
```

- **Tips** Use IsNumeric to determine if a Variant variable converts to numeric.
  - To make assignments between variables of different user-defined types that have been defined by Type...End Type, use LSet or the = operator. Using the = operator to assign an <exprs> of one data type to a <variable> of a different type converts the value of <exprs> into the data type of <variable>.
- **Example** The following example uses Let to assign a value to Pi:

```
Sub Start( )
Dim Msg As String, Pi As Double
Super::Start( )
' Assign a value to Pi with the Let statement
Let Pi = 4 * Atn(1)
' Assign a value to Msg without Let statement
```

```
Msg = "The area of a circle of radius 3 inches is "
+ & (Pi * (3^2)) & " inches."
ShowFactoryStatus( Msg )
End Sub
```

See also Const statement IsNumeric function LSet statement

### Line Input statement

Reads from a sequential file until it reaches a carriage-return-linefeed (CRLF) pair or the end of the file (EOF), and assigns the information to a variable.

Syntax Line Input #<file number>, <receiving variable>

**Description** Line Input reads data from a sequential file one line at a time.

Unlike Input, Line Input treats commas, spaces, and quotes no differently than it does any other characters. It is useful for reading plain ASCII files. Line Input stops reading into <receiving variable> when it encounters a carriage-return-linefeed character-pair (CRLF). It skips the carriage-return-linefeed character-pair itself and—assuming it is called again—continues reading from the first character after the CRLF until it reaches another CRLF or the end of the file (EOF).

#### Parameters <file number>

Numeric expression that is the file number assigned to the target file when it was opened in the Input mode.

#### <receiving variable>

String or variant expression to which Line Input assigns the data it reads from the target file.

The following example reads one line of data from Notes.txt into the variable A\$:

Open "Notes.txt" For Input As #2 Line Input #2, A\$

The following example reads the first line of Poem.txt into the variable A\$, the second line into B\$, and the third into C\$:

```
Open "Poem.txt" For Input As #1
Line Input #1, A$
Line Input #1, B$
Line Input #1, C$
```

**Rules** • <file number> must match the number of a currently open file.

- The number sign (#) preceding <file number> is required.
- The file corresponding to <file number> must be open under the Input mode.
- You must have write access to the open file. That is, the file must not have been opened using a Lock Read or Lock Read Write clause.
- **Example** The following example creates a test file, reads three lines from it, then deletes the test file:

```
Sub Start()
  Dim strEachLine(3) as String
  Dim I As Integer
  Super::Start ()
  Open "LineInputNum.txt" For Output As #1
  For I = 1 to 3
     Print #1, "This is line number " & I & " in the test file"
  Next I
  Reset
  Open "LineInputNum.txt" For Input As #1
  For I = 1 to 3
     Line Input #1, strEachLine(I)
     ShowFactoryStatus( strEachLine(I) )
  Next I
  Reset
  ShowFactoryStatus( "Deleting test file." )
  Kill "LineInputNum.txt"
End Sub
```

See also Input statement Open statement Print statement

## ListToArray function

Converts a list of values into an array.

Syntax ListToArray(<list>, <array>)

ListToArray(<list>, <array>, <separator>)

- **Description** Use ListToArray to convert string lists with list elements delimited by separator characters into arrays.
- **Parameters** <list> String expression specifying the list to convert to an array.

#### <array>

Dynamic string array to populate with the elements of the list. This array is resized to exactly the number of elements found in the list. The base of the array is 1.

#### <separator>

Optional string expression specifying a single character used to delimit the elements in the list. If omitted or the value is an empty string or Null, a comma is used as the separator character. If the value is more than one character, only the first character is used.

- **Returns** Integer. The number of elements in the list and hence the number of elements in the array.
  - If the list is an empty string, ListToArray returns 1 and the array contains a single empty element.
  - If the list is Null, ListToArray returns 0 and the array is reset to contain no elements.
  - **Tip** String lists are frequently found in external data, such as parameters and database columns. Use arrays as a more convenient format for calculations.

#### **Example** The following example declares a dynamic array and converts a list into an array:

```
Sub Start( )
Dim list As String
Super::Start ( )
List = "A|B|C"
' Declare a dynamic array.
Dim array() As String
Dim numberOfElements As Integer
numberOfElements = ListToArray( list, array, "|" )
Dim i As Integer
For i = 1 To numberOfElements
ShowFactoryStatus( "Element " & I & " = " & array( i ) )
Next i
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

### Loc function

Returns the current position in an open file.

Syntax Loc(<file number>)

#### Parameters <file number>

Numeric expression that is the file number used to open the file with Open statement.

Rule: File must be open.

- Returns Integer
  - For Sequential files: Returns the current byte position in the file divided by 128.
  - For Random files: Returns the number of the last variable read from or written to the file.
  - For Binary mode files: Returns the position of the last byte read or written.
  - **Tip** You can open sequential files with a buffer of a different size if you use the Len parameter in Open. However, this does not change the result of Loc, which always divides by 128 when applied to sequential files.
- **Example** The following example opens a test file, asks the user how much of it to read, then reports the current position in the file:

```
Sub Start()
  Dim CountVar As Integer, Msg As String
  Dim NowLoc As Integer, TempVar, MaxNum
  Super::Start( )
  ' Open file for output
  Open "Testdata.fil" For Output As #1
' Generate rnd values
  For CountVar = 1 To 300
' Put data in file
     Print #1, CountVar
  Next CountVar
' Close test file
  Close #1
' Open file just created
  Open "Testdata.fil" For Input As #1
  MaxNum = CInt(299 * Rnd + 1)
  For CountVar = 1 To MaxNum
' Read some data from it
     Input #1, TempVar
  Next CountVar
' Find location in file
  NowLoc = Loc(1)
' Close file
  Close #1
  Msg = MaxNum & " data elements have been read from a file of"
   & " 300. The current location in the Testdata.fil"
+
     & " file is: " & NowLoc & ". "
+
```

+ & "The sample file will now be deleted."
ShowFactoryStatus( Msg )
' Delete file from diskKill "Testdata.fil"
End Sub

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also EOF function LOF function Open statement

### Lock...Unlock statement

Prohibits access by other users or processes to all or part of an open file.

Syntax Lock [#]<open file number> [,{<record> | [<start>] To <end>}]

[<data process statements>]

Unlock [#]<open file number> [,{<record> | [<start>] To <end>}]

**Description** Lock restricts multiuser access to a specified area of a file. When you lock part of the file or the whole file, no other user on a network can access it until you Unlock the file or file segment.

Lock and Unlock apply only to files opened with Open by Actuate Basic. They do not apply to Access databases.

The following conditions apply to Lock...Unlock:

- The parameters for Lock and Unlock in a Lock...Unlock statement must match exactly.
- If you use Lock to restrict access to a file, you must Unlock the file before you close that file or terminate your program.
- You cannot use Lock...Unlock with versions of MS-DOS earlier than 3.1.
- If you do use MS-DOS, and Share.exe is not already loaded, you must exit Actuate Basic and run Share.exe under DOS to enable locking operations before you can use Lock...Unlock.

#### Parameters <open file number>

Numeric expression that is the file number you used to open the target file. The following conditions apply to <open file number>:

- Must be the number of a currently open file.
- The number sign (#) before <open file number> is required.

#### <record>

Numeric expression that specifies the number of the record or byte to lock. The following conditions apply to <record>:

- Must be between 1 and 2,147,483,647.
- Record length cannot exceed 65,535 bytes.

#### <start>

Numeric expression that specifies the number of the first record or byte to lock. The default is 1.

#### То

Keyword that separates <start> from <end> in the clause that specifies a range of records or bytes to lock.

#### <end>

Numeric expression that specifies the number of the last record or byte to lock.

The terms <record>, or <start> and <end> specify different things according to the mode under which the target file was opened. Table 6-31 summarizes these differences.

File mode	<record>, or <start> and <end> refer to</end></start></record>
Binary	The number of a byte relative to the beginning of the open file. The first byte in a file is byte 1.
Random	The number of a record relative to the beginning of the open file. The first record is record 1.
Sequential Input or Output	The entire file, no matter what range you specify with <start> To <end>.</end></start>

Table 6-31 Differences between using <record> and using <start> and <end>

#### <data process statements>

One or more valid Actuate Basic statements that direct the operations your program performs on the open file while others are temporarily locked out. In the following example, Lock restricts other users or processes from accessing any part at all of the file previously opened as #1. When processing of the file is done, Unlock lifts those restrictions.

```
Lock #1
[<data process statements>]
Unlock #1
```

The following Lock...Unlock statement block prohibits and then permits others' access to a single record, the 112th in the random access file Maillist.dat:

```
Open "Maillist.dat" for Random As #1 Len = 32
Lock #1, 112
[<data process statements>]
Unlock #1, 112
```

The following Lock...Unlock statement block locks and then unlocks others' access to byte positions 11 through 25, inclusive, of the binary file Testfile.dat:

```
Open "Testfile.dat" For Binary As #1
Lock #1, 11 TO 25
[<data process statements>]
Unlock #1, 11 TO 25
```

The following Lock...Unlock statement block locks and then unlocks byte positions 1 through 229, inclusive, of the binary file Testfile.dat:

```
Open "Testfile.dat" For Binary As #1
Lock #1, TO 229
[<data process statements>]
Unlock #1, TO 229
```

## Tips Use Lock...Unlock in networked environments, in which several users or processes might need access to the same file.

- Use Lock...Unlock to ensure that two users simultaneously updating a database record do not accidentally save old data on top of new.
- **Example** The following example creates a sample data file and accesses a single record, RecNum. RecNum is locked to other users while the record is being updated. Access is permitted again when the update is complete. The example overrides Start() and calls a sub procedure to create the sample data file. The data file is deleted at the conclusion of the Start function. To use this example, paste the Declare statement and the procedure MakeDataFile1216 into a Actuate Basic source code (.bas) library file.

```
Declare

Type AcctRecord

Payer As String

Address As String

City As String

State As String

Owe As Currency

End Type

End Declare

Sub MakeDataFile1216()

Dim CustRecord As AcctRecord

Open "Testfile" For Random Shared As #1 Len = 150

' Put information in each field of the record

CustRecord.Payer = "Emilio Enbilliam"

CustRecord.Address = "6 Jane Street"
```

```
CustRecord.City = "New York"
  CustRecord.State = "NY"
  ' Initialize amount owed
  CustRecord.Owe = 12
  'Put record in file
  Put #1, 1, CustRecord.Payer & "," & CustRecord.Address & ","
+ & CustRecord.City & "," & CustRecord.State & "," &
  CustRecord.Owe
   ' Close the file
Close #1
End Sub
Sub Start()
  Dim CustRecord As AcctRecord, Change As String
  Dim DataRecord As Variant
  Dim Msq As String, RecNum As Integer
  Dim commaPos1, commaPos2 as Integer
  Super::Start( )
  ' Set up error handler
  On Error GoTo ErrorHandler
  ' Create sample data file
  MakeDataFile1216
  Open "Testfile" For Random Shared As #1 Len = 150
  ' There is only one record, lock the current record
  RecNum = 1
  Lock #1, RecNum
  ' Read a record
  Get #1, RecNum, DataRecord
  commaPos1 = InStr(DataRecord, ", ")
  CustRecord.Payer = Mid(DataRecord, 1, commaPos1-1)
  commaPos2 = Instr(commaPos1+1, DataRecord, ",")
    CustRecord.Address = Mid(DataRecord, commaPos1+1, commaPos2-
  commaPos1-1)
  commaPos1 = commaPos2 + 1
  commaPos2 = Instr(commaPos1, DataRecord, ",")
  CustRecord.City = Mid(DataRecord, commaPos1, commaPos2-commaPos1)
  commaPos1 = commaPos2 + 1
  commaPos2 = Instr(commaPos1, DataRecord, ",")
  CustRecord.State = Mid(DataRecord, commaPOs1, commaPos2-
  commaPos1)
  CustRecord.Owe = Mid(DataRecord, RevInStr(DataRecord, ", ")+1)
  Msg = "Customer " & CustRecord.Payer
   & " previously owed: "
     & Format(CustRecord.Owe, "$#,##0.00")
  ' Show data and the change
  Change = 1000 * Rnd - 500
  If Len(Change) = 0 Then Change = 0
  CustRecord.Owe = CustRecord.Owe + Change
```

```
' Update record
  Put #1, RecNum, CustRecord.Payer & "," & CustRecord.Address
+ & "," & CustRecord.City & "," & CustRecord.State & "," &
  CustRecord.Owe
  ' Unlock the record
  Unlock #1, RecNum
  Msg = Msg & " The change to the amount is "
   & Format(Change, "$#,##0.00")
+
   & " The amount is now "
+
     & Format (CustRecord.Owe, "$#,##0.00")
+
  ShowFactoryStatus ( Msg )
  ' Close the file
  Close #1
Cleanup:
  Msg = "Transaction complete. Now Deleting sample "
+
     & "data file."
  ShowFactoryStatus ( Msg )
  ' Remove file from disk
' Kill "TESTFILE"
Exit Sub
ErrorHandler:
' Permission denied error
  If Err = 70 Then
     Msg = {"Sorry, you must run Share.exe before running
this example. Exit Actuate Basic, Exit Windows, run Share.exe,
and reenter Actuate Basic torun this example. Do not shell to
DOS and run Share.exe or you might not be able to run other
programs until you reboot." }
' Some other error occurred
  Else
     Msg = "Error " & Err & " occurred. " & Error$
  End If
  ' Display error message
  ShowFactoryStatus ( Msg )
  ' Close files, flush buffers
  Reset
  ' Do an orderly exit
Resume Cleanup
End Sub
```

```
See also Get statement
Open statement
Put statement
```

## LOF function

Returns the size of an open file in bytes.

**Syntax** LOF(<open file number>)

#### Parameters <open file number>

Numeric expression that is the file number you used with Open to open the target file.

Rules for <open file number>:

- Must be the number of a currently open file.
- Must refer to a disk file.
- **Example** The following example opens an existing disk file named Test.fil and then uses LOF to determine its size in bytes:

Open "Test.fil" For Input As #1 SizeOfFile = LOF(1)

```
Returns Integer
```

**Tip** To determine the length of a file that is not open, use FileLen.

**Example** The following example creates a test file on disk that contains some random numbers. The procedure then reopens the test file and reports its length. This example overrides Start() which calls a sub procedure to generate the test file. The test file is deleted at the end of the Start function. To use this example, paste the procedure MakeDataFile0314 after the End Function of the Start() procedure or paste it into the Actuate Basic source code (.bas) file.

```
Sub Start()
  Dim Msg As String
  Dim FileLength As Integer
  Super::Start( )
  ' Create sample data file
  MakeDataFile0327
  ' Open newly created file
  Open "Test.fil" For Input As #1
  ' Get length of file
  FileLength = LOF(1)
  ' Close test file
  Close #1
  Msg = "The length of the Test.fil file just created is "
     & FileLength & " bytes. "
+
     & "The sample data file will now be deleted."
+
  ShowFactoryStatus ( Msg )
   ' Delete test file
```

```
Kill "Test.fil"
End Sub
Sub MakeDataFile0327()
Dim I As Integer
 ' Open file for output
Open "Test.fil" For Output As #1
 ' Generate values 0-250
For I = 0 To 250
Print #1, I
Next I
 ' Close test file
Close #1
End Sub
```

See also EOF function FileLen function Loc function Open statement

## Log function

Gives the natural logarithm for a number.

Syntax	Log( <number>)</number>
Parameters	<pre><number> Numeric expression or Variant of VarType 8 (String) for which you want to find the logarithm. <number> must be greater than 0.</number></number></pre>
Returns	Double
	If <number> evaluates to Null, Log returns Null.</number>
Tips	<ul> <li>To calculate logarithms in a base other than e, use Log(<number> divided by Log(<base/>), where <base/> is the number of the desired base other than e.</number></li> </ul>
	<ul> <li>Log is the inverse of Exp.</li> </ul>
Example	The following example generates a number, then returns the natural logarithm for that number:
	Sub Start( ) Super::Start( ) On Error Goto ErrorHandler

```
Dim UserNum, Msg As String
UserNum = 1000 * Rnd
Msg = "The log (base e), or Log(" & UserNum & ") is "
+ & Log(UserNum )
ShowFactoryStatus( Msg )
Exit Sub
ErrorHandler:
ShowFactoryStatus( "Error: " & Err & " -- " & Error$(Err) )
ShowFactoryStatus( "Please try again." )
Resume Next
End Sub
```

See also Exp function

## LSet statement

Left-aligns a string within the space of a string variable.

Syntax LSet <string variable> = <string exprs>

**Description** LSet left-aligns a string expression within a variable.

Parameters <string variable>

Name of a string variable in which LSet stores the left-aligned <string exprs>.

#### <string exprs>

String expression that you want LSet to left-align within <string variable>.

The behavior of LSet depends on whether <string exprs> is shorter or longer than <string variable>, as shown in Table 6-32.

Table 6-32 LSet behavior

<string exprs=""></string>	Behavior of LSet
Shorter than <string variable&gt;</string 	Left-aligns <string exprs=""> within <string variable="">. Replaces any leftover characters in <string variable=""> with spaces.</string></string></string>
Longer than <string variable&gt;</string 	Places only the first characters, up to the length of the <string variable="">, in <string variable="">. Truncates characters beyond the length of <string variable=""> from the right.</string></string></string>

**Example** The following example left-aligns text within a 20-character string variable:

```
Sub Start()
  Dim Msg, TmpStr As String
  Super::Start( )
  ' Create 20-character string
  TmpStr = String(20, "*")
  Msg = "The following two strings that have been right"
  & "and left justified in a " & Len( TmpStr )
+
     & "-character string."
  ShowFactoryStatus ( TmpStr )
  ' Right justify
  RSet TmpStr = "Right->"
  ShowFactoryStatus ( TmpStr )
  ' Left justify
  LSet TmpStr = "<-Left"
  ShowFactoryStatus ( TmpStr )
End Sub
```

See also RSet statement

## LTrim, LTrim\$ functions

Returns a copy of a string expression after removing leading spaces.

LTrim(<string exprs>) Syntax LTrim\$(<string exprs>) **Parameters** <string exprs> String from which LTrim[\$] strips leading spaces. Leading spaces are any spaces that occur before the first non-space character in a string. <string exprs> must be a variable string, a literal string, a string constant, the return value of any function that returns a string, or a Variant that can evaluate to a String. Returns LTrim: Variant LTrim\$: String A copy of <string exprs> with leading spaces removed. If there are no leading spaces, LTrim[\$] returns <string exprs>. If <string exprs> evaluates to Null, LTrim[\$] returns Null. **Tips** • To simultaneously strip both leading and trailing spaces in a string, use Trim\$. To find spaces in the middle of a string, use InStr. Example The following example uses LTrim\$ to strip leading spaces from a string variable and uses RTrim\$ to strip trailing spaces:

```
Sub Start()
  Dim Msg As String
  Dim CustName As String, CustName1 As String
  Super::Start( )
  CustName = " Harold Painter "
  ' Strip spaces
  CustName1 = LTrim$(RTrim$(CustName))
  Msg = "The original customer name " & "'"
     & CustName & "'" & " ...was " & Len(CustName)
+
     & " characters long. There were two leading "
+
     & "spaces and two trailing spaces."
  ShowFactoryStatus ( Msg )
  Msg = "The name returned after stripping the spaces "
     & "is:" & "'" & CustName1
+
     & "'" & "...and it contains only "
+
     & Len(CustName1) & " characters."
  ShowFactoryStatus ( Msg )
End Sub
```

```
See also Format, Format$ functions
RTrim, RTrim$ functions
```

### Mid, Mid\$ functions

Returns the specified portion of a string expression.

Syntax Mid(<string exprs>, <start>[,<length>])

Mid\$(<string exprs>, <start>[,<length>])

#### Parameters <string exprs>

Source string from which you are copying a portion. It can be a variable string, a literal string, a string constant, the return value of any function that returns a string, or a Variant that can evaluate to a String.

#### <start>

Integer that specifies the character position within a <string exprs> where copy begins. <start> must be between 1 and 2,147,483,647, inclusive.

#### <length>

Integer indicating how many characters to copy.

The following conditions apply to <length>:

 If <length> is omitted, Actuate Basic uses all characters from <start> to the end of <string exprs>. ■ Must be between 0 and 2,147,483,647.

The following examples are equivalent:

Mid\$("Widget",4,3) Mid\$("Widget",4)

The following example returns Widget:

```
Mid$("Widget",1,99)
```

## Returns Mid: Variant Mid\$: String

- If <start> is greater than length of <string exprs>, returns zero-length string.
- If <length> = 0, returns zero-length string.
- If <length> is greater than length of <string exprs>, returns only the characters up to the length of <string exprs>.
- If any parameter evaluates to Null, Mid[\$] returns Null.
- **Tip** Use Len to find the number of characters in <string exprs>.
- **Example** In the following example, Mid[\$] returns the middle word from a variable containing three words:

```
Sub Start()
  Dim MiddleWord, Msg, TestStr
  Dim SpacePos1, SpacePos2, WordLen
  Super::Start( )
  ' Create text string
  TestStr = "Mid function Example"
  ' Find first space
  SpacePos1 = InStr(1, TestStr, " ")
  ' Find next space
  SpacePos2 = InStr(SpacePos1 + 1, TestStr, " ")
  ' Calc 2nd word length
  WordLen = (SpacePos2 - SpacePos1) - 1
  ' Find the middle word
  MiddleWord = Mid(TestStr, SpacePos1 + 1, WordLen)
  Msg = "The word in the middle of """ & TestStr & """ is """
   & MiddleWord & """"
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Left, Left\$ functions Len function Right, Right\$ functions

### Mid, Mid\$ statements

Replaces a segment of a string with another string.

**Syntax** Mid(<string variable>,<start>[,<length>]) = <string exprs>

Mid\$(<string variable>,<start>[,<length>]) = <string exprs>

#### Parameters <string variable>

String expression or Variant variable to modify. <string variable> cannot be a literal, function, or other expression.

#### <start>

Position in <string variable> at which replacement text begins. The position of the first character is 1. <start> must be between 1 and 2,147,483,647, inclusive.

#### <length>

Number of characters from <string exprs> to use. The default is the unchanged <string exprs>.

The following conditions apply to <length>:

- Must be between 1 and 2,147,483,647, inclusive.
- No matter how large you specify <length> to be, the resulting string is never longer than <string variable>.

#### <string exprs>

String expression that replaces segment of <string variable>. It can be a variablelength string, a literal string, a string constant, the return value of any function that returns a string, or a Variant that can evaluate to a String. <string exprs> cannot entirely replace <string variable> with a shorter <string exprs>.

**Example** The following example replaces part of the string FREUDIAN with all or part of Jung in the variable Hillman:

```
Sub Start( )
Dim Hillman As String
Super::Start( )
Hillman = "FREUDIAN"
' now Hillman = "JungDIAN"
Mid$(Hillman,1,5) = "Jung"
ShowFactoryStatus( Hillman )
Hillman = "FREUDIAN"
' same result, "JungDIAN"
Mid(Hillman,1,4) = "Jung"
ShowFactoryStatus( Hillman )
Hillman = "FREUDIAN"
' now Hillman = "JunUDIAN"
```

```
Mid$(Hillman,1,3) = "Jung"
ShowFactoryStatus(Hillman)
Hillman = "FREUDIAN"
' now Hillman = "FREUJung"
Mid(Hillman,5) = "Jung"
ShowFactoryStatus(Hillman)
End Sub
```

#### See also Left, Left\$ functions Mid, Mid\$ functions Right, Right\$ functions

## MidB, MidB\$ functions

Returns the specified portion of a string expression.

These functions are provided for backward compatibility with previous Actuate releases. Actuate uses UCS-2 as internal encoding, therefore passing strings consisting of characters from different code pages might produce unexpected results. For this reason, Actuate recommends you use the Mid and Mid\$ functions instead of the MidB and MidB\$ functions.

Syntax MidB(<string exprs>, <start>[,<length>])

MidB\$(<string exprs>, <start>[,<length>])

#### Parameters <string exprs>

String expression from which you are copying a byte or a series of contiguous bytes. Can be a variable string, a literal string, a string constant, the return value of any function that returns a string, or a Variant that can evaluate to a String.

#### <start>

Integer that specifies the byte position within a <string exprs> where copy begins. <start> must be between 1 and 2,147,483,647, inclusive.

#### <length>

Integer indicating how many bytes to copy.

The following conditions apply to <length>:

- If <length> is omitted, Actuate Basic uses all characters from <start> to the end of <string exprs>.
- <length> must be between 0 and 2,147,483,647.

The following example statements are equivalent:

```
MidB$("Widget",4,3)
MidB$("Widget",4)
```

The following statement returns "Widget":

```
MidB$("Widget",1,99)
```

Returns MidB: Variant MidB\$: String

- If <start> is greater than length of <string exprs>, the function returns zerolength string.
- If <length> = 0, the function returns zero-length string.
- If <length> is greater than length of <string exprs>, the function returns only the characters up to the length of <string exprs>.
- If any parameter evaluates to Null, MidB[\$] returns Null.
- **Tips** Use LenB[\$] to find the number of bytes in <string exprs>.
  - Use InStrB[\$] to find the position of a specified byte in <string exprs>.

**Example** In the following example, MidB returns the middle word from a variable containing three words:

```
Sub Start()
  Dim MiddleWord, Msg, TestStr
  Dim SpacePos1, SpacePos2, WordLen
  Super::Start( )
  ' Create text string
  TestStr = "MidB function Example"
  ' Find first space
  SpacePos1 = InStr(1, TestStr, " ")
  ' Find next space
  SpacePos2 = InStr(SpacePos1 + 1, TestStr, " ")
  ' Calc 2nd word length
  WordLen = (SpacePos2 - SpacePos1) - 1
  ' Find the middle word
  MiddleWord = MidB(TestStr, SpacePos1 + 1, WordLen)
  Msg = "The word in the middle of """ & TestStr & """ is """
     & MiddleWord & """"
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also InStrB function LeftB, LeftB\$ functions LenB function Mid, Mid\$ functions RightB, RightB\$ functions

## MidB, MidB\$ statements

Replaces a segment of a string with another string.

These statements are provided for backward compatibility with previous Actuate releases. Actuate uses UCS-2 as internal encoding, therefore passing strings consisting of characters from different code pages might produce unexpected results. For this reason, Actuate recommends you use the Mid and Mid\$ statements instead of the MidB and MidB\$ statements.

**Syntax** MidB(<string variable>,<start>[,<length>]) = <string exprs>

MidB\$(<string variable>,<start>[,<length>]) = <string exprs>

#### Parameters <string variable>

String expression or Variant to modify. <string variable> must be a variable, not a literal, function, or other expression.

#### <start>

Position in <string variable> at which replacement text begins. The position of the first byte is 1. <start> must be between 1 and 2,147,483,647, inclusive.

#### <length>

Number of bytes from <string exprs> to use. The default is the unchanged <string exprs>.

The following conditions apply to <length>:

- Must be between 1 and 2,147,483,647, inclusive.
- No matter how large you specify <length> to be, the resulting string is never longer than <string variable>.

#### <string exprs>

String expression that replaces segment of <string variable>. It can be a variable string, a literal string, a string constant, the return value of any function that returns a string, or a Variant that can evaluate to a String. <string exprs> cannot entirely replace <string variable> with a shorter <string exprs>.

**Example** The following example replaces part of the string FREUDIAN with all or part of Jung in the variable Hillman:

```
Sub Start( )
  Dim Hillman As String
  Super::Start( )
  Hillman = "FREUDIAN"
```

```
' now Hillman = "JungDIAN"
  MidB(Hillman,1,5) = "Jung"
  ShowFactoryStatus ( Hillman )
  Hillman = "FREUDIAN"
  ' same result, "JungDIAN"
  MidB(Hillman,1,4) = "Jung"
  ShowFactoryStatus ( Hillman )
  Hillman = "FREUDIAN"
  ' now Hillman = "JunUDIAN"
  MidB$(Hillman,1,3) = "Jung"
  ShowFactoryStatus ( Hillman )
  Hillman = "FREUDIAN"
  ' now Hillman = "FREUJung"
  MidB$(Hillman,5) = "Jung"
  ShowFactoryStatus ( Hillman )
End Sub
```

See also LeftB, LeftB\$ functions MidB, MidB\$ functions RightB, RightB\$ functions

### **Minute function**

Returns an integer from 0 to 59, inclusive, that represents the minute of the hour specified by a date expression.

Syntax Minute(<date exprs>)

#### Parameters <date exprs>

Date expression, or any numeric or string expression that can be interpreted as a date, a time, or both a date and a time:

- Can be a string such as November 12, 1982 8:30 PM, Nov. 12, 1982 08:30 PM, 11/12/82 8:30pm, 08:30pm, or any other string that can be interpreted as a date, a time, or both a date and a time in the valid range.
- Can be a date serial number such as 30267.854, which corresponds to November 12, 1982 at 8:30 P.M., or any other number that can be mapped to a date, a time, or both a date and a time in the valid range.
- For date serial numbers, the integer component represents the date itself while the fractional component represents the time of day on that date, where January 1, 1900 at precisely noon has the date serial number 2.5, and negative numbers represent dates prior to December 30, 1899 (0).

The default is 0.

The following conditions apply to <date exprs>:

- If <date exprs> includes a date, it must be a valid date, even though Minute does not return a date. A valid date is any date in the range January 1, 100 through December 31, 9999, expressed in one of the standard date formats.
- If <date exprs> includes a time, it must be in the range 00:00:00 (12:00:00 A.M.) through 23:59:59 (11:59:59 P.M.), in either the 12- or 24-hour format.
- If <date exprs> is a numeric expression, it must be in the range -657434.0 to +2958465.9999, inclusive.
- If <date exprs> is a variable containing a date serial number, the variable must be explicitly declared as one of the numeric types.
- <date exprs> is parsed according to the formatting rules of the current run-time locale.

For example, the following statements are equivalent. Each assigns 35 to the variable UserMinute.

```
UserMinute = Minute("6/7/64 2:35pm")
UserMinute = Minute("5:35 pm")
UserMinute = Minute("June 7, 1964 2:35 PM")
UserMinute = Minute("Jun 7, 1964") + 35
UserMinute = Minute(23535.6077)
UserMinute = Minute(0.6077)
```

#### Returns Integer

 If <date exprs> cannot be evaluated to a date, Minute returns Null. For example:

Minute("This is not a date.") returns Null

 If <date exprs> fails to include all date components (day, month, and year), Minute returns Null. For example:

Minute("Nov 12, 1982 7:11 AM")

returns 11, but:

Minute("Nov 1982 7:11 AM")

returns Null.

If <date exprs> is Null, Minute returns Null.

The following assumes that the AC\_CENTURY\_BREAK property is set to the default value of 30. For information about using AC\_CENTURY\_BREAK, please see *Accessing Data using e.Report Designer Professional*.

 For abridged date expressions ending in 00 through 29 that do not specify the century, relevant date functions estimate the unabridged year by adding 2000. For example, while Year (#4/30/1910#) returns 1910, the abridged expression Year (#4/30/10#) returns 2010 (10 + 2000).

For abridged date expressions ending in 30 through 99 that do not specify the century, relevant date functions estimate the unabridged year by adding 1900. For example, while Year(#11/12/2082#) returns 2082, the abridged expression Year(#11/12/82#) returns 1982 (82 + 1900).

Note that a date can contain an abridged month, such as Nov. 12, 1982 as opposed to November 12, 1982.

- **Tip** If you use a date expression in one locale, it might be misinterpreted in another locale. For instance, in the United States, 1/2/2005 means January 2nd, 2005, but in France, it means February 1st, 2005. To avoid such ambiguities, use DateSerial to specify all your dates.
- **Example** The following example displays the number of hours, minutes, and seconds remaining until midnight:

```
Sub Start()
  Dim HrDiff As Integer, MinDiff As Integer, SecDiff As Integer
  Dim RightNow As Double, Midnight As Double
  Dim TotalDiff As Double, TotalMinDiff As Double
  Dim TotalSecDiff As Double, Msg As String
  Super::Start( )
  Midnight = TimeValue( "23:59:59" )
  ' Get current time
  RightNow = Now
  ' Get diffs from midnight
  HrDiff = Hour( Midnight ) - Hour( RightNow )
  MinDiff = Minute( Midnight ) - Minute( RightNow )
  SecDiff = Second( Midnight ) - Second( RightNow ) + 1
  ' Restate seconds and minutes if necessary
  If SecDiff = 60 Then
     ' Add 1 to minute
     MinDiff = MinDiff + 1
     ' And set 0 seconds
     SecDiff = 0
  End If
  If MinDiff = 60 Then
     ' Add 1 to hour
     HrDiff = HrDiff + 1
     ' And set 0 minutes
     MinDiff = 0
  End If
  ' Now get totals
  TotalMinDiff = ( HrDiff * 60 ) + MinDiff
  TotalSecDiff = ( TotalMinDiff * 60 ) + SecDiff
  TotalDiff = TimeSerial( HrDiff, MinDiff, SecDiff )
```
```
' Prepare msg for display
Msg = "There are a total of " & Format(TotalSecDiff, "#,##0")
+ & " seconds until midnight. That translates to "
+ & HrDiff & " hours, "
+ & MinDiff & " minutes, and "
+ & SecDiff & " seconds. "
+ & "In standard time notation, it becomes "
' Remember not to use "mm" for minutes! m is for month.
Msg = Msg & Format( TotalDiff, "hh:nn:ss" ) & "."
ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Day function Hour function

Hour function Month function Now function Second function Time, Time\$ functions Weekday function Year function

# **MIRR** function

Returns the modified internal rate of return for a series of periodic cash flows (payments and receipts) in an existing array.

**Syntax** MIRR(<casharray>(),<finance rate>, <reinvestment rate>)

#### Parameters <casharray>()

Array of Doubles that specifies the name of an existing array of cash flow values. The <casharray> array must contain at least one positive value (receipt) and one negative value (payment).

#### <finance rate>

Numeric expression that specifies the interest rate paid as the cost of financing. <finance rate> must be a decimal value that represents a percentage.

#### <reinvestment rate>

Numeric expression that specifies the interest rate received on gains from cash reinvestment. <reinvestment rate> must be a decimal value that represents a percentage.

The following example assumes you have filled the array MyArray() with a series of cash flow values. If the interest rate you pay for financing is 12%, and the rate

your earn on income is 11.5%, what is the modified internal rate of return? The answer is assigned to the variable MIRRValue.

```
MIRRValue = MIRR(MyArray(), 0.12, 0.115)
```

#### Returns Double

The modified internal rate of return is the internal rate of return (IRR) when payments and receipts are financed at different rates. MIRR takes into account both the cost of the investment (<finance rate>) and the interest rate received on the reinvestment of cash (<reinvestment rate>).

- **Rules** You must express cash paid out (such as deposits to savings) using negative numbers, and cash received (such as dividend checks) using positive numbers.
  - <casharray>() must contain at least one negative and one positive number.
  - In cases where you have both a positive cash flow (income) and a negative one (payment) for the same period, use the net flow for that period.
  - If no cash flow or net cash flow occurs for a particular period, you must type 0 (zero) as the value for that period.
  - **Tip** Because MIRR uses the order of values within the array to interpret the order of payments and receipts, be sure to type payment and receipt values in the correct sequence.
- **Example** The following example returns the modified internal rate of return for a series of cash flows contained in the array, CashValues(). LoanAPR represents the financing interest, and InvAPR represents the interest rate received on reinvested cash. Option Base is assumed to be set to zero.

```
Sub Start()
  ' Set up array
  Static CashValues(5) As Double
  Dim LoanAPR As Double, InvAPR As Double
  Dim Fmt As String, ReturnRate As Double, Msg As String
  Super::Start( )
  ' Loan rate
  LoanAPR = .1
  ' Reinvestment rate
  InvAPR = .12
  ' Define money format
  Fmt = "#0.00\%"
  ' Business start-up costs
  CashValues(0) = -80000
  ' Now set up positive cash flows for income for four years:
  CashValues(1) = 23000: CashValues(2) = 27000
  CashValues(3) = 31000: CashValues(4) = 35000
   ' Calculate internal rate
```

```
ReturnRate = MIRR(CashValues, LoanAPR, InvAPR)
Msg = "The modified internal rate of return for these cash "
+ & "flows is: " & Format(Abs(ReturnRate), Fmt) & "."
ShowFactoryStatus( Msg )
ShowFactoryStatus( CStr( CashValues(1) ) )
ShowFactoryStatus( CStr( CashValues(2) ) )
ShowFactoryStatus( CStr( CashValues(3) ) )
ShowFactoryStatus( CStr( CashValues(4) ) )
End Sub
For information about using the code examples, see "Using the code examples,"
earlier in this chapter.
```

See also IRR function Rate function

### **MkDir statement**

Creates a new directory or subdirectory on the specified or current drive.

Syntax MkDir <path name>

**Description** If you do not specify the full path name, MkDir creates the new subdirectory on the current drive under the current directory.

#### Parameters <path name>

String expression that is the name of the directory to be created.

<path name> has the following syntax:

[<drive:>] [ \ ]<directory>[\<directory>]...(Windows)

[/]<directory>[/<directory>]...(UNIX)

The following conditions apply to <path name>:

- Must contain fewer than 259 characters.
- Cannot be the name of a directory that already exists.

#### <drive:>

Character, followed by a colon, that is the name of the drive (Windows only).

#### <directory>

String expression that is the name of the directory or subdirectory to create.

The following example creates the subdirectory Docs under the current directory on the default drive:

MkDir "Docs"

The following example creates the subdirectory Docs under the root directory of the current drive:

DirName\$ = "\Docs" MkDir DirName\$

On a Windows system, the following example creates the subdirectory Docs under the root directory of drive D:

MkDir "D:\Docs"

- **Tips** MkDir is similar to the operating system MkDir. Unlike the DOS command, however, it cannot be abbreviated.
  - If you use one or more embedded spaces in <path name> when you create a directory, use Actuate Basic's RmDir to remove the directory.
  - To determine the current directory, use CurDir.
- **Example** The following example determines whether a \Tmp subdirectory exists on the current drive. If it does not, the example creates the directory.

```
Sub Start()
  Dim UserAns As Integer, ThisDrive As String
  Dim Msq As String, TempDir As String
  Super::Start( )
  ' Set up error handler
  On Error Resume Next
  ' Get current drive letter
  ThisDrive = Left$(CurDir, 2)
  ' Construct full path spec
  TempDir = UCase$(ThisDrive & "\Tmperase")
  ' Make the new directory
  MkDir TempDir
  ' Does it exist?
  If Err = 41 Then
     Msg = "Sorry, " & TempDir & " directory already exists."
  Else
     Msg = TempDir & " directory was created.
  End If
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also CurDir, CurDir\$ functions ChDir statement RmDir statement

### **Month function**

Returns an integer between 1 and 12, inclusive, that represents the month of the year for a specified date argument.

Syntax Month(<date exprs>)

#### Parameters <date exprs>

Date expression, or any numeric or string expression that can evaluate to a date or a date and a time:

- Can be a string such as November 12, 1982, Nov 12, 1982, 11/12/82, 11-12-82, or any other string that can be interpreted as a date in the valid range.
- Can be a date serial number such as 30267.854, which corresponds to November 12, 1982 at 8:30 P.M., or any other number that can be mapped to a date in the valid range.
- For date serial numbers, the integer component represents the date itself while the fractional component represents the time of day on that date, where January 1, 1900 at precisely noon has the date serial number 2.5, and negative numbers represent dates prior to December 30, 1899 (0).

The following conditions apply to <date exprs>:

- If <date exprs> a string expression, must specify a date in the range January 1, 100 through December 31, 9999, inclusive.
- If <date exprs> is a numeric expression, must be in the range -657434 to +2958465, inclusive.
- If <date exprs> is a variable containing a date serial number, the variable must be explicitly declared as one of the numeric types.
- <date exprs> is parsed according to the formatting rules of the current run-time locale.

For example, the following statements are equivalent. Each assigns 6 to the variable UserMonth.

```
UserMonth = Month("6/7/64")
UserMonth = Month("June 7, 1964 2:35 PM")
UserMonth = Month("Jun 7, 1964")
UserMonth = Month(23535)
UserMonth = Month(4707*5)
```

#### Returns Integer

• If <date exprs> cannot be evaluated to a date, Month returns Null. For example, the following statement returns Null:

```
Month("This is not a date.")
```

 If <date exprs> fails to include all date components (day, month, and year), Month returns Null. For example:

```
Month("Nov 12, 1982")
returns 11, but:
Month("Nov 1982")
returns Null.
```

■ If <date exprs> is Null, Month returns Null.

The following assumes that the AC\_CENTURY\_BREAK property is set to the default value of 30. For information about using AC\_CENTURY\_BREAK, please see *Accessing Data using e.Report Designer Professional*.

- For abridged date expressions ending in 00 through 29 that do not specify the century, relevant date functions estimate the unabridged year by adding 2000. For example, while Year(#4/30/1910#) returns 1910, the abridged expression Year(#4/30/10#) returns 2010 (10 + 2000).
- For abridged date expressions ending in 30 through 99 that do not specify the century, relevant date functions estimate the unabridged year by adding 1900. For example, while Year(#11/12/2082#) returns 2082, the abridged expression Year(#11/12/82#) returns 1982 (82 + 1900).

Note that a date can contain an abridged month, such as Nov. 12, 1982 as opposed to November 12, 1982.

- **Tip** If you use a date expression in one locale, it might be misinterpreted in another locale. For instance, in the United States, 1/2/2005 means January 2nd, 2005, but in France, it means February 1st, 2005. To avoid such ambiguities, use DateSerial to specify all your dates.
- **Example** The following example analyzes today's date. It uses various date functions to display the year, month, day, and weekday of the given date. Finally, it gives the date's serial number.

```
Sub Start( )
Dim UserEntry, UserYear As Integer, UserMonth As Integer
Dim UserDay As Integer, UserDOW As Integer, DOWName As String
Dim Msg As String, LeapYear As String
Super::Start( )
' Get date
UserEntry = Now
UserEntry = CDate( UserEntry )
' Calculate year
UserYear = Year( UserEntry )
' Calculate month
UserMonth = Month( UserEntry )
' Calculate day
UserDay = Day( UserEntry )
```

```
' Calculate day of week
  UserDOW = Weekday (UserEntry)
  ' Convert to name of day
  DOWName = Format$ (UserEntry, "dddd")
  ' Determine if the year is a leap year or a centesimal
  If UserYear Mod 4 = 0 And UserYear Mod 100 = 0 Then
  ' Evenly divisible by 400?
     If UserYear Mod 400 = 0 Then
       LeapYear = "is a leap year."
     ' Not evenly divisible
     Else
       LeapYear = "is a centesimal but not a leap year."
     End If
  ElseIf UserYear Mod 4 = 0 Then
       LeapYear = "is a leap year."
  Else
        LeapYear = "is not a leap year."
  End If
  ' Display results for the user after the pattern:
  ' The given year 1982 is not a leap year. The weekday number
  ' for the 12th day of the 11th month in 1982 is 3, which
  ' means that day was a Tuesday.
  ' The date serial number for 11/12/82 is 30267.
  Msg = "The given year, " & UserYear & ", " & LeapYear
  ShowFactoryStatus ( Msg )
  Msg = "The weekday number for the " & UserDay
   & Suffix(UserDay) & " day of the " & UserMonth
+
     & Suffix(UserMonth) & " month in " & UserYear
+
   & " is " & UserDOW & ", which means that day is a "
     & DOWName & "." & "The date serial "
   & "number for " & UserEntry & " is: "
     & CDbl(DateSerial(UserYear, UserMonth, UserDay))
  ShowFactoryStatus ( Msg )
End Sub
' The previous procedure calls this function to tell us what
' suffix (1st, 2nd, 3rd) to use with a number.
Function Suffix (DateNum) As String
  Select Case DateNum
     Case 1, 21, 31
        Suffix = "st"
     Case 2, 22
       Suffix = "nd"
```

```
Case 3, 23
Suffix = "rd"
Case Else
Suffix = "th"
End Select
End Function
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Date, Date\$ functions Day function Hour function Minute function Now function Second function Weekday function Year function

# **MsgBox function**

Displays a user-defined message in a window with a specified set of response buttons, waits for the user to choose a button, and returns a value indicating which button the user selected.

Syntax MsgBox(<message text> [, [<configuration>][, <window title>] ])

#### Parameters <message text>

String expression that is the message to be displayed in the dialog box. Lines break automatically at the right edge of the dialog box. If the <message text> for an application modal message box is longer than 1024 characters, the message is truncated.

#### <configuration>

Numeric expression that is the bitwise sum of up to four separate values:

- Number and type of buttons
- Icon style
- Default button
- Modality

Table 6-33 summarizes values for <configuration> and their corresponding meanings.

Group	Symbolic constant	Value	Meaning
Button group	MB_OK	0	Display OK button.
	MB_OKCANCEL	1	Display OK and Cancel buttons.
	MB_ABORTRETRYIGNORE	2	Display Abort, Retry, and Ignore buttons.
	MB_YESNOCANCEL	3	Display Yes, No, and Cancel buttons.
	MB_YESNO	4	Display Yes and No buttons.
	MB_RETRYCANCEL	5	Display Retry and Cancel buttons.
Icon style group	MB_ICONSTOP	16	Display Stop icon.
	MB_ICONQUESTION	32	Display Question Mark icon.
	MB_ICONEXCLAMATION	48	Display Exclamation Point icon.
	MB_ICONINFORMATION	64	Display Information icon.
Default button group	MB_DEFBUTTON1	0	First button is default.
	MB_DEFBUTTON2	256	Second button is default.
	MB_DEFBUTTON3	516	Third button is default.
Modality group	MB_APPLMODAL	0	Application modal. User must respond to message box before continuing in current application.
	MB_SYSTEMMODAL	4096	System modal. User must respond to message box before continuing with any application.

Table 6-33<configuration> values for the MsgBox function

Add these numbers or constants together to specify the corresponding attributes of the message box. The default is 0 and you can use only one number from each group.

For example, each of the following statements displays a message box with the Question Mark icon, the message "Are you sure you want to overwrite the file?", and the Yes, No, and Cancel buttons. In each case, the No button is the default. The user's response is assigned to the variable UserAns.

```
UserAns = MsgBox("Do you want to overwrite the file? ", 3+32+256)
UserAns = MsgBox("Do you want to overwrite the file? ", 291)
UserAns = MsgBox("Do you want to overwrite the file? ",
    MB YESNOCANCEL + MB ICONQUESTION + MB DEFBUTTON2)
```

#### <window title>

String expression that is the text displayed in the title bar of the dialog box. The default is no title in the title bar.

#### Returns Integer

The value that MsgBox function returns tells you which button the user selected.

Table 6-34 summarizes possible user selections and their corresponding return values.

Selection at run time	MsgBox returns	Symbolic Constant is
OK button	1	IDOK
Cancel button	2	IDCANCEL
Abort button	3	IDABORT
Retry button	4	IDRETRY
Ignore button	5	IDIGNORE
Yes button	6	IDYES
No button	7	IDNO

Symbolic constants and MsgBox return values for user selections at run time

- **Tip** To set line breaks yourself, place a linefeed (ANSI character 10) before the first character of the text that is to begin each new line.
- **Example** The following example displays a different message box based on whether the user selected the Yes or No button in a dialog box:

```
Sub Start( )
Dim Title As String, MsgTxt As String
Dim DialogDef As Integer, UserResponse As Integer
```

```
Super::Start()
Title = "Demonstration of MsgBox Function"
MsgTxt = "Show the display of a critical-error message."
MsgTxt = MsgTxt & " Do you want to continue?"
DialogDef = MB_YESNO + MB_ICONSTOP + MB_DEFBUTTON2
UserResponse = MsgBox(MsgTxt, DialogDef, Title)
If UserResponse = IDYES Then
    MsgTxt = "You selected Yes."
Else
    MsgTxt = "You selected No or pressed Enter."
End If
MsgBox MsgTxt
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapterUsing the code examples.

See also InStr function MsgBox statement

### MsgBox statement

Displays a user-defined message in a window with a specified set of response buttons, and waits for the user to choose a button. MsgBox statement does not return a value indicating which button the user has clicked.

Syntax MsgBox <message text> [, [<configuration>][, <window title>] ]

#### Parameters <message text>

String expression that is the message displayed in the dialog box. Lines break automatically at the right edge of the dialog box. For application modal message boxes, <message text> must not be longer than 1024 characters, or it is truncated.

#### <configuration>

Numeric expression that is the bitwise sum of up to four separate values:

- Number and type of buttons
- Icon style
- Default button
- Modality

Table 6-35 summarizes values for <configuration> and their corresponding meanings.

Group	Symbolic constant	Value	Meaning
Button group	MB_OK	0	Display OK button.
	MB_OKCANCEL	1	Display OK and Cancel buttons.
	MB_ABORTRETRYIGNORE	2	Display Abort, Retry, and Ignore buttons.
	MB_YESNOCANCEL	3	Display Yes, No, and Cancel buttons.
	MB_YESNO	4	Display Yes and No buttons.
	MB_RETRYCANCEL	5	Display Retry and Cancel buttons.
Icon style group	MB_ICONSTOP	16	Display Stop icon.
	MB_ICONQUESTION	32	Display Question Mark icon.
	MB_ICONEXCLAMATION	48	Display Exclamation Point icon.
	MB_ICONINFORMATION	64	Display Information icon.
Default button group	MB_DEFBUTTON1	0	First button is default.
	MB_DEFBUTTON2	256	Second button is default.
	MB_DEFBUTTON3	516	Third button is default.
Modality group	MB_APPLMODAL	0	Application modal. User must respond to message box before continuing in current application.
	MB_SYSTEMMODAL	4096	System modal. User must respond to message box before continuing with any application.

 Table 6-35
 <configuration> values for MsgBox statements

Add these numbers or constants together to specify the corresponding attributes of the message box.

The default is 0 and you may use only one number from each group.

For example, each of the following displays a message box with the Exclamation Point icon, the message: Sorry, insufficient disk space, and the OK button. In each case, the No button is the default button.

MsgBox "Sorry, insufficient disk space", 0+48
MsgBox "Sorry, insufficient disk space", 48
MsgBox "Sorry, insufficient disk space",
+ (MB OK + MB ICONEXCLAMATION)

#### <window title>

String expression that specifies the words to appear in the title bar of the message window. The default is no title in title bar.

**Tip** To set line breaks yourself, place a linefeed (ANSI character 10) before the first character of the text that is to begin each new line. This does not apply to Microsoft Windows 3.0.

**Example** The following example displays three message boxes consecutively:

```
Sub Start( )
Super::Start( )
' Define the dialog box
MsgBox "The dialog box shows a Yes and a No button.",
+ MB_YESNO
MsgBox "The dialog box shows only an OK button."
MsgBox "The dialog box shows an OK button, a Cancel button, "
+ & "and an info icon.", MB_OKCANCEL + 64.
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapterUsing the code examples.

See also InStr function MsgBox function

### Name statement

Renames a file or directory. Also, moves a file from one directory to another.

- Syntax Name <old name> As <new name or path>
- **Description** Name is similar to the operating system's Rename command. Name, however, can change the name of a directory as well as that of a file.

#### Parameters As

Keyword that separates the name of the original file or directory from the name of the target file or directory.

#### <old name>

String expression that specifies an existing file or directory to rename. The default path is the current default drive and directory.

The following conditions apply to <old name>:

- Indicated file or directory must exist.
- Drive letter, if specified, must be the same as that specified in <new name or path>.
- Indicated file must not currently be open by Actuate Basic.

#### <new name or path>

String expression that specifies the new file or directory name. The default path is the current drive and directory.

The following conditions apply to <new name>:

- Indicated file must not already exist if its path is the same as the path in <old name>.
- Indicated path must already exist if it is not the same as the path in <old name>.
- Drive letter, if specified, must be the same as that specified in <old name>.
- Indicated file must not currently be open by Actuate Basic.

<old name> or <new name or path> can optionally specify a full path, in which case use the following syntax:

[<drive:>] [ \ ]<directory>[\<directory>][\<file name>] (Windows)

[ / ]<directory>[/<directory>][<file name>] (UNIX)

#### <drive:>

Character, followed by a colon, that specifies the drive on which the directory to be renamed or the file to be renamed or moved is located (Windows only). Both <old name> and <new name or path> must refer to the same drive.

#### <directory>

String expression that specifies the name of a directory or subdirectory in the full path specification of <old name> or of <new name or path>. You can use Name to change the name of a directory, but you cannot move directories.

The following example changes the name of the file Test1.fil in the current directory to Test2.fil:

Name "Test1.fil " As "Test2.fil "

The following example moves the file Test1.fil from the C:\Temp subdirectory to the root directory:

Name "C:\Temp\Test1.fil " As "C:\Test1.fil "

**Example** The following example moves a file from one directory to another and renames the file:

```
Sub Start()
  Dim FileName1 As String, FileName2 As String
  Dim Msg As String, TestDir As String
  Super::Start( )
  ' Define filenames
  FileName1 = "Nameaaaa.dat"
  FileName2 = "Namebbbb.dat"
  ' Test directory name
  TestDir = "\Test.dir"
  ' Create test file
  Open FileName1 For Output As #1
  ' Put sample data in file
  Print #1, "This is a test."
  Close #1
  ' Make test directory.
  MkDir TestDir
  ' Move and rename file
  Name FileName1 As TestDir & "\" & FileName2
  Msg = "A new file, " & FileName1 & " was created in "
     & CurDir$ & ". After creation, it was moved to "
+
     & TestDir & " and renamed to " & FileName2 & "."
+
     & "The test data "
     & "file and directory will now be deleted."
  ShowFactoryStatus ( Msg )
  ' Delete file from disk
  Kill TestDir & "\" & FileName2
  ' Delete test directory
  RmDir TestDir
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Kill statement

### **NewInstance function**

Dynamically creates an instance of a class given the name of that class as a String. Use NewInstance to choose at run time which class to instantiate.

Syntax NewInstance(<class name>)

#### Parameters <class name>

String expression that specifies the fully qualified name of the class to instantiate.

Rule: The class name must be valid or a run-time error is sent.

- **Returns** The reference to the new object instance.
- **Example** The following example shows two methods for instantiating an object:

```
Sub Start( )
Dim Obj1 As AcLabelControl
Dim Obj2 As AcLabelControl, Msg As String
Super::Start( )
' Standard instantiation
Set Obj1 = New AcLabelControl
' Dynamic instantiation
Set Obj2 = NewInstance ("AcLabelControl")
Obj2.BackgroundColor = Red
Msg = "The numeric value for the background color "
+ & "of Obj2 is: " & Obj2.BackgroundColor
ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also IsPersistent function NewPersistentInstance function Set statement

### NewPersistentInstance function

Dynamically creates a persistent instance of a class given the name of that class as a String. Use NewPersistentInstance to choose at run time which class to instantiate.

**Syntax** NewPersistentInstance(<class name>, <file number>)

#### Parameters <class name>

String expression that specifies the name of the class to instantiate. <class name> must be valid or a run-time error is sent.

#### <file number>

Numeric expression that specifies the number of the open report object instance (.roi) file.

The following conditions apply to <file number>:

If no file number is specified, the object is persistent in the current default ROI.

- If two or more ROIs are open, you must specify the file number.
- Returns String

The reference to the new object instance.

**Example** The following example dynamically creates a persistent instance of an AcLabelControl object:

```
Sub Start( )
   Dim Obj As AcLabelControl, Msg As String
   Super::Start( )
   Set Obj = NewPersistentInstance ( "AcLabelControl", 1 )
   Obj.BackgroundColor = Red
   Msg = "The numeric value for the background color "
+ & "of Obj is: " & Obj.BackgroundColor
   ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also IsPersistent function NewInstance function Set statement

### Now function

Returns a date that represents the current date and time according to the user's operating system.

#### Syntax Now

The following example assigns the current year to the variable ThisYear:

```
ThisYear = Year(Now)
```

The following example assigns the current date and time to the variable ThisInstant:

ThisInstant = Now

The following example assigns the full date serial number representing the present moment to the double-precision variable FreezeFrame:

FreezeFrame# = Now

#### Returns Date

The value that Now returns usually looks like a date and/or time but is stored internally as a Double-precision floating point number—a date serial number—that represents a date between midnight January 1, 1980, and December 31, 2036,

inclusive, and/or a time between 00:00:00 and 23:59:59, or 12:00:00 A.M. and 11:59:59 P.M., inclusive.

**Example** In the following example, Now returns the current date and Format displays it as a long date that includes the day and date:

```
Sub Start( )
  Dim Today As Double
  Dim Msg As String
  Super::Start( )
  Today = Now
  Msg = "Today is " & Format(Today, "dddd, mmmm dd, yyyy")
  ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Date, Date\$ functions Day function Hour function Minute function Month function Second function Time, Time\$ functions Weekday function Year function

### **NPer function**

Returns the number of periods for an annuity based on periodic, constant payments, and on an unvarying interest rate.

Syntax NPer(<rate per period>, <each pmt>, <present value>, <future value>, <when due>)

#### Parameters <rate per period>

Numeric expression that specifies the interest rate that accrues per period. <rate per period> must be given in the same units of measure as <each pmt>. For instance, if <each pmt> is expressed as a monthly payment, then <rate per period> must be expressed as the monthly interest rate.

#### <each pmt>

Numeric expression that specifies the amount of each payment. <each pmt> must be given in the same units of measure as <rate per period>. For instance, if <rate per period> is expressed in months, then <each pmt> must be expressed as a monthly payment.

#### <present value>

Numeric expression that specifies the value today of a future payment or of a stream of payments.

For example, if you put \$23.94 in the bank today and leave it there for 15 years at an interest rate of 10% compounded annually, you will end up with about \$100. In this case, the present value of \$100 is approximately \$23.94.

#### <future value>

Numeric expression that specifies the cash balance you want after you have made your final payment.

- **Examples** You set up a savings plan with a goal of having \$75,000 in 18 years to pay for your child's education. For this plan, the future value is \$75,000.
  - You take out a loan for \$11,000. The future value is \$0.00, as it is for any typical loan.

#### <when due>

Numeric expression that specifies whether each payment is made at the beginning (1), or at the end (0) of each period. The default is 0 and <when due> must be 0 or 1.

The following example assumes you are making monthly payments at the first of each month on a loan of \$20,000, at an APR of 11.5%. If each payment is \$653.26, how many payments will you have to make to finish paying off the loan? The answer (36) is assigned to the variable NumPeriods.

NumPeriods = NPer(.115/12, -653.26, 20000, 0, 1)

#### Returns Double

An annuity is a series of cash payments, constant in value, made over a period of time. An annuity can be an investment, such as a monthly savings plan, or a loan, such as a home mortgage.

Rules:

- <rate per period> and <each pmt> must be expressed in terms of the same units (weekly/monthly/yearly, and so on).
- You must express cash paid out (such as deposits to savings) using negative numbers, and cash received (such as dividend checks) using positive numbers.
- **Example** The following example prompts the user for information about a loan. The example then returns the number of payments the user must make to pay off the loan.

```
Declare
Global Const ENDPERIOD = 0
' When payments are made
Global Const BEGINPERIOD = 1
```

```
End Declare
Sub Start()
  Dim FutureVal As Double, PresVal As Double, APR As Double
  Dim Payment As Double, PayWhen As Integer
  Dim TotalPmts As Integer, Msg As String
  Super::Start( )
  ' Usually 0 for a loan
  FutureVal = 0
  ' Amount to borrow
  PresVal = 200000
  ' Interest rate
  APR = 0.0625
  ' Amount to pay each month
  Payment = 1500
  ' Assume pay at end of month
  PayWhen = ENDPERIOD
  ' Do computation
  TotalPmts =
     NPer(APR / 12, -Payment, PresVal, FutureVal, PayWhen)
+
  If Int(TotalPmts) <> TotalPmts Then
     TotalPmts = Int(TotalPmts) + 1
  End If
  Msg = "It will take " & TotalPmts
     & " months to pay off your loan of " & PresVal
+
     & " by paying " & Payment & " each month."
+
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also FV function IPmt function Pmt function PPmt function PV function Rate function

# **NPV** function

Returns the net present value of a varying series of periodic cash flows, both positive and negative, at a given interest rate.

```
Syntax NPV(<rate>,<casharray>())
```

#### Parameters <rate>

Numeric expression that specifies the discount rate over the length of the period.

Rule for <rate>: Must be expressed as a decimal.

#### <casharray>()

Array of Doubles that specifies the name of an existing array of cash flow values. The <casharray> array must contain at least one positive value (receipt) and one negative value (payment).

The following example assumes you have filled the array MyArray with a series of cash flow values, and that the interest rate is 11%. What is the net present value? The answer is assigned to the variable NetPValue.

NetPValue = NPV(.11,MyArray())

#### Returns Double

While PV determines the present value of a series of constant payments, NPV does the same for a series of varying payments. Net present value is the value in today's dollars of all future cash flows associated with an investment minus any initial cost. In other words, it is that lump sum of money that would return the same profit or loss as the series of cash flows in question, if the lump sum were deposited in a bank today and left untouched to accrue interest at the rate given by <rate> for the same period of time contemplated by the cash flow stream.

# **Rules** The NPV investment begins one period before the date of the first cash flow value and ends with the last cash flow value in the array.

- If your first cash flow occurs at the beginning of the first period, its value must be added to the value returned by NPV and must not be included in the cash flow values of <casharray>().
- You must express cash paid out (such as deposits to savings) using negative numbers, and cash received (such as dividend checks) using positive numbers.
- <casharray>() must contain at least one negative and one positive number.
- In cases where you have both a positive cash flow (income) and a negative one (payment) for the same period, use the net flow for that period.
- If no cash flow or net cash flow occurs for a particular period, you must enter 0 (zero) as the value for that period.
- **Tip** Because NPV uses the order of values within the array to interpret the order of payments and receipts, be sure to enter your payment and receipt values in the correct sequence.
- **Example** The following example returns the net present value for a series of cash flows contained in an array, CashFlows(). The variable, ReturnRate, represents the fixed internal rate of return. The example assumes Option Base is set to zero.

```
Sub Start()
  ' Set up the array
  Static CashFlows(6) As Double
  Dim Fmt As String, ReturnRate As Double
  Dim Msg As String, NetPVal As Double
  Super::Start( )
  ' Money display format
  Fmt = "$###, ##0.00"
  ' Set fixed internal rate
  ReturnRate = .12
  ' Business start-up expense
  CashFlows(0) = -100000
  ' Positive cash flows for income for four successive years:
  CashFlows(1) = 20000: CashFlows(2) = 40000
  CashFlows(3) = 80000: CashFlows(4) = 0: CashFlows(5) = 10000
  ' Calc net present value
  NetPVal = NPV( ReturnRate, CashFlows )
  Msq = "The net present value of these cash flows is: "
     & Format( NetPVal, Fmt ) & ". "
+
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also FV function IRR function PV function

# **Oct, Oct\$ functions**

Converts a numeric expression from decimal to octal notation, and from numeric to string.

Syntax Oct(<numeric exprs>)

Oct\$(<numeric exprs>)

#### Parameter <numeric exprs>

Numeric expression to be converted from decimal to octal notation.

The following conditions apply to <numeric exprs>:

- Oct[\$] rounds <numeric exprs> to the nearest whole number before evaluating it.
- If <numeric exprs> is a String, it is parsed according to the formatting rules of the current run-time locale.

For example, the following statements are equivalent. Each returns &O010, which is the full form of the octal notation equivalent of decimal number 8.

```
"&O" & Oct$(8)
"&O" & Oct$(2*4)
```

In the following example, the first statement returns the decimal value 16, but the second statement generates an error because the return value of Oct[\$] is not preceded by the octal prefix, &O:

```
("&O" & Oct$(8)) * 2
Oct$(8) * 2
```

Returns Oct: Variant Oct\$: String

- If <numeric exprs> evaluates to Null, Oct[\$] returns Null.
- If <numeric exprs> is an Integer, Variant of type 2 (Integer), Variant of type 0 (Empty) or any other numeric or Variant data type, Oct[\$] returns up to 11 octal characters.
- - To generate the full octal representation of <numeric exprs>, supply the radical prefix &O, because Oct[\$] does not return that component.
- **Example** The following example generates a decimal number, then uses Oct[\$] to convert that number to octal notation:

```
Sub Start( )
   Dim Msg, Num
   Super::Start( )
   Num = Int(100000 * Rnd)
   Msg = Num & " in decimal notation is &O"
+   & Oct(Num) & " in octal notation."
   ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Hex, Hex\$ functions Val function

### **On Error statement**

Tells the program what to do if an error occurs at run time.

- Syntax On Error { GoTo <line | line label> | Resume Next | GoTo 0 }
- **Description** Unless you use On Error, any run-time error that occurs is fatal. Actuate Basic generates an error message and stops executing the program.

If another error occurs while an error handler is active, Actuate Basic passes control back through any previous calling procedure or procedures until it finds an inactive error handler, which it then activates. If it cannot find such an inactive error handler, the error is fatal.

Each time control passes back to a calling procedure, that procedure becomes the current one. Once an error handler in any procedure succeeds in handling an error, program execution resumes in the current procedure at the point designated by its own error handler's Resume statement—which may not be the error handler or the Resume you perhaps intended to execute a few levels down the calling hierarchy.

An error-handler is automatically disabled when you exit its procedure.

- **Rules** The error handler indicated by <line> or <line label> cannot be a sub or function procedure. It must be a block of code marked by a line label or line number.
  - To prevent error-handling code from becoming part of the normal program flow even when there is no error—that is, from inadvertently executing when it should not—place an Exit Sub or Exit Function immediately before the label of the error handler.

#### Parameters GoTo

Clause that enables the error-handling routine that starts at <line >, or, more commonly, <line label>. Enabling the routine means that if a run-time error occurs, program control branches to <line>. The error-handler at <line> remains active until a Resume, Exit Sub, or Exit Function is executed.

#### <line> or <line label>

The line number or line label to which the program branches when a run-time error occurs. error occurs. error.

For example, the following statement tells Actuate Basic to branch to the label ErrorTrap when an error occurs:

On Error GoTo ErrorTrap

#### **Resume Next**

Clause specifying that when a run-time error occurs, control passes to the statement immediately following that in which the error occurred. That is, directs Actuate Basic to continue executing code even though an error has occurred.

#### GoTo 0

Clause that disables any enabled error handler in the current procedure. <GoTo 0> cannot be used to specify line number 0 as the start of the errorhandling code.

- **Tips** An error-handling routine should be exited only by using some form of Resume.
  - You can use Err to obtain the error number of a run-time error.
  - Your error-handling routine should test or save the value of Err before any other error can occur, or before it calls any procedure that could cause an error.
  - You can use Error[\$] function to return the error message associated with any given run-time error returned by Err.
- **Example** The following example generates an error, then displays the error's number and description:

```
Sub Start()
  Dim Drive As String, Msg As String
  Super::Start( )
  ' Set up error handler
  On Error GoTo ErrorHandler7
   ' Attempt to open file
  Open Drive & ":\Test\X.dat" For Input As #1
  Close #1
   ' Exit before entering the error handler
  Exit Function
  ' Error handler line label
ErrorHandler7:
  Msq = "Error " & Err & " occurred, which is: " & Error$(Err)
  ShowFactoryStatus ( Msg )
  ' Resume procedure
Resume Next
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Erl function Err function Error, Error\$ functions Resume statement

### **Open statement**

Enables reading and writing—input and output (I/O)—to a file.

Syntax Open <file name> [For <mode>[encoding\_name]] [Access <permissions>] [<lock>] As [#]<file number> [Len = <record or buffer size>] **Description** You must open a file before any I/O operation can be performed on it. Open allocates a buffer to the file for I/O, and determines the mode of access used with the buffer.

In Actuate Basic there are three types of file access, Random, Sequential and Binary. The type you use in Open causes Actuate Basic to assume that <file name> has certain characteristics. These assumptions are summarized in Table 6-36.

Access	Mode	Actuate Basic assumes <file name=""></file>
Random	Random	Consists of a series of records of identical length, and all of the same data type.
Sequential	Input Output Append	Is a text file, or consists of a series of text characters and/or of text formatting characters such as newline.
Binary	Binary	Can consist of data of any type, with records of any length, or even of variable lengths.

**Table 6-36**Assumptions for <file name> based on access type

The following conditions apply to the Open Statement:

- In Binary, Input, and Random modes, you do not need to first close an open file before opening it with a different <file number>.
- In Append and Output modes, you must first close an open file before opening it with a different <file number>.

#### Parameters <file name>

String expression, enclosed in double quotes, that is the name of the target file, as shown in the following example:

Open "C:\Windows\Mynotes.file" As #1

If the file does not exist when it is opened for Append, Binary, Output, or Random modes, it is created and then opened.

#### For <mode>

Clause that specifies the file mode. Indicates whether and how to read from or write to the file. <mode> might be Append, Binary, Input, Output, or Random, as shown in the following example:

Open "C:\Windows\Mynotes.fil" For Input As #1

#### <encoding\_name>

String expression that specifies the encoding value.

The following conditions apply to <encoding\_name>:

 If <encoding\_name> is not specified or is an invalid value, Open uses the value of the current run-time encoding.  If <encoding\_name> is "text", Open assumes the file uses the same code page as the current system.

#### Access <permissions>

Clause that specifies what operations this process is permitted to perform on <file name>. Another process can be more or less restricted than this one is. The <permissions> keyword can consist of any one of the following: Read, Write, or Read Write. If another process already opened the file and made <permissions> unavailable to others, the Open operation fails and a Permission denied error occurs. For example:

Open "C:\Windows\Mynotes.fil" For Input Access Read As #1

#### <lock>

Keyword or set of keywords that specifies what operations other processes are permitted to perform on <file name>. In a multiuser environment or multiprocessing environment, <lock> restricts access by other processes or users. The <lock> keyword can consist of any one of the following: Shared, Lock Read, Lock Write, or Lock Read Write. For example:

Open "C:\Windows\Mynotes.fil" For Binary As #1 Shared

#### As <file number>

Clause that assigns a number to the open file.

#### <file number>

Integer that specifies the number that is associated with the file as long as it remains open. Other I/O statements can then use <file number> to refer to the file. <file number> must be between 1 and 255, inclusive.

#### Len = <record or buffer size>

Clause specifying how much information to take from the file into the buffer.

Table 6-37 summarizes the behavior of the Len = <record or buffer size> clause, which depends on the setting of <mode>.

Table 6-37	Relationship of mode types and the behavior of Len = <record or<br="">buffer size&gt; clauses</record>
Mode	Len = <record buffer="" or="" size=""></record>
Random	Specifies the record length, the number of bytes in a record. Default for <record buffer="" or="" size=""> in bytes is 128.</record>
Append Input Output	Specifies the number of bytes to be placed in the buffer. Default for <record buffer="" or="" size=""> in bytes is 512.</record>
Binary	Ignored.

#### <record or buffer size>

Integer expression that indicates either the record length or buffer size depending on <mode>. <record or buffer size> must be positive and less than or equal to 32,767 bytes. For example:

Open "C:\Windows\Mynotes.fil" For Input As #1 Len = 512

#### Options for Random

**(mode)** Keyword that specifies random-access file mode. Instructs Actuate Basic to make three attempts to open the file with these permissions, in the following order:

- Read and write
- Write-only
- Read-only

#### Binary

Keyword that specifies binary file mode so you can read and write information to any byte position in the file. If you do not supply an Access clause, instructs Actuate Basic to make three attempts to open the file, following the same order as for Random files. If <file name> does not exist, it is created, then opened. Use Get and Put to write to any byte position in a Binary file.

#### Input

Keyword that specifies sequential input mode. <file name> must exist.

#### Output

Keyword that specifies sequential output mode. If <file name> does not exist, it is created, then opened.

#### Append

Keyword that specifies sequential output mode, and sets the file pointer to the end of the file. If <file name> does not exist it is created, then opened. Use Print or Write to append to the file. For example:

Open "C:\Windows\Mynotes.fil" For APPEND As #1

The default for <mode> is Random.

#### Options for Read

**Access** Keyword instructing Actuate Basic to open the file for a reading process only.

#### Write

Keyword that instructs Actuate Basic to open the file for a writing process only.

#### **Read Write**

Keyword that instructs Actuate Basic to open the file for both reading and writing processes.

Access is valid only if <mode> is Random, Binary, or Append.

# Options for Shared <a href="https://www.shared.com">Shared Shared Shared

Keyword indicating that any process on any machine has permission to read from or write to <file name> while this Open is in effect.

#### Lock Read

Keyword indicating that no other process has permission to read <file name> while this Open is in effect. Lock Read is applied only if no other process has a previous read access to the file.

#### Lock Write

Keyword indicating that no other process has permission to write to <file name> while this Open is in effect. Lock Write is applied only if no other process has a previous write access to the file.

#### Lock Read Write

Keyword indicating that no other process has permission to read from or write to <file name> while this Open is in effect.

Lock Read Write can only occur if:

- No other process has already been granted read or write access.
- A Lock Read or Lock Write is not already in effect.

The default for <lock> is Lock Read Write.

- **Tips** Use Random access files when you want easy access to individual records. Do not use them when you want to conserve storage space.
  - Use Sequential access files when you want to process files consisting only of text. Do not use them for other types of information.
  - Use Binary access files when you want to save storage space by varying your record lengths, sizing them only as needed to fit the data they contain. Do not use them if you do not want to keep track of the differently sized records.
- **Example** The following example opens six files for sequential output, then puts test data into them:

```
Sub Start( )
   Dim FileName As String, FileNum As Integer, I As Integer
   Dim UserMsg As String, SampleTxt As String
   Super::Start( )
' Create test string
   SampleTxt = "Here is some sample text."
   For I = 1 To 6
' Determine file number
     FileNum = FreeFile
   FileName = "TEST" & FileNum & ".DAT"
     ' Open for sequential output
     Open FileName For Output As #FileNum
```

```
' Write test data to file
    Print #FileNum, SampleTxt
    Next I
' Close all open files
    Close
    UserMsg = "Six test files have been created on your disk. "
+ & "The files will now be deleted."
    ShowFactoryStatus( UserMsg )
    ' Delete the test files
Kill "TEST?.DAT"
End Sub
```

The following example opens eight files specifying the encoding for each file:

```
Sub Start() As Boolean
Super::Start()
Open "Korean.txt" For Input "windows-949" As #1
Open "KoreanOut.txt" For Output "windows-949" As #2
Open "Japanese.txt" For Input "windows-932" As #3
Open "JapaneseOut.txt" For Output "windows-932" As #4
Open "Chinese.txt" For Input "windows-936" As #5
Open "ChineseOut.txt" For Output "windows-936" As #5
Open "ChineseOut.txt" For Output "windows-936" As #6
Open "Unicode.txt" For Input " UCS-2" As #7
Open "UnicodeOut.txt" For Output "UCS-2" As #8
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Close statement FreeFile function Get statement Print statement Put statement Write statement

### **Option Base statement**

Declares the default lower bound for array subscripts.

**Syntax** Option Base {0 | 1}

**Description** • Affects only arrays in the same module.

- Has no effect on arrays within user-defined types. The lower bound for all such arrays is always 0.
- Can appear only once in a module.
- Can occur only in the Declare... End Declare section.
- Must be used before you declare the dimensions of any array.

#### Parameters

Specifies that the lowest subscript for arrays in the current module is 0.

1

Ω

Specifies that the lowest subscript for arrays in the current module is 1.

Default: 0.

Rule: Must be 0 or 1.

**Example** The following statement declares that the lowest subscript for any arrays you create is 1:

Option Base 1

- **Tips** The To clause in Dim, ReDim, and Static provides a more flexible way to control the range of an array's subscripts.
  - If you do not explicitly set the lower bound with a To clause, you can use Option Base to change the default lower bound to 1.
- **Example** The the following example uses Option Base to override the default base array subscript value of 0:

After you compile and run this example once, delete the Option Basic statement lines from the BAS file and compile and test the example again. The lower bound reverts to zero, the default.

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Dim statement LBound function ReDim statement Static statement

# **Option Compare statement**

Declares default comparison mode used when comparing string data.

Syntax Option Compare { Binary | Text }

#### Parameters Binary

String comparisons in the module are case-sensitive by default. For example, if you use Option Compare Binary, then by default McManus does not match MCMANUS or mcmanus.

#### Text

String comparisons in the module are not case-sensitive by default. If you use Option Compare Text, then by default McManus matches MCMANUS or mcmanus.

**Rules** Use Option Compare within a Declare... End Declare section.

- If used, Option Compare must appear before any statements that declare variables or define constants.
- **Example** The following example shows the difference between using Option Compare Text and Option Compare Binary:

```
Declare
   Option Compare Text
End Declare
Sub Start( )
   Dim NameStr As String, UName As String, LName As String
   Dim Msg As String
   Super::Start( )
   NameStr = "Mary Ndebele"
   UName = UCase( NameStr )
   LName = LCase( NameStr )
   If UName Like LName Then
       Msg = UName & " matches " & LName
   Else
       Msg = UName & " does NOT match " & LName
```

```
End If
ShowFactoryStatus( Msg )
End Sub
```

When you compile your report and try the example, note that the uppercase and lowercase version of the customer's name match. Now delete the Option Compare Text statement from your Actuate Basic source code and compile the example again. This time, the uppercase and lowercase versions of the customer's name will not match because without any Option Compare statement, Option Compare Binary is in effect by default.

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Format, Format\$ functions StrComp function

# **Option Strict statement**

Specifies whether declarations must be typed.

Syntax Option Strict {On | Off}

**Description** Option Strict can occur only in the Declare... End Declare section, and must be used before any other source code statements.

#### Parameters On

Checks that all variable, function, and sub procedure argument declarations are typed. Declarations without an As <data type> clause are flagged during compilation.

#### Off

Specifies that no data type checking is done, and untyped variables, functions, and arguments are of type Variant.

The default is Off.

The following statement makes data typing required for all declarations, so that no variable types are set to Variant by default:

Option Strict On

**Example** The following example uses Option Strict to enforce data typing in declarations.:

```
Declare

' Equivalent to Strict On

Option Strict

Global GlobalWithNoType

End Declare
```

#### ParseDate function

```
Class Strict

Dim v

Dim var As Variant' Explicit Variants are OK

Static s

Function ff( a )

Dim fv

Static fs

End Function

Sub ss( a )

End Sub

End Class 'Strict

Function fff( a )

End Function

Sub sss( a )

End Sub
```

When you compile this example several compiler errors are generated, all having to do with missing data types. Change the Option Strict statement to Option Strict Off and compile the example again. No errors are generated and all the untyped declarations default to type Variant.

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Dim statement Function...End Function statement Global statement Static statement Sub...End Sub statement

# **ParseDate function**

Controls how dates are converted. Use when your program must be localeindependent, or when the format of date strings in your database differs from the format of the current locale. You can also specify the locale to use for date conversion.

**Syntax** ParseDate(<strDateSource>, <strFormatSpec>)

ParseDate(<strDateSource>, <strFormatSpec>, <locale>)

#### Parameters <strDateSource>

String expression. The date string to convert from. The source date string. <strDateSource> must be, or evaluate to, a valid date string. For example:

"2/1/2010" "1 Feb, 2010"

```
"February 1, 2010"
Format$((DateSerial(2010, 2, 1)), "mm/dd/yyyy")
```

#### <strFormatSpec>

String expression. The date string to convert from. The format in which the date string currently appears. Range of valid dates is January 1, 100 through December 31, 9999, inclusive.

The following conditions apply to <strFormatSpec>:

- Must be, or evaluate to, a valid format string.
- You can use the format characters described in Table 6-38 to convert date expressions.

	Tormat characters for converting date expressions	
Format character	Description	
?	Indicates an optional field.	
d	Specifies that the next field is a day. Must appear only once in a date expression. If you use the d format character, you must also use the m and y format characters.	
1	Specifies that the date format specified in the current locale should be used. If this character is used, it must be the only format specified.	
m	Specifies that the next field is a month. Must appear only once in a date expression. If you use the m format character, you must also use the y and d format characters.	
р	The base year for the pivot.	
t	Specifies that the next field is the time. Must appear only once in a date expression.	
W	Specifies that the next field is a (noise) word, such as the name of the day of the week.	
У	Specifies that the next field is a year. Must appear only once in a date expression. If you use the y format character, you must also use the m and d format characters.	

 Table 6-38
 Format characters for converting date expressions

• To convert a date string from the French date format, use the following format string for <strFormatSpec>:

"dmy"

 To convert a date from the U.S. date format, use the following format string for <strFormatSpec>:

"mdy"

 To convert a date from the ISO date format, use the following format string for <strFormatSpec>:

"ymd"

 To convert date expressions that contain a day of the week or a word, use the ParseDate() function with the w format character. You can also use the optional field indicator format character '?'.

Table 6-39 shows the ParseDate() return values with a day of week specified.

Format character	Date expression	Return value
w?mdy	Friday, September 28, 2001	2001-9-28
w?mdy	September 28, 2001	2001-9-28
wmdy	Friday, September 28, 2001	2001-9-28
wmdy	September 28, 2001	NULL
wmdy	on September 28, 2001	2001-9-28
mdy	September 28, 2001	2001-9-28

Table 6-39Converting date expressions that specify<br/>a day of the week

The locale settings control default date parsing. The locale settings generally do not include the w or w? field. Therefore, if a date expression contains a day of week and the locale setting is not specified properly for the day of week, Actuate date and time functions, such as DateAdd(), DateDiff(), DatePart(), DateValue(), and IsDate() return a null date.

For more information about formatting data, see Format function.

#### <locale>

String expression specifying the locale to use for date conversion. Must be in the <language>\_<location> format, for example fr\_FR for French locale.

Rule: If <locale> is Null or invalid, ParseDate uses the current run-time locale.

Returns Date

If the date string is invalid, ParseDate returns Null.

**Description** Actuate Basic implicitly converts strings to dates and dates to strings. This conversion is patterned after Visual Basic. If you want your applications to be Y2K-compliant, to run in multiple locales, or to work with string date data imported from another application, be aware of the limitations of the implicit date conversions.

Implicit string-to-date conversions occur when you:

- Pass a string to a function that takes a date argument
- Assign a string to a date variable
Use the CDate or CVDate functions

In all these cases, Actuate converts dates using the date format of the current locale or the locale specified by the <locale> parameter and Actuate's default pivot year of 30. Actuate converts a date of 2/1/99 to a date of February 1, 1999 in the U.S., and to January 2, 1999 in the UK.

Implicit date-to-string conversion occurs when you:

- Pass a date to a function that takes a string argument
- Assign a date to a string
- Use the CStr function
- Output the date using the Print statement

In all cases, Actuate performs implicit date-to-string conversion using the date format of the current locale or the locale specified by the <locale> parameter and Actuate's default pivot year of 30. If you assign the date February 1, 1999 to a string in the U.S., the result will be 2/1/1999. In the UK, however, the string will be 1/2/1999.

Like Visual Basic, Actuate Basic attempts to parse a date using a number of different formats in order, until one works or until there is nothing else to try. This means that a date that is invalid for the given locale might still be parsed using a different format. For example, in the U.S. locale, a date of 13/1/99 converts to January 1, 1999 using the European format.

The format rules, in order of use, are:

- Current locale
- U.S. format (m/d/y)
- European format (d/m/y)
- ISO format (y/m/d)

When you write data streams, ParseDate allows you to precisely control how dates are converted. Hence, if Basic's default rules do not work, you can use ParseDate to perform the conversion correctly.

- **Tips** Use ParseDate to convert type String date input to type DateTime as early in the program run as possible.
  - Convert dates to strings as late as possible before you display or output the dates to a file. To manage this conversion, use the Format function.
  - Always explicitly declare your date variables to be of type DateTime, do not rely on using Variants.
  - To accurately convert dates of type Date or type DateTime to strings, use the Format function.

- To accurately convert dates of type String to dates of type DateTime, use the ParseDate function.
- To convert date expressions containing a day of week and/or a word, use ParseDate with the format character 'w'. You can also use the optional field indicator format character '?'.
- **Example** The following example converts a date in French format to a date in U.S. format:

```
Sub Start( )
Dim dtSafeDate As Date
Dim strFr_Date As String, strUS_Date As String
Dim strMsg As String
Super::Start( )
strFr_Date = "10/12/1959"
dtSafeDate = ParseDate(strFr_Date, "dmy")
strMsg = "In US format, strFr_Date is shown as: "
+ & CStr(dtSafeDate)
ShowFactoryStatus( strMsg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also CVDate function Format, Format\$ functions VarType function

# **ParseNumeric function**

Parses the numeric expression using the specified radix and thousand separator characters.

Syntax ParseNumeric(<numeric expression>, <radix character>, <thousand separator>)

ParseNumeric(<numeric expression>, <radix character>, <thousand separator>, <currency symbol>)

Parameters <numeric expression> Numeric expression of type String to be converted to a Double data type.

# <radix character>

String expression that specifies the radix character used in the numeric expression. If <radix character> is Null or an empty string, ParseNumeric uses the currency symbol of the current run-time locale.

### <thousand separator>

String expression that specifies the thousand separator character used in the numeric expression.

The following conditions apply to <thousand separator>:

- If <thousand separator> contains more that one character, ParseNumeric uses only the first character.
- If <thousand separator> is Null or an empty string, ParseNumeric uses the thousands separator character of the current run-time locale.

#### <currency symbol>

String expression that specifies the currency symbol used in the numeric expression.

The following conditions apply to <currency symbol>:

- If <currency symbol> contains more that one character, ParseNumeric uses only the first character.
- If <currency symbol> is Null or an empty string, ParseNumeric uses the currency symbol of the current run-time locale.
- Returns Variant
  - If <numeric expression> is Null, ParseNumeric returns Null.
  - If the expression is not valid or cannot be parsed using the specified decimal or thousand separator character, ParseNumeric returns Null.
- **Example** Each of the following statements returns 123456.78:

```
ParseNumeric("123,456.78", ".", ",", NULL)
ParseNumeric("123.456,78", ",", ".", NULL)
ParseNumeric("123!456*78", "*", "!", NULL)
```

The following statement returns 1500.00:

```
ParseNumeric("$1,500.00", ".", ",", "$")
```

# **Pmt function**

Returns the payment for an annuity, based on periodic, constant payments, and on an unvarying interest rate.

Syntax Pmt(<rate per period>, <number pay periods>, <present value>, <future value>, <when due>)

#### Parameters <rate per period>

Numeric expression that specifies the interest rate that accrues per period. <rate per period> must be given in the same units of measure as <number pay

periods>. For instance, if <number pay periods> is expressed in months, then <rate per period> must be expressed as a monthly rate.

#### <number pay periods>

Numeric expression that specifies the total number of payment periods in the annuity. <number pay periods> must be given in the same units of measure as <rate per period>. For instance, if <rate per period> is expressed as a monthly rate, then <number pay periods> must be expressed in months.

#### <present value>

Numeric expression that specifies the value in today's dollars of a future payment, or stream of payments.

For example, if you put \$23.94 in the bank today and leave it there for 15 years at an interest rate of 10% compounded annually, you end up with about \$100. In this case, the present value of \$100 is approximately \$23.94.

#### <future value>

Numeric expression that specifies the cash balance you want after you have made your final payment.

- You set up a savings plan with a goal of having \$75,000 in 18 years to pay for your child's education. For this plan, the future value is \$75,000.
- You take out a loan for \$11,000. The future value is \$0.00, as it is for any typical loan.

#### <when due>

Numeric expression that specifies whether each payment is made at the beginning (1), or at the end (0) of each period. The default is the end of the period (0). <when due> must be 0 or 1.

The following example assumes you are making monthly payments the first of each month on a loan of \$20,000, over 36 months, at an APR of 11.5%. How much will each of your payments be? The answer (\$653.26) is assigned to PaymentAmt.

PaymentAmt = Pmt(.115/12, 36, -20000, 0, 1)

#### Returns Double

An annuity is a series of cash payments, constant in value, made over a period of time. An annuity can be an investment, such as a monthly savings plan, or a loan, such as a home mortgage.

- **Rules** rate per period> and <number pay period>> must be expressed in terms of
  the same units (weekly/weeks, monthly/months, yearly/years).
  - You must express cash paid out (such as deposits to savings) using negative numbers, and cash received (such as dividend checks) using positive numbers.
- **Example** The following example uses information about a loan to determine the amount of payments:

```
Declare
  Global Const ENDPERIOD = 0
  ' When payments are made
  Global Const BEGINPERIOD = 1
End Declare
Sub Start()
  Dim Fmt As String, FutureVal As Double, PresVal As Double
  Dim APR As Double, TotPmts As Integer, PayWhen As Integer
  Dim Period As Integer, InterestPmt As Double
  Dim PrincipPmt As Double, Payment As Double, Msq As String
  Super::Start( )
  ' Specify money format
  Fmt = "$###, ###, ##0.00"
   ' Normally 0 for a loan
  FutureVal = 0
  PresVal = 250000
  APR = 0.07
  TotPmts = 200
  ' Assume payment at beginning of period
  PayWhen = BEGINPERIOD
     Payment =
     Pmt( APR / 12, TotPmts, -PresVal, FutureVal, PayWhen )
+
  Msg = "Your payments are " & Format ( Payment, Fmt ) & "."
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also FV function IPmt function NPer function PPmt function PV function Rate function

# **PPmt function**

Returns the principal payment for a given period of an annuity, based on periodic, constant payments, and on an unvarying interest rate.

Syntax PPmt(<rate per period>,<single period>, <number pay periods>, <present value>, <future value>, <when due>)

# **Parameters** <rate per period> Numeric expression that specifies the interest rate that accrues per period.

Rule for <rate per period>: Must be given in the same units of measure as <number pay periods>. For instance, if <number pay periods> is expressed in months, then <rate per period> must be expressed as a monthly rate.

#### <single period>

Numeric expression that specifies the particular period for which you want to determine how much of the payment for that period represents interest. <single period> must be in the range 1 through <number pay periods>.

### <number pay periods>

Numeric expression that specifies the total number of payment periods in the annuity. <number pay periods> must be given in the same units of measure as <rate per period>. For instance, if <rate per period> is expressed as a monthly rate, then <number pay periods> must be expressed in months.

### <present value>

Numeric expression that specifies the value today of a future payment, or stream of payments.

For example, if you put \$23.94 in the bank today and leave it there for 15 years at an interest rate of 10% compounded annually, you end up with about \$100. In this case, the present value of \$100 is approximately \$23.94.

### <future value>

Numeric expression that specifies the cash balance you want after you have made your final payment.

- You set up a savings plan with a goal of having \$75,000 in 18 years to pay for your child's education. For this plan, the future value is \$75,000.
- You take out a loan for \$11,000. The future value is \$0.00, as it is for any typical loan.

# <when due>

Numeric expression that specifies whether each payment is made at the beginning (1), or at the end (0) of each period. The default is 0 and <when due> must be 0 or 1.

The following example assumes you are making monthly payments at the first of each month on a loan of \$20,000, over 36 months, at an APR of 11.5%. How much of your 5th payment represents principal? The answer (\$481.43) is assigned to Principal5.

Principal5 = PPmt(.115/12, 5, 36, -20000, 0, 1)

# Returns Double

An annuity is a series of cash payments, constant in value, made over a period of time. An annuity can be an investment, such as a monthly savings plan, or a loan, such as a home mortgage.

Each payment in an annuity consists of two components: principal and interest. PPmt returns the principal component of the payment.

- **Rules** <rate per period> and <number pay periods> must be expressed in terms of the same units such as weeks, months or years.
  - You must express cash paid out (such as deposits to savings) using negative numbers, and cash received (such as dividend checks) using positive numbers.
- **Example** The following example uses various particulars about a loan. It then returns, for a single payment period the user specifies, the portions of that payment that are interest and principal.

```
Declare
  Global Const ENDPERIOD = 0
  ' When payments are made
  Global Const BEGINPERIOD = 1
End Declare
Sub Start()
  Dim Fmt As String, FutureVal As Double, PresVal As Double
  Dim APR As Double, TotPmts As Integer, PayWhen As Integer
  Dim Period As Integer, IntPmt As Double, PrinPmt As Double
  Dim Payment As Double, Msg As String
  Super::Start( )
   ' Specify money format
  Fmt = "$###, ###, ##0.00"
   ' Normally 0 for a loan
  FutureVal = 0
  PresVal = 450000
  APR = 0.0725
  TotPmts = 180
   ' Assume end period payrment
  PayWhen = ENDPERIOD
  Period = Int(180 * Rnd + 1)
  IntPmt = IPmt( APR / 12, Period, TotPmts, -PresVal,
+ FutureVal, PayWhen )
  PrinPmt = PPmt( APR / 12, Period, TotPmts, -PresVal,
     FutureVal, PayWhen )
+
  Payment = Pmt( APR / 12, TotPmts, -PresVal, FutureVal,
     PayWhen )
+
  Msg = "At period " & Period & " you will pay a total "
     & " of " & Format (Payment, Fmt) & ", representing "
+
     & Format(PrinPmt, Fmt) & " principal and "
     & Format(IntPmt, Fmt) & " interest.
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also FV function IPmt function NPer function Pmt function PV function Rate function

# **PreciseTimer function**

- **Example** Returns a value in seconds. The absolute value returned is not defined, but comparing values from two calls gives the time between those calls. This function is supported on Windows systems only, and may not work on some Windows systems due to hardware limitations. If the function is called on the wrong operating system or is not supported by the hardware, it returns Null. This function is intended only for use as a development and debugging aid in e.Report Designer Professional, and is not supported for production use.
- Syntax PreciseTimer
- **Returns** Double value in seconds.
- **Example** The following example retrieves time information twice, and computes the amount of time required between the two calls. Create variables timeOne, timeTwo, and timeUsed of type Double on the class.

```
Sub Start( )
    ' Retrieve the first time
    timeOne = PreciseTimer
End Sub
Sub Finish( )
    ' Retrieve second time
    timeTwo = PreciseTimer
    ' Compute time required
    timeUsed = timeTwo - timeOne
    ShowFactoryStatus("Time used between two calls: " & timeUsed)
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

# Print statement

Writes unformatted data to a sequential file.

- Syntax Print #<file number>, [ [{ Spc(<number of blanks>) | Tab(<tab column>) } ] [ <expression list> ] [ { ; | , } ]... ]
- **Description** Print usually writes Variant data to a file the same way it writes any other Actuate Basic data type. Table 6-40 lists exceptions to this behavior.

 Table 6-40
 Exceptions to how Print writes Variant data to a file

Data type	Print writes this to the file
Variant of VarType 0 (Empty)	Nothing at all
Variant of VarType 1 (Null)	The literal value Null
Variant of VarType 7 (Date)	The date, using whatever Short Date format is defined in the Control Panel
Variant of VarType 7 (Date), but with either the date or time component missing or zero	Only the part of the date provided

The following conditions apply to the Print statement:

- <file number> must match the number of a currently open file.
- The file corresponding to <file number> must be open under either the Output or Append mode.
- You must have write access to the open file. That is, the file must not have been opened using a Lock Write or Lock Read Write clause.
- The data cannot be an object variable, user-defined data type structure, handle to a class, CPointer, or OLE object.

#### Parameters <file number>

Numeric expression that is the file number used to Open a sequential file to which Print writes data.

The following example writes Hello, world! to the file opened under file number 1, and appends a carriage-return-linefeed character-pair (CRLF) to the file, so that whatever is next written to the file appears on the next line:

```
Print #1, "Hello, world!"
```

The following example shows three statements that work together, although they can be separated from one another by any number of lines of code. The first statement opens a file called Mynotes.fil, the second one writes to that file, and the third, by closing it, turns the file over to the operating system to complete the process of writing it to the disk.

```
Open "C:\Myfiles\Mynotes.fil" For Output As #2
Print #2, "This datum.", "This datum, too. "
Close #2
```

## Spc(<number of blanks>)

Clause that inserts spaces into the output. May be repeated; separated by commas or semicolons.

#### <number of blanks>

Numeric expression that specifies the number of spaces Spc inserts. For example:

```
Print #2, "This datum.", Spc(10), "This datum, too.", Spc(5),
+ "And this."
```

# Tab(<tab column>)

Clause that tabs to the specified column before the program prints <expression list>. May be repeated; separated by commas or semicolons.

# <tab column>

Numeric expression that specifies the column to which the program jumps before it prints the <expression list>. For example:

```
Print #2, "This datum."; Tab(5); "This datum, too."; Tab(7);
+ "And this."
```

### <expression list>

Numeric or string expressions that specify the data that Print writes to the file. There can be any number of these. The default is a blank line, if comma is used.

# ; (semicolon)

Character (semicolon) that determines the position of the next character printed. Instructs Actuate Basic to print the next character immediately after the last one.

#### , (comma)

Character (comma) that determines the position of the next character printed. Instructs Actuate Basic to print the next character at the start of the next print zone. Print zones begin every 14 columns.

The default position for the next character is the next line.

The following example writes the number 250, and the content of P\$ to the file. The comma after the number 250 causes P\$ to be written at the next print zone. The semicolon after P\$ causes Actuate Basic to suppress the printing of the carriage-return-linefeed character-pair, so that whatever is next written to the file occurs immediately after the text in P\$.

Print #1, 250, P\$;

- **Tips** To ensure integrity of separate data fields, use Write instead of Print. Write delimits data properly and ensures that the data can be read correctly in any locale.
  - Use Print with ASCII text files.
  - Spacing of data displayed on a text screen with monospaced characters may not work well when the data is redisplayed in a graphical environment using proportionally spaced characters.

**Example** The following example creates a file, Testifle.txt, writes some data to it, then closes it. Then, the example reopens the file to read the data just written. Finally, after displaying a message, it closes the file and deletes it.

```
Sub Start()
  Dim TestData As String, UserMsg As String
  Super::Start( )
  ' Open to write file
  Open "Testifle.txt" For Output As #1
  Print #1, "Test of the Print statement."
  ' Print extra blank line to file
  Print #1,
  ' Print in two print zones
  Print #1, "Zone 1", "Zone 2"
  ' Print two strings together
  Print #1, "With no space between"; "these"
  Close #1
  ' Open to read file
  Open "Testifle.txt" For Input As #1
  ' Read entire file
  Do While Not EOF(1)
     ' Read a line
     Line Input #1, TestData
     ' Construct message
     UserMsg = UserMsg & TestData
     ShowFactoryStatus ( UserMsg )
  Loop
  Close #1
  ' Delete file from disk
  Kill "Testifle.txt"
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Open statement Write statement

# Put statement

Writes data from a variable to a disk file.

Syntax Put [#] <open file number>, [<record number>], <variable to write>

**Description** For files opened in Random mode:

 If the length of the data you write is less than the length specified in the Len clause of the Open statement, Put still writes subsequent records on recordlength boundaries. The space between the end of one record and the beginning of the next is padded with the existing contents of the file buffer.

In the following example, the first Put writes the 4 bytes of the variable HoldingVar1 to the disk file Myfile.txt at byte positions 1-4. Any data that was there is overwritten. The second Put, however, does not pick up where the last one left off but instead skips to the record-length boundary (20) and starts counting from there, which means it writes the 5 bytes of HoldingVar2 to byte positions 21-25. The intervening bytes (5-20) remain what they were, but now represent padding between records.

```
Dim HoldingVar1 As String * 4
Dim HoldingVar2 As String * 5
Open "Myfile.txt" For Random As #1 Len = 20
HoldingVar1 = "1234"
HoldingVar2 = "12345"
Put #1, , HoldingVar1
Put #1, , HoldingVar2
```

 Table 6-41 summarizes how Put behaves, depending on the data type of <variable to write>. Open refers to the particular statement that opened the file in question.

<variable to="" write=""></variable>	iable to write> Put writes, in this order		Rules		
Variable-length string		2-byte descriptor containing string length	Record length specified in the Len clause of Open must be at least 2 bytes		
	2	Data in the variable	greater than the actual length of the string.		
Numeric Variant (Variant Types 0-7)		2-byte descriptor that identifies the type of the Variant	Len clause length must be at least 2 bytes greater than the actual number of		
		Data in the variable	bytes required to store the variable.		
String Variant (Variant Type 8)		2-byte Variant Type descriptor	Len clause length must be at least 4 bytes greater		
		2 bytes indicating the string length	than the actual length of the string.		
		Data in the variable			
Any other type of variable	1	Data in the variable	Len clause length must be greater than or equal to the length of the data.		

 Table 6-41
 Relationships of <variable to write> data types to Put behavior

For files opened in Binary mode:

All the Random descriptions and rules above apply except that the Len clause in Open has no effect. Put writes all variables to disk contiguously, that is, with no padding between records.

In the following example, the first Put writes the 4 bytes of the variable HoldingVar1 to the disk file Myfile.txt at byte positions 1-4. The second Put writes the 5 bytes of HoldingVar2 to byte positions 5-10. No data shows between them as padding.

```
Dim HoldingVar1 As String * 4
Dim HoldingVar2 As String * 5
Open "Myfile.txt" For Binary As #1 Len = 20
HoldingVar1 = "1234"
HoldingVar2 = "12345"
Put #1, , HoldingVar1
Put #1, , HoldingVar2
```

For variable-length strings, Put does not write a 2-byte descriptor; it writes the number of bytes equal to the number of characters already in the string.

The following example writes 12 bytes to file number 1:

```
VariLen$ = String$(12, "*")
Put #1, , VariLen$
```

#### Parameters <open file number>

Numeric expression that specifies the number you used in the previously issued Open that opened the file to which you now want to write using Put.

#### <record number>

Numeric expression:

- For files opened in Random mode, <record number> is the number of the record to be written.
- For files opened in Binary mode, it is the byte position at which writing starts, where the first byte in a file is at position 1, the second byte at position 2, and so on.

The default is the next record or byte—that is, the one following the last Get or Put statement, or the one pointed to by the last Seek2 function.

The following conditions apply to <record number>:

 If you omit <record number> you must still include the delimiting commas that would have been on either side of it had you included it. For example:

Put #1, , HoldingVar1

Must be between 1 and 2,147,483,647.

# <variable to write>

String expression that specifies the variable to write to the open file.

The following conditions apply to <variable to write>:

- Cannot be an object variable, user-defined data type structure, handle to a class, CPointer, or OLE object.
- Cannot be an array variable that refers to an entire array, although you can use one that describes a single element of an array.
- **Tip** To avoid corrupting or misreading data, be sure your record lengths and/or variable lengths match the lengths of the data you want to read or write.
- **Example** The following example prompts the user for three customer names, then writes each name to a test file and reads the names:

```
Sub Start()
  Dim CustName As String, Msg As String
  Dim Indx As Integer, Max As Integer
  Super::Start( )
  CustName = String( 20, " " )
  ' Create a sample random-access file using PUT:
  Open "testfil3.txt" For Random As #1 Len = 50
  ' Put records into file on disk
  Put #1, 1, "Customer 1"
  Put #1, 2, "Customer Two"
  Put #1, 3, "Third Customer"
  ' Read the sample random-access file using GET:
  Max = Loc(1)
' Check if records exist
  If Max = 0 Then
     Msg = "Sorry, your file contains no names."
  Else
     Msg = "Your file contains the following names:"
  End If
  ShowFactoryStatus ( Msg )
  For Indx = 1 To Max
' Read from test file
     Get #1, Indx, CustName
     ShowFactoryStatus ( CustName )
  Next Indx
  ' Close file
  Close #1
  Msg = "The test file will now be deleted."
  ShowFactoryStatus (Msg)
' Delete test file
  Kill "testfil3.txt"
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Get statement Open statement Type...End Type statement

# **PV** function

Returns the present value of an annuity based on periodic, constant payments to be paid in the future, and on an unvarying interest rate.

Syntax PV(<rate per period>,<number pay periods>,<each pmt>, <future value>, <when due>)

# Parameters <rate per period>

Numeric expression that specifies the interest rate that accrues per period. <rate per period> must be given in the same units of measure as <number pay periods>. For instance, if <number pay periods> is expressed in months, then <rate per period> must be expressed as a monthly rate.

### <number pay periods>

Numeric expression that specifies the total number of payment periods in the annuity. <number pay periods> must be given in the same units of measure as <rate per period>. For instance, if <rate per period> is expressed as a monthly rate, then <number pay periods> must be expressed in months.

#### <each pmt>

Numeric expression that specifies the amount of each payment. <each pmt> must be given in the same units of measure as <rate per period>. For instance, if <rate per period> is expressed in months, then <each pmt> must be expressed as a monthly payment.

# <future value>

Numeric expression. Specifies the cash balance you want after you have made your final payment.

- You set up a savings plan with a goal of having \$75,000 in 18 years to pay for your child's education. For this plan, the future value is \$75,000.
- You take out a loan for \$11,000. The future value is \$0.00, as it is for any typical loan.

# <when due>

Numeric expression that specifies whether each payment is made at the beginning (1), or at the end (0) of each period. The default is 0 and <when due> must be 0 or 1.

The following example assumes you are considering the purchase of a corporate bond with a \$1000 face value. The bond pays an annual coupon of \$100, matures in 15 years, and the next coupon is paid at the end of one year. The yield to

maturity on similar bonds is 12.5%. What is a fair price for this bond (its present value)? The answer, \$834.18, is assigned to the variable PresentValue.

PresentValue = PV(.125, 15, 100, 1000, 0)

The following examples assumes you have won the lottery. The jackpot is \$10 million, which you receive in yearly installments of \$500,000 per year for 20 years, beginning one year from today. If the interest rate is 9.5% compounded annually, how much is the lottery worth today? The answer, \$4,406,191.06, is assigned to PresentValue.

```
PresentValue = PV(.095, 20, 50000,10000000, 0)
```

The following example assumes you want to save \$11,000 over the course of 3 years. If the APR is 10.5% and you plan to save \$325 monthly, and if you make your payments at the beginning of each month, how much would you need to start off with in your account to achieve your goal? The answer, \$2,048.06, is assigned to StartValue. Note that <each pmt> is expressed as a negative number because it represents cash paid out.

StartValue = PV(.105/12, 3\*12, -325, 11000, 1)

Returns Double

An annuity is a series of cash payments, constant in value, made over a period of time. An annuity can be an investment, such as a monthly savings plan, or a loan, such as a home mortgage. The present value is the value today of a future payment, or of a stream of payments structured as an annuity.

For example, if you put \$23.94 in the bank today and leave it there for 15 years at an interest rate of 10% compounded annually, you end up with about \$100. So, the present value of \$100 is approximately \$23.94.

- - You must express cash paid out (such as deposits to savings) using negative numbers, and cash received (such as dividend checks) using positive numbers.
- **Example** The following example prompts the user for the amount to save or invest each month, the annual percentage rate (APR) of the interest, the total number of payments, a target amount, and when during the month payments are made. Then, it interprets the present value as the amount the user would have to start with to achieve the goal.

```
Declare
Global Const ENDPERIOD = 0
Global Const BEGINPERIOD = 1
End Declare
```

## QBColor function

```
Sub Start()
  Dim EachPmt As Double, APR As Double, Fmt As String
  Dim Msg As String, TotalPmts As Single, PayWhen As Integer
  Dim PresentVal As Double, FutureVal As Double
  Super::Start( )
  ' Specify money format
  Fmt = "$#, ##0.00"
  EachPmt = 500 ' Amount to save each month
  APR = 0.0275 ' Annual percentage rate for theinterest
  TotalPmts = 60 ' Number of months to save
  ' Assume end period payment
  PayWhen = ENDPERIOD
  FutureVal = 50000 ' The goal amount
  PresentVal = PV( APR / 12, TotalPmts, -EachPmt, FutureVal,
     PayWhen )
+
  Msg = "Starting with " & Format( -PresentVal, Fmt )
   & " and saving " & Format( EachPmt, Fmt )
     & " every month at an interest rate of "
+
     & Format( APR, "#0.00%" ) & " for a period of "
     & TotalPmts & " months brings you to your goal of "
     & Format( FutureVal, Fmt ) & "."
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also FV function IPmt function

NPer function Pmt function PPmt function Rate function

# **QBColor function**

Translates the numeric representations for colors from older BASIC languages such as Quick Basic to those used by Actuate Basic.

Syntax QBColor(<number to translate>)

# Parameters <number to translate>

Integer that specifies the number of a color, specifically one of the 16 standard Windows colors.

Rules:

- Must be between 00 and 15, inclusive.
- If <number to translate> evaluates to Null, QBColor returns Null.

Returns Long

**Tips** There are at least four ways to represent a color in Actuate Basic:

- Specify the relative mix of red, green, and blue in a color by using RGB with decimal numbers.
- Directly specify, without using RGB or any function, the relative mix using hexadecimal numbers in a certain format.
- Use one of the 16 standard Windows colors by supplying the appropriate integer to QBColor.
- Use one of the appropriate color name constants from Header.bas.

**Example** The following example displays a variety of color shades on a component:

```
Sub Start( )
   Dim UserAns, Msg
   Super::Start( )
   UserAns = CInt(Rnd * 15)
   Msg = "The hexadecimal code for " & UserAns &
+ " is: " & Hex(QBColor(UserAns))
   ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also RGB function

# **Randomize statement**

Initializes the random number generator so it can generate a new sequence of apparently random numbers.

Syntax Randomize [<seed number>]

**Description** Actuate Basic supplies you with Randomize and Rnd as tools for generating random numbers.

Randomize generates a sequence of pseudo-random numbers by starting with a seed value. The randomizing function then takes that initial seed value and uses it to generate the first number in the pseudo-random sequence. Then it uses that number in turn to produce the second one, and so on. The result is a series of

numbers that vary well enough to seem to have no pattern, or in other words, to be random.

- If you do not use Randomize, Rnd returns the same sequence of numbers every time the program is run.
- Use Randomize before Rnd, or it has no effect.

#### Parameters <seed number>

Numeric expression that specifies the number that initializes the random number generator by supplying it with a new seed value. <seed number> must be between -2,147,483,648 and 2,147,483,647.

For example:

Randomize Randomize 34

- **Tips** To generate numbers that seem truly random, use Randomize—with no argument—at the beginning of your program. In that case, Actuate Basic uses a randomizing seed based on the current running system time, which is almost certain to vary randomly whenever you run the program.
  - Although, you can use Randomize several times throughout a program, it works best if you use it only once, at the beginning.
  - To help you debug certain programs, use Randomize with a specific <seed number>. That way, the same random numbers repeat every time you run the program.
- **Example** The following example simulates rolling a die twice by generating random values between 1 and 6 for each roll:

```
Sub Start()
  Dim DieRoll1, DieRoll2, UserMsg
  Super::Start( )
' Seed random num generator
  Randomize
' First roll
  DieRoll1 = Int(6 * Rnd + 1)
' Second roll
  DieRoll2 = Int(6 * Rnd + 1)
  UserMsg = "You rolled a " & DieRoll1
     & " and a " & DieRoll2
+
     & " giving a total of "
+
     & (DieRoll1 + DieRoll2) & "."
  ShowFactoryStatus ( UserMsg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Rnd function

# **Rate function**

Returns the interest rate per period for an annuity.

**Syntax** Rate(<number pay periods>,<each pmt>, <present value>, <future value>, <when due>, <starting guess>)

### Parameters <number pay periods>

Numeric expression that specifies the total number of payment periods in the annuity. <number pay periods> must be given in the same units of measure as <each pmt>. For instance, if <each pmt> is expressed as a monthly payment, then <number pay periods> must be expressed in months.

#### <each pmt>

Numeric expression that specifies the amount of each payment. <each pmt> must be given in the same units of measure as <number pay periods>. For instance, if <number pay periods> is expressed in months, then <each pmt> must be expressed as a monthly payment.

#### <present value>

Numeric expression that specifies the value today of a future payment, or of a stream of payments.

For example, if you put \$23.94 in the bank today and leave it there for 15 years at an interest rate of 10% compounded annually, you will end up with about \$100. So in this case, the present value of \$100 is approximately \$23.94.

#### <future value>

Numeric expression that specifies the cash balance you want after you have made your final payment.

- You set up a savings plan with a goal of having \$75,000 in 18 years to pay for your child's education. For this plan, the future value is \$75,000.
- You take out a loan for \$11,000. The future value is \$0.00, as it is for any typical loan.

#### <when due>

Numeric expression that specifies whether each payment is made at the beginning (1), or at the end (0) of each period. The default is 0 and <when due> must be 0 or 1.

#### <starting guess>

Numeric expression that specifies the value you estimate Rate will return. In most cases, this is 0.1 (10 percent).

The following example assumes you have taken out a loan for \$20,000, that you are paying off over the course of 3 years. If your payments are \$653.26 per month, and you make them at the beginning of each month, what interest rate (APR) are you paying? The answer, .115 or 11.5%, is assigned to the variable InterestRate.

Note that the return value of Rate must be multiplied by 12 to yield an annual rate.

InterestRate = Rate(3\*12, -653.26, 20000, 0, 1, .1) \* 12

#### Returns Double

An annuity is a series of cash payments, constant in value, made over a period of time. An annuity can be an investment, such as a monthly savings plan, or a loan, such as a home mortgage.

Rate calculates the interest rate on an annuity iteratively. Starting with the value of <starting guess>, it repeats the calculation until the result is accurate to within 0.00001 percent. If it cannot determine a result after 20 iterations, the function fails.

Rules:

- <number pay periods>, and <each pmt> must be expressed in terms of the same units (weekly/weeks, monthly/months, yearly/years, and so on).
- You must express cash paid out, such as deposits to savings, using negative numbers and cash received, such as dividend checks, using positive numbers.
- **Tips** Because Rate uses the order of values within the array to interpret the order of payments and receipts, be sure to enter your payment and receipt values in the correct sequence.
  - If Rate fails, try a different value for <starting guess>.
- **Example** The following example prompts the user for particulars about a loan, then returns the interest rate. To use this example, paste the first portion at or near the beginning of your Actuate Basic source code (.bas) file.

```
Declare
  Global Const ENDPERIOD = 0
  ' When payments are made
  Global Const BEGINPERIOD = 1
End Declare
Sub Start( )
  Dim Fmt As String, FutVal As Double, Guess As Double
  Dim PresVal As Double, Pymt As Double, TotalPmts As Integer
  Dim PayWhen As Integer, APR As Double
  Dim Msg As String
  Super::Start( )
  ' Define percentage format
  Fmt = "#0.00%"
  ' Usually 0 for a loan
  FutVal = 0
```

```
' Guess of 10 percent
Guess = 0.1
PresVal = 20000 ' The loan amount
Pymt = 600 ' The monthly payment
TotalPmts = 36 ' The number of monthly payments
' Assume end period payment
PayWhen = ENDPERIOD
APR = (Rate(TotalPmts, -Pymt, PresVal, FutVal, PayWhen,
+ Guess) * 12)
Msg = "Your interest rate is " & Format(APR, Fmt ) & "."
ShowFactoryStatus(Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also FV function IPmt function NPer function Pmt function PPmt function PV function

# **ReDim statement**

Changes the size of an existing dynamic array variable, and reallocates memory if needed.

Syntax ReDim [Preserve] <array variable name>(<subscripts>) [As <type>], <array variable name>(<subscripts>) [As <type>]. . .

# Parameters Preserve

Retains the data in an array when you resize the last dimension. With Preserve, if you decrease the size of any dimension, you will lose some of the data in that dimension although the data in the others will be retained.

# <array variable name>

Name of the array. Can be a variable name or expression, for example myArray or my.Array. An expression cannot contain parentheses, for example my.Array(1).

# <subscripts>

Each <subscripts> argument specifies the number of elements in a dimension. The number of <subscripts> arguments specifies the number of dimensions in the array.

Array dimensions using the following syntax:

[<lower> To]<upper>[,[<lower> To]<upper>]...

The following conditions apply to <subscripts>:

- <lower> and <upper> can range from -2,147,483,647 to 2,147,483,647 inclusive.
- <lower> must always be less than <upper>.
- Each ReDim can specify up to 60 dimensions.
- You cannot add dimensions. That is, the number of <subscripts> arguments must not exceed the number given when the array was originally declared.
- You cannot add dimensions. The number of [<lower>To] <upper> pairs must not exceed the number given when the array was originally declared.

In the following example, the statements are equivalent if you do not use Option Base:

ReDim P(7,5) ReDim P(0 TO 7, 0 TO 5) ReDim P(7, 0 TO 5)

# As <type>

Specifies a data type or class for the variable. If you specify a class, the variable can hold a reference to an instance of that class or descendant classes. You must use As <type> to declare the variable that you use to specify the dimensions of an array. If you do not specify a data type in Actuate Basic, the data type for a variable defaults to the Variant data type, which cannot be used as a parameter for sizing an array with the ReDim statement.

Rules:

- Declare an array with the Dim statement before using ReDim.
- Use ReDim only at the procedure level.
- Use ReDim to change the number of elements in an array, not the number of dimensions.
- You cannot use ReDim to resize a fixed-size array.
- You cannot change the data type of an array.
- The Erase statement recovers the memory previously allocated to a dynamic array.
- **Tips** To work with large arrays, use ReDim, which dynamically allocates memory as needed.
  - To clear the values in an array, use ReDim without Preserve.

• To avoid debugging problems, note that variables declared with Dim are automatically initialized as shown in Table 6-42.

 Table 6-42
 Initialization of the variables that are declared with Dim

Туре	Initialized As
Numeric	0
Variant	Empty
Variable-length strings	Empty string("")
User-defined type	Separate variables
Object or class	Nothing (before Set)
CPointer	Null

**Example** The following example dynamically resizes an array while a procedure is running:

```
Sub Start()
  Dim Customer1 As Integer, Customer2 As Integer
  Dim I As Integer, MyArray() As Integer
  Super::Start( )
  Customer1 = 5
  Customer2 = 10
  ReDim MyArray (Customer1)
  For I = 1 To Customer1
     MyArray(I) = I^2
     ShowFactoryStatus( I & ": " & MyArray(I) )
  Next I
  ReDim MyArray(Customer2)
  For I = 1 To Customer2
     MyArray(I) = I^2
     ShowFactoryStatus( I & ": " & MyArray(I) )
  Next I
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Dim statement Erase statement

# **Rem statement**

Inserts comments or explanatory remarks in a program.

Syntax {Rem | :Rem | ' }<comment>

#### Parameters <comment>

Any characters or numbers, including spaces or punctuation. <comment> is a remark that explains what is happening in the program.

**Description** • All text to the right of Rem is ignored by the compiler.

- <comment> appears exactly as you entered it when the program is listed.
- If you branch from a GoTo statement to a line that contains Rem, program execution continues with the first executable statement that follows Rem.

To place a comment on the same line as a code instruction, precede the comment either with the apostrophe identifier or with :Rem.

For A = 1 To 20 'This code steps through the 20 items For A = 1 To 20 : REM This code steps through the 20 items

To place a comment on a line by itself, precede the comment with any one of the three identifiers.

```
Rem The 3rd line after this steps through 20 items of an array.
: Rem The 2nd line after this steps through 20 items of array.
' The next line steps through the 20 items of the array.
For A = 1 To 20
```

- Tips
- Use comments liberally throughout your program to explain how the code works.
  - To disable a line of code when debugging, precede it with one of the three Rem identifiers. Temporarily commenting out the line, lets you see what happens when that line is left out of the program.

**Example** The following example shows the various ways of using Rem:

```
Sub Start( )
Dim Message1, Message2, Message3, AllMessages
Super::Start( )
Rem The compiler will ignore this entire line.
' This line will also be ignored.
Message1 = "First message line." :Rem Colon before REM here
Message2 = " Second message line." ' This one needs no colon
: Rem This line has a colon, but really does not need it.
Message3 = " This example shows different ways to use REM."
```

```
AllMessages = Message1 & Message2 & Message3
ShowFactoryStatus( AllMessages )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Format, Format\$ functions Str, Str\$ functions

# **Reset statement**

Writes all data in open file buffers to the appropriate disk files, then closes all open disk files.

Syntax Reset

**Description** Performs the same task as Close with no parameters.

**Example** The following example creates and opens two test files, closes them using Reset, then reopens them and reads back the data. Finally, it deletes the test files.

```
Sub Start()
  Dim FirstVar, SecondVar
  Super::Start( )
  Open "testalpha.fil" for Output As #1
  Open "testbeta.fil" for Output As #2
  Print #1, "This line is in the first test file."
  Print #2, "This line is in the second test file."
  Reset
  Open "testalpha.fil" for Input As #1
  Open "testbeta.fil" for Input As #2
  Line Input #1, FirstVar
  Line Input #2, SecondVar
  Reset
  ShowFactoryStatus ( FirstVar & " " & SecondVar )
  Reset
  ShowFactoryStatus( "The test files will now be deleted." )
  Kill "testalpha.fil"
  Kill "testbeta.fil"
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Close statement End statement

# **Resume statement**

Releases control from an error-handling routine and resumes program execution at a specified place.

Syntax Resume { [0] | Next | <line> | <line label> }

# Parameters [0]

Keyword (an optional zero) indicating that Actuate Basic is to resume program execution with the statement that caused the error, or with the statement that last called out of the procedure containing the error handler.

For example, the following two statements are equivalent:

Resume 0

# Next

Keyword indicating that Actuate Basic is to resume program execution with the statement following the one that caused the error, or with the statement following the one that last called out of the procedure containing the error handler. For example:

Resume Next

### <line> or <line label>

Keyword indicating the line number or line label at which Actuate Basic is to resume execution. This must be in the same procedure as the error handler. For example, the following example instructs Actuate Basic to continue execution at the label Continuation:

Resume Continuation

**Description** Where Actuate Basic resumes execution depends on the location of the error handler in the which the error is trapped, not necessarily on the location of the error itself. Table 6-43 summarizes where a program resumes execution when either the Resume [0] or Resume Next statement is used.

Error occurred	Resume [0]	Resume Next
In the same procedure as the error handler.	The statement that caused the error.	The statement after the one that caused the error.
In a different procedure from the error handler.	The statement that last called out of the procedure containing the error handler.	The statement after the one that last called out of the procedure containing the error handler.

 Table 6-43
 Where a program resumes when using Resume [0] or Resume Next

- **Rules** You can only use Resume in an error-handling routine.
  - You can only exit an error-handling routine by using Resume, not by using End Sub or End Function.
- **Example** The following example generates two errors in succession. In the first case, the error routine displays the error number and description, then uses Resume to resume execution of the code at the line following the line that caused the error. In the second case, the error routine displays a simple message and uses Resume to branch to a specified routine. This branch causes the flow of program execution to skip the line following the one that caused the error.

```
Sub Start()
  Super::Start( )
  On Error GoTo ErrorHandler
  Dim Zilch, Zero, DivisionByZero
  Zilch = 0
  Zero = 0
  DivisionByZero = Zilch/Zero
  ShowFactoryStatus("The line after the divide-by-zero error.")
  Open "z:\zzzzzz.zzz" for Input As #1
  ShowFactoryStatus( "Theis line is after the error. Skipped." )
Badfile:
  ShowFactoryStatus ( "This line is in the Badfile routine." )
  Exit Sub
ErrorHandler:
  If Err = 8 Then
     ShowFactoryStatus ( "Error: " & Err & " occurred: " &
       Error$(Err) )
+
     Resume Next
  Else
     ShowFactoryStatus( "File error. Going to badfile label." )
     Resume Badfile
  End If
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also On Error statement

# **RevInStr function**

Returns the starting position of the last occurrence of one string within another. RevInStr is used to extract a filename like Myhouse.txt from a long or fully qualified string like C:/Country/State/City/Street/Myhouse.txt.

Syntax RevInStr(<string being searched>, <string to find>)

RevInStr(<end>, <string being searched>, <string to find>)

**Description** RevInStr is like InStr, except that where InStr returns the position of the first character of the first occurrence of a string, RevInStr returns the position of the first character of the last occurrence of a string.

In cases where <string to find> occurs only once within <string being searched>, first and last occurrences coincide, and both InStr and RevInStr return the same value.

In the following examples, InStr and RevInStr return different values:

In the following examples, both InStr and RevInStr return the same value:

### Parameters <string being searched>

String you are inspecting in order to locate the last occurrence of <string to find>.

<string being searched> must be a variable string, a literal string, a string constant, the return value of any function that returns a string, or a variant that can evaluate to a string.

#### <string to find>

String in which last occurrence you are seeking is within <string being searched>.

<string to find> must be a variable string, a literal string, a string constant, the return value of any function that returns a string, or a variant that can evaluate to a string.

#### <end> (optional)

The position within <string being searched> at which to end searching. The first character occupies position 1.

- If you supply <end>, Actuate Basic searches for the last occurrence of <string to find> that appears before the character at position <end>.
- If you do not supply <end>, Actuate Basic searches to the last character of <string being searched>.
- Must be a number or numeric expression.
- Must be between 1 and 2,147,483,647, inclusive.

#### Returns Integer

 If <string to find> is found within <string being searched>, RevInStr returns the position of the first character at which the last match was found.

- If <string to find> is not found within <string being searched>, RevInStr returns 0.
- If <string being searched> is zero-length, RevInStr returns 0.
- If <end> is greater than the length of <string being searched>, RevInStr returns 0.
- **Example** The following example shows how to use RevInStr to find the filename portion of a fully qualified path that contains slashes.

Because RevInStr returns the position of the last occurrence of a slash, the following code displays Johnsmith.fil regardless of the number of slashes in the file's full name or path:

```
Sub Start( )
Dim baseName, fileName As String
Dim posn As Integer
Super::Start( )
fileName =
+  "C:/USA/Massachusetts/Springfield/Zip01109/Johnsmith.fil"
posn = RevInStr( fileName, "/" )
If posn = 0 Then
baseName = fileName
Else
baseName = Mid$( fileName, posn + 1 )
End If
ShowFactoryStatus( baseName )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Format, Format\$ functions Left, Left\$ functions Len function Mid, Mid\$ functions Right, Right\$ functions

# **RGB** function

Returns a number representing the RGB color value for a given mix of red, green and blue values.

**Syntax** RGB(<red>, <green>, <blue>)

# Parameters <red>

Integer indicating the red value of the color.

■ If <red> is greater than 255, <red> defaults to 255.

- If <red> is empty, <red> defaults to 0.
- <red> must be greater than 0.
- <red> must not be Null.

## <green>

Integer indicating the green value of the color.

- If <green> is greater than 255, <green> defaults to 255.
- If <green> is empty, <green> defaults to 0.
- <green> must be greater than 0.
- <green> must not be Null.

# <blue>

Integer indicating the blue value of the color.

- If <blue> is greater than 255, <blue> defaults to 255.
- If <blue> is empty, <blue> defaults to 0.
- <blue> must be greater than 0.
- <blue> must not be Null.

# Returns Long

- Returns a number between 0 and 16,777,215, inclusive. This single number represents the relative intensities of red, green, and blue that make up a particular color.
- Table 6-44 shows some standard colors. For each color, you see the decimal values returned by RGB, the hexadecimal equivalent, the corresponding QBColor number, and the mix of red, green and blue decimal values that make up the color.

					-	
Color	RGB Value (Dec)	RGB Value (Hex)	QBColor	Red Value	Green Value	Blue Value
Black	0	&H00	0	0	0	0
Blue	16711680	&HFF0000	1	0	0	255
Green	65280	&HFF00	2	0	255	0
Cyan	16776960	&HFFFF00	3	0	255	255
Red	255	&HFF	4	255	0	0
Magenta	16711935	&HFF00FF	5	255	0	255
Yellow	65535	&HFFFF	6	255	255	0
					(с	ontinues)

**Table 6-44**Colors and the corresponding RGB values

Color	RGB Value (Dec)	RGB Value (Hex)	QBColor	Red Value	Green Value	Blue Value
White	16777215	&HFFFFFF	15	255	255	255
Gray	8421504	&H808080	8	128	128	128

 Table 6-44
 Colors and the corresponding RGB values (continued)

**Tips** There are four ways in which you can precisely represent a color in Actuate Basic:

- Specify the relative mix of red, green and blue in a color by using RGB with decimal numbers, as described here.
- Directly specify, without using RGB or any function, the relative mix using hexadecimal numbers in a certain format.
- Use one of the 16 standard Windows colors by supplying the appropriate integer to QBColor.
- Use one of the built-in color constant names, such as BLUE.

# **Example** The following example returns the RGB value for a random mix of red, green, and blue values:

```
Sub Start()
  Dim RedPart As Integer, GreenPart As Integer
  Dim BluePart As Integer
  Dim HexVersion, Msg As String, RGBNumber
  Super::Start( )
  RedPart = CInt(Rnd * 255)
  GreenPart = CInt( Rnd * 255 )
  BluePart = CInt( Rnd * 255 )
  RGBNumber = RGB( RedPart, GreenPart, BluePart )
  HexVersion = Hex( RGBNumber )
  Msg = "Red component: " & RedPart
  & " Green component: " & GreenPart
+
    & " Blue component: " & BluePart
+
    & " The decimal representation of that mix of "
+
   & "red, green, and blue is: "
+
   & RGBNumber & ". "
+
     & "The hexadecimal representation of it is: "
     & HexVersion
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also QBColor function

# **Right, Right\$ functions**

Returns a segment of a Variant or String, starting at the last character and working toward the beginning.

**Syntax** Right(<string exprs>, <length>)

Right\$(<string exprs>, <length>)

# Parameters <string exprs>

Source string from which you are copying the last portion. Can be a variable string, a literal string, a string constant, the return value of any function that returns a string, or a Variant that can evaluate to a String.

#### <length>

Long that specifies how many characters to copy from the right of <string exprs>.

- <length> must be an Integer or expression between 0 and 2,147,483,647.
- In the following example, the statements are equivalent. Both return Widget:

```
Right$("Widget",6)
Right$("Widget",99)
```

# Returns Right: Variant

Right\$: String

- If <length> = 0, returns zero-length string.
- If <length> is greater than or equal to the length of <string exprs>, returns an exact copy of <string exprs>.
- If any parameter evaluates to Null, Right[\$] returns Null.
- **Tip** Use Len to find the number of characters in <string exprs>.
  - Use InStr to find the position of a specified character in <string exprs>.
- **Example** The following example parses a string for a customer's first and last names:

```
Sub Start( )
Dim FName As String, Msg As String, LName As String
Dim SpacePos As Integer, Customer As String
Super::Start( )
Customer = "Manuel Barajas"
' Find the space
```

```
SpacePos = InStr(1, Customer, " ")
  If SpacePos Then
     ' Get first and last name
     FName = Left$(Customer, SpacePos - 1)
     LName = Right$(Customer, Len(Customer) - SpacePos)
     Msg = "The first name is """ & FName & "." & """"
      & " The last name is """
+
       & LName & "." & """"
+
  Else
     Msq = Customer & " does not have a first and last name!"
  End If
  ' Display the message
ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Format, Format\$ functions Left, Left\$ functions Len function Mid, Mid\$ functions

# **RightB, RightB\$ functions**

Returns a segment of a Variant or String, starting at the last byte and working toward the beginning.

These functions are provided for backward compatibility with previous Actuate releases. Actuate uses UCS-2 as internal encoding, therefore passing strings consisting of characters from different code pages might produce unexpected results. For this reason, Actuate recommends you use the Right and Right\$ functions instead of the RightB and Right\$ functions.

Syntax RightB(<string exprs>, <length>)

RightB\$(<string exprs>, <length>)

#### Parameters <string exprs>

Source string from which you are copying the last portion. It can be a variable string, a literal string, a string constant, the return value of any function that returns a string, or a Variant that can evaluate to a String.

#### <length>

Long that specifies how many bytes to copy from the right of <string exprs>.

Rule: Must be an Integer or expression between 0 and 2,147,483,647.

**Example** The following statements are equivalent. Both return Widget:

RightB\$("Widget",6) RightB\$("Widget",99)

Returns RightB: Variant RightB\$: String

- If <length> = 0, the function returns zero-length string.
- If <length> is greater than or equal to the length of <string exprs>, the function returns an exact copy of <string exprs>.
- If any parameter evaluates to Null, RightB[\$] returns Null.
- **Tips** Use LenB to find the number of bytes in <string exprs>.
  - Use InStrB to find the position of a specified bytes in <string exprs>.

# See also InStrB function

LeftB, LeftB\$ functions LenB function MidB, MidB\$ functions Right, Right\$ functions

# **RmDir statement**

Removes a subdirectory from a disk.

Syntax RmDir <directory name>

#### Parameters <directory name>

String expression that is the name of the directory or subdirectory to be deleted. The default is the current drive and directory.

The following conditions apply to <directory name>:

- Must specify a valid directory.
- Target directory must contain no child subdirectories.
- Target directory must be empty.

<directory name> has the following syntax.

[<drive:>] [ \ ]<directory>[\<directory>]...(Windows)

```
[/]<directory>[/<directory>]...(UNIX)
```

#### <drive:>

Character, followed by a colon, that specifies the drive (Windows only).

### <directory>

String expression that is the name of the directory or subdirectory to remove.

For example, the following statement removes the subdirectory Test under the current directory on the current drive:

RmDir "Test"

The following removes the subdirectory Test under the root directory of the current drive:

```
RmDir "\Test"
```

The following removes the subdirectory Test under the root directory of drive D:

```
RmDir "D:\Test"
```

- **Description** RmDir works like the DOS command of the same name. Unlike the DOS command it cannot be abbreviated.
  - **Tips** If a directory name contains embedded spaces, you cannot use the DOS RmDir command to remove it. Instead, use RmDir.
    - To determine the current directory, use CurDir.
  - **Example** The following example determines whether a \Tmpzzz subdirectory exists on the current drive. If the subdirectory does not exists, the example creates the subdirectory, prompts the user to retain or remove the subdirectory, and responds accordingly.

```
Sub Start()
  Dim UserAns As Integer, ThisDrive As String
  Dim Msg As String, TempDir As String
  Super::Start( )
  ' Set up error handler
  On Error Resume Next
  ' Get current drive letter
  ThisDrive = Left$(CurDir, 2)
  ' Construct full path spec
  TempDir = UCase$(ThisDrive + "\Tmpzzz")
  ' Make the new directory
  MkDir TempDir
  ' Does it exist?
  If Err = 41 Then
     Msg = "Sorry, " & TempDir & " directory already exists. "
  Else
     Msg = TempDir & " directory was just created. "
     ShowFactoryStatus ( Msg )
     ShowFactoryStatus( "The directory will now be deleted." )
     RmDir TempDir
```
End If End Sub

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also CurDir, CurDir\$ functions MkDir statement

### **Rnd function**

Returns a random number.

Rnd[(<sequencing code>)] Syntax

#### Parameters <sequencing code>

Numeric expression for the code number that identifies the value to return from the pseudo-random sequence. The pseudo-random sequence is the fixed sequence of numbers you generate if you have not used Randomize and you call Rnd repeatedly without any argument.

If <sequencing code> is omitted, Actuate Basic moves to the next number in the pseudo-random sequence.

Table 6-45 shows the possible values of <sequencing code> and the corresponding values Rnd returns from the pseudo-random sequence.

that Rnd returns		
<sequencing code=""></sequencing>	Rnd returns	
Less than 0	The same number every time. The specific value varies as a function of <sequencing code="">.</sequencing>	
Greater than 0	The next random number in the pseudo-random sequence.	
Equal to 0	The number most recently generated.	

Table 6-45 Possible values of <sequencing code> and the corresponding values

For example, the following expressions are all true, but only if you evaluate them in the order shown and do not use a Randomize statement before them.

If you use a Randomize statement, the last of these expressions is always different.

```
Rnd(-5) = 0.298564536496997
Rnd(0) = .298564536496997
Rnd(1) = .223996873013675
Rnd = .0928899832069874
```

- **Returns** Single
  - Rnd returns a value less than 1 but greater than or equal to 0.
  - Unless you use Randomize beforehand, Rnd generates the same randomnumber sequence every time the program is run.
  - As long as <sequencing code> is omitted or is greater than zero, each successive call to Rnd uses the previous random number as the seed for the next one.
  - **Tips** To generate a more apparently random sequence every time the program is run, use Randomize without an argument before you use Rnd.
    - To produce random integers in a given range, use the following formula:

Int((<high number> - <low number> + 1) \* Rnd + <low number>)

In this formula, <high number> represents the highest number in the range, and <low number> the lowest.

**Example** The following example simulates rolling a die twice by generating random values between 1 and 6 for each roll:

```
Sub Start()
  Dim DieRoll1, DieRoll2, UserMsg
  Super::Start( )
  ' Seed random num generator
  Randomize
  ' First roll
  DieRoll1 = Int(6 * Rnd + 1)
  ' Second roll
  DieRoll2 = Int(6 * Rnd + 1)
  UserMsg = "You rolled a " & DieRoll1
  & " and a " & DieRoll2
+
     & " giving a total of "
+
     & (DieRoll1 + DieRoll2) & "."
+
  ShowFactoryStatus ( UserMsg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Randomize statement Time, Time\$ functions

### **RSet statement**

Right-aligns a string within the space of a string variable.

Syntax RSet <string variable> = <string exprs>

#### Parameters <string variable>

Name of a string variable in which RSet stores the right-aligned <string exprs>.

#### <string exprs>

String expression that you want RSet to right-align within <string variable>.

Table 6-46 shows the settings for <string exprs> and the corresponding RSet behaviors.

<b>1 able 6-46</b> Settings for string expressions and corresponding RSet bena	avioi
--	-------

<string exprs=""></string>	Behavior of RSet
Shorter than <string variable=""></string>	Right-aligns <string exprs=""> within <string variable="">. Replaces any leftover characters in <string variable=""> with spaces.</string></string></string>
Longer than <string variable=""></string>	Places only the right most characters, up to the length of the <string variable="">, in <string variable="">. Truncates characters beyond the length of <string variable=""> from the left.</string></string></string>

**Example** The following example right-aligns text within a 20-character String variable:

```
Sub Start()
  Dim Msg, TmpStr As String
  Super::Start( )
  ' Create 20-character string
  TmpStr = String(20, "*")
  Msg = "The following two strings that have been right"
     & "and left justified in a " & Len( TmpStr )
+
     & "-character string."
+
  ShowFactoryStatus ( TmpStr )
  ' Right justify
  RSet TmpStr = "Right->"
  ShowFactoryStatus ( TmpStr )
  ' Left justify
  LSet TmpStr = "<-Left"
  ShowFactoryStatus ( TmpStr )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

### See also LSet statement

### **RTrim, RTrim\$ functions**

Returns a copy of a string after removing the trailing spaces.

```
Syntax RTrim(<string exprs>)
```

### RTrim\$(<string exprs>)

### Parameters <string exprs>

String expression from which RTrim\$ strips trailing spaces. Trailing spaces are any spaces that occur after the last non-space character in a string. The <string exprs> must be a variable string, a literal string, a string constant, the return value of any function that returns a string, or a Variant that can evaluate to a String.

Returns RTrim: Variant RTrim\$: String

- If there are no trailing spaces, RTrim[\$] returns the original <string exprs>.
- If <string exprs> evaluates to Null, RTrim[\$] returns Null.
- **Tips** To simultaneously strip both leading and trailing spaces in a string, use Trim[\$].
  - To find other spaces in the middle of a string, use InStr.

**Example** The following example strips trailing spaces from a string variable, while LTrim[\$] strips leading spaces:

```
Sub Start()
  Dim Msq As String, NL As String
  Dim CustName As String, CustName1 As String
  Super::Start( )
  CustName = " Harold Pinter "
  ' Strip left and right spaces
  CustName1 = LTRIM$(RTRIM$(CustName))
  Msg = "The original client name " & "'" & CustName
     & "' was " & Len(CustName) & " characters long. "
+
     & "There were two leading spaces "
+
   & "and two trailing spaces. "
+
   & "The name returned after stripping the spaces "
+
     & "is: " & "'" & CustName1 & "'"
+
     & "...and it contains only "
+
     & Len(CustName1) & " characters."
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Format, Format\$ functions LTrim, LTrim\$ functions Trim, Trim\$ functions

### SafeDivide function

Divides two given numbers, but prevents division by zero. Returns a specified value if division by zero might otherwise have occurred.

Syntax SafeDivide(<num> As Variant, <denom> As Variant, <ifZero> As Variant) As Variant

### Parameters <num>

Any number, or valid numeric expression. The numerator of a fraction or ratio.

### <denom>

Any number, or valid numeric expression. The denominator of a fraction or ratio.

### <ifZero>

Value to return when <denom> is zero.

Returns Variant

If <denom> = 0, SafeDivide returns <ifZero>. Otherwise, SafeDivide returns <num>/<denom>.

**Example** The following example averages amounts in a number of records. If no records exist, the example does not generate a division-by-zero error and returns zero.

AverageAmtVar = SafeDivide(Sum(Amount)), Count(), 0)

For information about using the code examples, see "Using the code examples," earlier in this chapter.

### See also Err function Error, Error\$ functions On Error statement

## Second function

Returns an Integer from 0 to 59, inclusive, that represents the second of the minute specified by a date expression.

Syntax Second(<date expression>)

### Parameters <date expression>

Date expression, or any numeric or string expression that can be interpreted as a date, a time, or both a date and a time:

Can be a string such as November 12, 1982 8:30 PM, Nov 12, 1982 08:30 PM, 11/12/82 8:30pm, or 08:30pm, or any other string that can be interpreted as a date, a time, or both a date and a time in the valid range.

- Can be a date serial number such as 30267.854, which corresponds to November 12, 1982 at 8:30 P.M., or any other number that can be mapped to a date, a time, or both a date and a time in the valid range.
- For date serial numbers, the integer component represents the date itself while the decimal component represents the time of day on that date, where January 1, 1900, at noon has the date serial number 2.5, and negative numbers represent dates prior to December 30, 1899 (0).

The default if no time specified is 0.

The following conditions apply to <date expression>:

- If <date expression> includes a date, it must be a valid date, even though Second does not return a date. A valid date is any date in the range January 1, 100, through December 31, 9999, expressed in one of the standard date formats.
- If <date expression> includes a time, it must be in the range 00:00:00 (12:00:00 A.M.) through 23:59:59 (11:59:59 P.M.), in either 12- or 24-hour format.
- If <date expression> is a numeric expression, it must be in the range -657434.0 to +2958465.9999, inclusive.
- If <date expression> is a variable containing a date serial number, the variable must have been explicitly declared as one of the numeric types.
- <date expression> is parsed according to the formatting rules of the current run-time locale.

For example, the following statements are equivalent. Each assigns 22 to the variable UserSecond.

```
UserSecond = Second("6/7/64 2:35:22pm")
UserSecond = Second("5:35:22 pm")
UserSecond = Second("June 7, 1964 2:35:22 PM")
UserSecond = Second("Jun 7, 1964") + 22
UserSecond = Second(23535.60789)
UserSecond = Second(0.60789)
```

### Returns Integer

 If <date expression> cannot be evaluated to a date, Second returns Null. For example:

Second("This is not a date.") returns Null

If <date expression> fails to include all date components (day, month, and year), Second returns Null. For example:

Second("Nov 12, 1982 7:11:22 AM") returns 22, but Second("Nov, 1982 7:11:22 AM") returns Null

■ If <date expression> is Null, Second returns Null.

The following assumes that the AC\_CENTURY\_BREAK property is set to the default value of 30. For information about using AC\_CENTURY\_BREAK, please see *Accessing Data using e.Report Designer Professional*:

- For abridged date expressions ending in 00 through 29 that do not specify the century, relevant date functions estimate the unabridged year by adding 2000. For example, while Year(#4/30/1910#) returns 1910, the abridged expression Year(#4/30/10#) returns 2010 (10 + 2000).
- For abridged date expressions ending in 30 through 99 that do not specify the century, relevant date functions estimate the unabridged year by adding 1900. For example, while Year(#11/12/2082#) returns 2082, the abridged expression Year(#11/12/82#) returns 1982 (82 + 1900).
- Tip If you use a date expression in one locale, it might be misinterpreted in another locale. For instance, in the United States, "1/2/2005" means January 2nd, 2005; but in France, it means February 1st, 2005. To avoid such ambiguities, use DateSerial to specify all your dates.
- **Example** The following example displays the number of hours, minutes, and seconds remaining until midnight:

```
Sub Start()
  Dim HrDiff As Integer, MinDiff As Integer, SecDiff As Integer
  Dim RightNow As Double, Midnight As Double
  Dim TotalDiff As Double, TotalMinDiff As Double
  Dim TotalSecDiff As Double, Msg As String
  Super::Start( )
  Midnight = TimeValue( "23:59:59" )
  ' Get current time
  RightNow = Now
  ' Get diffs from midnight
  HrDiff = Hour( Midnight ) - Hour( RightNow )
  MinDiff = Minute( Midnight ) - Minute( RightNow )
  SecDiff = Second( Midnight ) - Second( RightNow ) + 1
  ' Restate seconds and minutes if necessary
  If SecDiff = 60 Then
     ' Add 1 to minute
     MinDiff = MinDiff + 1
     ' And set 0 seconds
     SecDiff = 0
  End If
  If MinDiff = 60 Then
     ' Add 1 to hour
     HrDiff = HrDiff + 1
     ' And set 0 minutes
     MinDiff = 0
```

```
End If
  ' Now get totals
  TotalMinDiff = ( HrDiff * 60 ) + MinDiff
  TotalSecDiff = ( TotalMinDiff * 60 ) + SecDiff
  TotalDiff = TimeSerial( HrDiff, MinDiff, SecDiff )
  ' Prepare msq for display
  Msg = "There are a total of "
     & Format( TotalSecDiff, "#, ##0" )
+
     & " seconds until midnight. That translates to "
+
   & HrDiff & " hours, "
+
     & MinDiff & " minutes, and "
+
    & SecDiff & " seconds. "
+
     & "In standard time notation, it becomes "
+
  ' Remember not to use "mm" for minutes! m is for month.
  Msg = Msg & Format(TotalDiff, "hh:nn:ss") & "."
  ShowFactoryStatus ( Msg )
End Sub
```

See also Day function Hour function Minute function Month function Now function Time, Time\$ functions Weekday function Year function

### Seek statement

Sets the position in a file for the next read or write. The Seek statement should not be confused with the Seek2 function.

Syntax Seek #<open file number>, <next position>

### Parameters <open file number>

Numeric expression is the file number for a file that is Open.

- Must be the number of a currently open file.
- The pound sign (#) preceding the file number is required.

### <next position>

Numeric expression that specifies the number that indicates where in <open file number> the next read or write should occur. <next position> must be in the range 1 to 2,147,483,647, inclusive.

<next position> means, or refers to, different things, depending on the mode under which the file corresponding to <open file number> is open. Table 6-47 summarizes these different meanings.

•	·
Mode	<next position=""> refers to</next>
Random	The number of a record in the open file. The first record is record 1.
Binary Input Output Append	The byte position relative to the beginning of the file. The first byte is byte 1.

 Table 6-47
 Interpretations of <next position> in different modes

For example, in the following code fragment, we open a file called Test.dat for random access and use Get to read record number 52 from it; this action positions the pointer at record 53. Then we use the Seek2 function to save this new position to the variable CurrentRec. Finally, we use the Seek statement to reposition the file pointer to record number 100.

```
Open "Test.dat" For Random As #1 Len = 32
Get #1, 52, TestRecord
CurrentRec = Seek2(1)
Seek #1, 100
```

- **Rule** If <next position> is beyond the end of a file, Seek extends the file.
  - Record numbers in Get and Put override Seek file positioning.
- **Example** The following example defines three customer names. It writes each name to a test file and then reads the names back.

```
Sub Start( )
Dim CustName As String
Dim I As Integer, Max As Integer, Msg As String
Super::Start( )
' Create a sample data file
Open "Testfile.dat" For Random As #1 Len = 50
' Put records into file on disk
Put #1, 1, "Customer 1"
Put #1, 2, "Customer Two"
Put #1, 3, "Third Customer"
Close #1
```

```
' Close the file and open again for reading
  Open "Testfile.dat" For Random As #1 Len = 50
  ' Calc total # of records
  Max = LOF(1) \setminus 50 + 1
  ' Read file backwards
  For I = Max To 1 Step -1
     ' Seek statement used
     Seek #1, I
     ' Get record at that position
     Get #1, , CustName
     Msg = "Record #" & (Seek2(1) - 1) & " contains: "
     Msq = Msq & CustName
     ShowFactoryStatus ( Msg )
  Next I
  ' Close test file
  Close #1
  Msg = "The test file will now be deleted."
  ShowFactoryStatus ( Msg )
  ' Delete file from disk
  Kill "Testfile.dat"
End Sub
```

See also Get statement Open statement Put statement Seek2 function

## Seek2 function

Returns the current position in an open file. This function should not be confused with the Seek statement.

Syntax Seek2(<open file number>)

### Parameters <open file number>

Numeric expression that represents the file number of a file that is Open. This must be the number of a currently open file.

For example, in the following code fragments, we open a file called Test.dat for random access and use Get to read record number 52 from it; this action positions the pointer at record 53. Then we use the Seek2 function to save this new position to the variable CurrentRec. Finally, we use the Seek statement to reposition the file pointer to record number 100.

```
Open "Test.dat" For Random As #1 Len = 32
Get #1, 52, TestRecord
CurrentRec = Seek2(1)
Seek #1, 100
```

### Returns Integer

- Returns a value between 1 and 2,147,483,647.
- The value Seek2 returns means, or refers to, different things, depending on the mode under which the file corresponding to <open file number> is open. Table 6-48 summarizes these different meanings.

ModeSeek2 refers to...RandomThe number of the next record to be read or written to.BinaryThe byte position at which the next operation is to takeInputplace.OutputAppend

**Table 6-48** Interpretations of Seek2 in different modes

**Example** The following example prompts the user for three customer names.

After writing the names to a test file, the Seek statement repositions the pointer three times successively, then displays the names in reverse order. The Seek2 function identifies the record numbers.

```
Sub Start()
Dim CustName As String
Dim I As Integer, Max As Integer, Msg As String
Super::Start( )
' Create a sample data file
Open "Testfile.dat" For Random As #1 Len = 50
' Put records into file on disk
Put #1, 1, "Customer 1"
Put #1, 2, "Customer Two"
Put #1, 3, "Third Customer"
Close #1
' Close the file and open again for reading
Open "Testfile.dat" For Random As #1 Len = 50
' Calc total # of records
Max = LOF(1) \setminus 50 + 1
' Read file backwards
For I = Max To 1 Step -1
  ' Seek statement used
  Seek #1, I
   ' Get record at that position
  Get #1, , CustName
```

```
Msg = "Record #" & (Seek2(1) - 1) & " contains: "
Msg = Msg & CustName
ShowFactoryStatus( Msg )
Next I
' Close test file
Close #1
Msg = "The test file will now be deleted."
ShowFactoryStatus( Msg )
' Delete file from disk
Kill "Testfile.dat"
End Sub
```

See also Get statement

Open statement Put statement Seek statement

### Select Case statement

Executes the first of several blocks of instructions for which an associated expression matches a given one.

Syntax Select Case <compare to this>

[Case <value to compare> [, <value to compare>]...

[<statement block 1>]]

[Case <value to compare> [, <value to compare>]...

[<statement block 2>]]

...

[Case Else

[<statement block n>]]

End Select

**Description** For the purposes of this discussion, one expression is said to match a second expression when they are equivalent, or when the first falls within a range specified by the second.

Actuate Basic compares <compare to this> against each <value to compare> until it finds a match. For the first—and only the first—of these matches, if there is one, Actuate Basic:

- Executes the instructions in the <statement block> that immediately follows the matching <value to compare>.
- Skips any other intervening lines.
- Passes control to the statement that follows End Select.

If no <value to compare> at all matches <compare to this>, Actuate Basic looks for a Case Else clause. If it finds one, it:

- Executes any statements that immediately follow it.
- Passes control to the statement immediately following End Select.

If it does not find a Case Else clause, it passes control to the statement immediately following End Select.

### Parameters <compare to this>

A valid expression of any data type that can be evaluated. <compare to this> is compared to <value to compare> and that determines whether to execute the associated case.

### Case

Keyword that begins a statement block of a Select Case statement. Case is always followed by a value to compare, and a statement to execute if <value to compare> is the same as <compare to this>. Case may be repeated.

#### <value to compare>

A valid expression of any data type. In addition to the form previously shown in the Syntax section, <value to compare> can also take either of the following forms, or it can take any combination of all three forms.

### **Range Form**

<lower value > To <higher value >

For <value to compare>, specifies a range of values between <lower value> and <higher value>, inclusively. If <compare to this> falls within the range, the case evaluates as True.

### То

Keyword used to specify a range of values.

### <lower value>, <higher value>

Numeric or string expressions.

The lower value must always precede the higher one. For example:

Case 65, 19 To 25, 34 To 49 Case "ZACHARY", "ADAMS" To "JEFFERSON"

### **Comparison Operator Form**

[Is] <comparison operator> <value to compare>

ls

An optional keyword preceding <comparison operator> <value to compare>.

#### <comparison operator>

Any valid comparison operator except Like. For example:

Case Is <= 25 , 34 To 49, Is = 65

#### Case Else

Keyword indicating the statement block Actuate Basic executes if it finds no match between <compare to this> and any of the <value to compare> expressions. If there is more than one Case Else clause, only the first is executed.

- **Tips** To handle any unforeseen <compare to this> values, always use a Case Else clause in your Select Case statements.
  - The Is comparison operator is different from the Is keyword in this statement.
  - To be sure that string data the user supplies always matches <value to compare> strings consistently, use UCase\$ on the user data before you use Select Case.
  - To specify whether string comparisons are or are not case-sensitive, use Option Compare.
  - It is good programming practice to evaluate Boolean variables by using the keywords True or False instead of by inspecting their content for a nonzero (True) or zero (False) numeric value.
- **Example** The following example uses logical operators with a Select Case statement:

```
Sub Start( )
Dim IVal As Integer, Tester As Integer, Msg As String
Super::Start( )
IVal = CInt (Rnd * 200)
'Set Tester to True, just to get started
Tester = 1
Select Case IVal
Case 100 To 199
Msg = IVal & " is between 100 and 199"
Case 10, 20, 30, 40, 50, 60, 70, 80, 90
Msg = IVal & " is divisible by 10."
Case Else
'The following nested Select Case structure helps sort out
'numeric input in cases where the user typed a number
```

```
Select Case IVal
Case 1 To 9
Msg = IVal & " is a number from 1 to 9."
Case Is < 100
Msg = IVal & " is a number greater than 9."
Case Else
Msg = IVal & " is either 0 or 200"
End Select
End Select
ShowFactoryStatus( Msg )
End Sub
```

See also GoTo statement If...Then...Else statement Option Compare statement

### Set statement

Assigns an object reference to a variable.

Syntax Set <object variable> = {<object reference> | New <class> | Nothing}

**Description** When you use Set to assign an object reference to a variable, Actuate Basic generally creates not a copy of the object but a reference to it. More than one object variable can refer to the same object. Because these variables are references and not copies, as the object changes, so do they.

When you use the New reserved word with Set, you create a new instance of the referenced type.

### **Parameters <object variable>** Expression that specifies the name of the variable to which you are assigning an object reference.

- Must be of a class consistent with that of <object reference>.
- Must have been explicitly declared unless declared for you by Actuate Basic.

#### <object reference>

Expression that specifies the name of an object, of another declared variable of the same class, or of a function or method that returns an object.

#### New

Reserved word used to create a new instance of a specific class. New cannot be used to create new variables of any of the fundamental data types.

### <class>

The name of a specific class.

### Nothing

Reserved word that discontinues association of <object variable> with any specific object. Assigning Nothing to <object variable> releases all resources associated with the previously referenced object, as long as no other variable refers to it.

**Example** The following example uses the Set statement with New to create a new instance of AcLabelControl:

```
Sub Start( )
Dim MyLabel As AcLabelControl, Msg As String
Super::Start( )
Set MyLabel = New AcLabelControl
If IsKindOf(MyLabel, "AcLabelControl") Then
Msg = "Yes, MyLabel is an AcLabelControl. "
+ & "We will now set its background color to red."
ShowFactoryStatus( Msg )
MyLabel.BackgroundColor = Red
End If
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Dim statement NewInstance function NewPersistentInstance function ReDim statement Static statement

### SetAttr statement

Sets attribute information for files.

- **Syntax** SetAttr(<file name>, <attributes>)
- **Description** The operating system assigns one or more attributes to each file. These attributes indicate whether the file is a normal, read-only, hidden, or system file; whether it has been modified since the last backup; or whether some combination of these is the case. SetAttr lets you modify these attributes.

# Parameters <file name> String expression that specifies the file for which to set one or more attributes. The default path is the current drive and current directory. <file name> cannot include wildcard characters and must refer to a valid file.

<file name> can be open for read-only access, but must not be opened for write access.

#### <attributes>

Numeric expression that specifies the sum of one or more of the values in Table 6-49, which shows what attribute each value indicates along with its Header.bas constant name. The return values not supported under UNIX are 0.

 Table 6-49
 Return values for <attribute> with the corresponding constant names and file types

Return value	Constant name	File type	<b>Operating system</b>
0	ATTR_NORMAL	Normal	Windows, UNIX
1	ATTR_READONLY	Read-only	Windows, UNIX
2 or 0	ATTR_HIDDEN	Hidden	Windows
4 or 0	ATTR_SYSTEM	System file	Windows
32 or 0	ATTR_ARCHIVE	Changed since last backup	Windows

<file name> can optionally specify a full path, in which case it has the following syntax:

[[<drive:>] [ \ ]<directory>[\<directory>]...<file name> (Windows)

[/]<directory>[/<directory>]...<file name> (UNIX)

#### <drive:>

Character, followed by a colon, that specifies the drive (Windows only).

#### <directory>

String expression that specifies the name of a directory or subdirectory. For example, the following statement sets the System and Read-only (4 + 1) attributes for a file in the current directory called Testfile.dat:

```
SetAttr("Testfile.dat", 5)
```

**Tips •** To determine whether, or in what mode, a file is open, use FileAttr.

- To determine what attributes of a file, directory, or volume have already been set, use GetAttr.
- **Example** In the following example, if the Archive attribute for the specified file is set, the example displays a message. Otherwise, it sets the Archive attribute:

```
Sub Start( )
Dim FileName As String, Msg As String
Super::Start( )
On Error GoTo ErrorHandler
```

```
FileName = "C:\Program Files\Actuate11\readme.rtf"
  'Check for Archive attribute
  If GetAttr(FileName) BAnd ATTR ARCHIVE Then
     Msg = FileName & " already has the Archive attribute."
     ShowFactoryStatus( Msg )
  Else
     ' Set Archive attribute
     SetAttr FileName, ATTR ARCHIVE
     Msg = FileName & " now has the Archive attribute."
     ShowFactoryStatus ( Msq )
  End If
  Exit Function
ErrorHandler
  Msg = {"Sorry! An error occurred. Please
change the file name or path in this method code and
try again."}
  ShowFactoryStatus ( Msg )
End Sub
For information about using the code examples, see "Using the code examples,"
```

See also FileAttr function GetAttr function

earlier in this chapter.

### **SetBinding function**

Establishes an alias for an instance variable of a class. SetBinding is used primarily to set up a mapping from database column names to the names of variables in a data row class.

Syntax SetBinding(<instance handle>, <column name>, <variable name>)

### Parameters <instance handle>

The instance handle to one instance of the class for which to set up the binding.

### <column name>

The name, usually a column name, to map to a variable.

### <variable name>

The name of the variable or constant to associate with the column name.

**Tip** The column name mapped to the column name within a class lets you access variables through their column aliases using GetValue.

SetBinding is called automatically for data rows created through the Actuate Query Builder, and you can call SetBinding through custom code for data rows you create manually. For any class for which you want to set up bindings, you must call SetBinding at the beginning of your code. **See also** GetValue function

### SetClipboardText function

Places a string expression in the operating environment Clipboard using the specified Clipboard format.

SetClipboardText(<data>[, <format>]) Syntax

#### **Parameters** <data>

String data to be placed on the Clipboard.

#### <format>

One of the Clipboard formats in Table 6-50. If <format> is omitted, CF\_TEXT is assumed.

Table 6-50	Clipboard formats
1 able 6-50	Clipboard ionnais

Symbolic constant	Value	Clipboard format
CF_LINK	&HBF00	DDE conversation information
CF_TEXT	1	Text

The symbolic constants and their values should be declared in your Actuate Basic code module (.bas file).

#### Returns Integer

- Returns 1 (True) if Clipboard text is successfully set.
- Returns 0 (False) if Clipboard text cannot be set.

Example The following example places the current date on the Clipboard, then displays the contents of the Clipboard:

```
Sub Start()
  Dim Msg As String
  Super::Start( )
  On Error Resume Next
  Msg = "The Clipboard contains: " & GetClipboardText
  ShowFactoryStatus ( Msg )
  Msg = "Placing today's date on the clipboard."
  ShowFactoryStatus ( Msg )
  ClearClipboard
  SetClipboardText(Format$(Date, "dddd, mm/dd/yyyy"))
  Msg = GetClipboardText
  ShowFactoryStatus( "The Clipboard now contains: " & Msg )
End Function
```

See also ClearClipboard function GetClipboardText function

### SetDefaultPOSMFile function

Determines which open report object instance (.roi) file is used as the default when you call New Persistent to instantiate an object.

```
Syntax SetDefaultPOSMFile(<intFileID>)
```

- **Description** Use SetDefaultPOSMFile in connection with advanced report bursting. For example, you are creating two ROIs simultaneously. Your report scans a set of rows and, depending on the values in a given row, adds the row to one or the other ROI. Use SetDefaultPOSMFile to switch the default back and forth between the two ROIs. This function must be called on open report files.
- Parameters <intFileID> Integer. Specifies an open ROI.
  - Returns Boolean
    - True, if all arguments are correct.
    - False, if any of the arguments are invalid.
  - **Example** The first line of the following example sets the default POSM file to the currently open ROI with the file ID 1. The second line instantiates an object, abc, in ROI #1.

```
SetDefaultPOSMFile(1)
Set abc = New Persistent className
```

**See also** SetStructuredFileExpiration function

### SetHeadline statement

Sets to a given value the headline associated with the completed request for a report.

Syntax SetHeadline(<newHeadline>)

**Description** Headlines are associated with the completed request for a report. By default, the headline is set to the value of the Headline parameter that appears in the Output Options section of the Requester. Use SetHeadline to programmatically change the headline to contain some other text. Note that:

To change the headline, do not change the Headline parameter itself. Instead, use SetHeadline.

### Parameters <newHeadline>

String expression. The text you want the headline to display.

- **Tip** To determine the current headline setting, use GetHeadline.
- **Example** The following example displays the current headline, sets a new headline, then displays the new headline:

```
Sub Start( )
Super::Start( )
' Getting the current headline.
ShowFactoryStatus( "Original headline is: " & GetHeadline() )
' Setting and displaying the new headline
SetHeadline( "New Headline" )
ShowFactoryStatus( "New headline is: " & GetHeadline() )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also GetFontAverageCharWidth function

### SetStructuredFileExpiration function

Sets the object aging rules on a report object web (.row) file.

Syntax SetStructuredFileExpiration(<intFileID>, <intAgingOptions>, <intMaxVers>,

| [ { [Null, <intExpAge>] | [<datExpDate>, Null] } ] )

### Parameters <intFileID>

Integer. Specifies an open ROW file.

### <intAgingOptions>

Integer. Specifies the aging policy. The aging policy is expressed by using the constants shown in Table 6-51 together with a BOR operator.

 Table 6-51
 Global constant names and values for aging policies

Global constant name	Value	Meaning
Age_NoOptions	&H00	No object aging options are to be used.
Age_ArchiveBeforeDelete	&H01	Archives the file before deletion. Combine this constant with either a version, date, or age-based expiration.
		(continues)

Global constant name	Value	Meaning
Age_DeleteDependencies	&H02	Deletes dependents for the file. Note that ROWs usually do not have dependents. Combine this constant with either a version, date, or age-based expiration.

Table 6-51	Global constant names and values for aging policies (continued	)
------------	--	---

For example, to create a policy that deletes dependencies and archives before deletion, use code such as the following:

```
Dim policy As Integer
```

policy = Age\_ DeleteDependencies BOR Age\_ArchiveBeforeDelete

### <intMaxVers>

Specifies the maximum number of versions to keep. Set this to Null or 0 if no version limit is to be set.

### <intExpAge>

Integer. Specifies the age, in minutes, after which to expire the object. Set this to Null or 0 if no expiration age is to be set.

### <datExpDate>

Date. Specifies the date after which to expire the object. Set this to Null if no expiration date is to be set.

- <datExpDate> and <intExpAge> are optional, but if <intExpAge> is provided, <datExpDate> must also be provided, even though it is Null.
- Only one of <datExpDate> or <intExpAge> should be non-Null. If both are non-Null, only <intExpAge> is used.
- It is legal for <intMaxVers>, <datExpDate>, and <intExpAge> to be Null. If all three are Null, no aging policy is set for the file, as shown in the following example:

```
SetStructuredFileExpiration( fileID, Age_NoOptions, Null,
+ Null, Null )
```

### Returns Boolean

- True, if all arguments are correct.
- False, if any of the arguments are invalid.
- **Description** You customize object aging settings for a file as follows:
  - Set the values of the object's aging properties. For example, if you want a report to never archive more than ten versions, set <intMaxVers> to 10.
  - To ensure that a report expires no later than a week after creation, set the age variable, <intExpAge>, or the date expiration variable, <datExpDate>, accordingly.

Rule: This function must be called on the open report file.

**Example** The following example specifies that an ROW is to be archived and deleted after thirty minutes:

```
SetStructuredFileExpiration(fileID, Age_ArchiveBeforeDelete,
+ Null, Null, 30)
```

See also SetDefaultPOSMFile function

### **SetValue function**

Sets the value of a variable in an object dynamically at run time.

- **Syntax** SetValue(<object reference>, <variable name>|<index>, <value>)
- **Description** You typically use SetValue in conjunction with GetVariableCount, GetVariableName, GetValue, and GetValueType to work with variables in objects whose types you do not know in advance.

#### Parameters <object reference>

AnyClass expression that specifies an object which has a variable whose value you want to set.

#### <variable name>

String expression that specifies the name of the variable whose value you want to set.

#### <index>

Integer expression that specifies the index of the variable whose value you want to set.

Indexing starts at 1. The index order puts all superclass variables before those defined in a subclass.

Within a given class, the index order of variables is the order in which the variables are defined in the Actuate Basic source. If e.Report Designer Professional generates the Basic source for a class, it lists the variables in alphabetical order.

#### <value>

Variant. The value to assign.

**Returns** True if the specified variable exists.

False if the specified variable does not exist. It is not an error if the variable does not exist.

**Example** The following example shows how to use GetVariableCount, GetVariableName, GetValueType, GetValue, and SetValue together:

```
' Simulate the behavior of the CopyInstance statement,
' but only copy integers > 0 whose names begin "Z "
Dim vCount As Integer
Dim vName As String
Dim vType As Integer
Dim vValue As Variant
Dim i As Integer
vCount = GetVariableCount( fromObject )
For i = 1 To vCount
  vName = GetVariableName( fromObject, i )
  vType = GetValueType( fromObject, vName )
  If (Left( vName, 2 ) = "Z ") And (vType = V INTEGER) Then
     vValue = GetValue( fromObject, vName )
     If (vValue > 0) Then
        SetValue( toObject, vName, vValue )
     End If
  End If
Next i
```

See also GetValue function GetValueType function GetVariableCount function GetVariableName function

## Sgn function

Indicates whether a number is positive, negative, or zero.

Syntax Sgn(<number>)

Parameters <number> Number, numeric expression, or Variant of VarType 8 (String) that specifies the number for which you want to determine the sign.

**Returns** Same data type as <number>. If <number> is a Variant of type 8 (String), returns Variant of type 5 (Double). Returns one of the numbers in Table 6-52, indicating whether <number> is positive, negative, or zero.

Return Values	Description
1	Positive number
-1	Negative number
0	Zero

Fable 6-52	Return values	for Sgn functions
------------	---------------	-------------------

**Example** The following example generates a number, then shows whether that number is positive, negative, or zero:

```
Sub Start( )
Dim Number As Double, Msg As String
Super::Start( )
Number = CInt(Rnd * 10) - 5
Select Case Sgn(Number)
Case 0
Msg = Number & " is zero."
Case 1
Msg = Number & " is a positive number."
Case -1
Msg = Number & " is a negative number."
End Select
ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Abs function

### Shell function

Runs a specified executable program.

Syntax Shell(<program name and parameters> [, <window style>])

```
Parameters <program name and parameters>
```

String expression containing a valid executable name. Name of the program to execute, with its required arguments or command line switches, if there are any.

Default file extension under Windows: .exe.

- Must have an extension of .exe, .com, .bat, or .pif (Windows).
- Must have execute permission set (UNIX).
- Must be enclosed within quotes.
- If the executable is not located in the default directory:
  - Program name must include the full path.
  - Program must be located in a directory specified in the PATH environment variable.

Each of the following examples runs the Windows calculator, then immediately executes the next statement in the program, regardless of whether Calc.exe has terminated:

```
DummyVar = Shell("Calc.exe")
X = Shell("C:\Windows\Calc.exe")
```

The following example opens Excel and the file Extract.csv at the same time:

Shell( "D:\Program Files\Microsoft Office\Office\Excel.exe
+ C:\Temp\Extract.csv" )

#### <window style>

Ignored on UNIX systems. Integer that specifies the number for the style of the window to use for the new program, and whether or not to give it the focus. Table 6-53 identifies each possible value for <window style> and the resulting style and focus of the window.

Value	Window style	Focus
1, 5, 9	Normal	New program receives
2	Minimized	New program receives
3	Maximized	New program receives
4,8	Normal	Current program retains
6,7	Minimized	Current program retains

Table 6-53 Values for <window style>

If omitted, the default is 2.

The following example calls Word for Windows, maximizes its window, and gives it focus:

```
Z = Shell("C:\Word\Winword.exe", 3)
```

#### Returns Integer

- Shell loads an executable file into the Windows environment, and returns the unique task ID number that Windows assigns to each running program as its identifier (Windows).
- Shell runs programs asynchronously. You cannot be certain whether or not a program you start with Shell will finish executing before the code following the Shell function in your Actuate Basic application is executed.
- If Shell cannot start the named program, it generates an error.
- **Tips** To shell to DOS—in other words, to run the DOS command processor—specify the program name Command.com with Shell (Windows).
  - To enjoy greater flexibility and exercise better control over the behavior of external programs, use an OLE technique rather than Shell (Windows).
- **Example** In the following Windows example, Shell runs the Windows Calculator. The next statement in the program is then executed immediately, regardless of whether Calc.exe has terminated.

```
Sub Start( )
Dim DummyVar
Super::Start( )
DummyVar = Shell("C:\Windows\System32\Calc.exe", 1)
ShowFactoryStatus( "Calculator opened." )
End Sub
```

### ShowFactoryStatus statement

Displays a message in the Factory output and records the message in the job status file when you generate a report. Recording these messages in the job status file requires additional hard drive space.

Syntax ShowFactoryStatus(<StatusMsg>)

### Parameters <StatusMsg>

String you want to display in the Factory output window or job status file, as shown in the following example:

```
ShowFactoryStatus("Preparing the statement.")
```

## Sin function

Gives the sine of an angle.

Syntax	Sin( <angle>)</angle>	
Parameters	<angle> Number, numeric expression, or Variant of type 8 (String) that specifies, in radians, the angle for which you want to find the sine. If <angle> is a String, it is parsed according to the formatting rules of the current run-time locale.</angle></angle>	
Returns	Double	
	If <number> evaluates to Null, Sin returns Null.</number>	
Example	The following example generates an angle expressed in radians and returns the sine of the angle:	
	<pre>Sub Start() Dim Angle As Double, Pi As Double Super::Start() Pi = 3.14159265358979 Angle = Pi * Rnd ShowFactoryStatus("Sine of " &amp; Angle &amp; " is: " &amp; Sin(Angle)) End Sub</pre>	

See also Acos function Asin function Atn function Cos function Tan function

### Sleep statement

Temporarily suspends report execution.

Syntax Sleep(<seconds>)

Parameters <seconds>

Double expression that specifies the number of seconds to suspend report execution. The value must be in the range 0–600. The value can be a fraction, for example 0.5.

- **Description** You can use Sleep to suspend execution while waiting for an external event, for example a file being created. The timing accuracy of Sleep is dependent on the underlying operating system.
  - **Tip** Shell commands run asynchronously. This means that a report continues to execute as soon as a Shell command is started. To wait for the result of a Shell command, make the Shell command create a file when it is done, then loop around checking whether that file exists. Use Sleep inside the loop to avoid extra processor cycles.
  - **Example** The following example shows how to use Sleep to temporarily suspend report execution:

```
' Run a script and wait for it to create a file
' Remove previous file, if any
If FileExists( "myfile.txt" ) Then
   Kill "myfile.txt"
End If
' Run the script
Shell("myscript.bat")
' Wait for the file to appear
Dim startTime As Date
startTime = Now()
Do
```

```
' Check for file
If FileExists( "myfile.txt" ) Then
Exit Do
End If
' Timeout after 60 seconds
If (DateDiff( "s", startTime, Now() ) > 60) Then
Error 9999, "Time out!"
End If
' Wait a while and try again
Sleep( 2.5 )
Loop
' Now we can use the file
```

See also Shell function

### **SLN** function

Returns the straight-line depreciation of an asset for a single period.

Syntax SLN(<initial cost>, <salvage value>, <asset lifespan>)

#### Parameters <initial cost>

Numeric expression that specifies the initial cost of the asset.

#### <salvage value>

Numeric expression that specifies the value of the asset at the end of its useful life. You can type a salvage value to view the straight line depreciation offset by the salvage value, or return straight line depreciation without salvage value by entering 0 in salvage value.

#### <asset lifespan>

Numeric expression that specifies the length of the useful life of the asset. <asset lifespan> must be given in the same units of measure you want the function to return. For example, if you want SLN to determine the annual depreciation of the asset, <asset lifespan> must be given in years.

The following example calculates the depreciation under the straight-line method for a new machine purchased at \$1400, with a salvage value of \$200, and a useful life estimated at 10 years. The result (\$120 annually) is assigned to AnnualDeprec.

AnnualDeprec = SLN(1400, 200, 10)

#### Returns Double

Straight-line depreciation is the oldest and simplest method of depreciating a fixed asset. It uses the book value of the asset less its estimated residual value,

and allocates the difference equally to each period of the asset's life. Such procedures are used to arrive at a uniform annual depreciation expense that is charged against income before calculating income taxes.

Rule: All arguments must be positive numbers.

**Example** The following example provides various particulars about an asset, then returns the asset's straight-line depreciation for any single period:

```
Declare
  ' Number of months in a year
  Global Const YEARMONTHS = 12
End Declare
Sub Start()
  Dim Fmt As String, InitCost As Double, SalvageVal As Double
  Dim MonthLife As Double, LifeSpan As Double
  Dim PerDepr As Double
  Dim Msg As String
  Super::Start( )
  ' Define money format
  Fmt = "$#, ##0.00"
  InitCost = 12500 ' The initial cost of the asset
  SalvageVal = 1500 ' The asset's value at useful life end
  MonthLife = 30 ' The asset's useful life in months
  ' Convert months to years
  LifeSpan = MonthLife / YEARMONTHS
  If LifeSpan <> Int(MonthLife / YEARMONTHS) Then
     ' Round up to nearest year
     LifeSpan = Int( LifeSpan + 1 )
  End If
  PerDepr = SLN(InitCost, SalvageVal, LifeSpan)
  Msg = "The depreciation is " & Format (PerDepr, Fmt)
          " per year."
     &
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also DDB function SYD function

### Space, Space\$ functions

Returns a string consisting of the specified number of spaces.

```
Syntax Space(<number of spaces>)
```

Space\$(<number of spaces>)

### Parameters <number of spaces>

Numeric expression indicating the length of the returned string of spaces.

- Must be between 0 and 2,147,483,647 or memory limit.
- The <number of spaces> is rounded to Long regardless of its numeric data type.
- Returns Space: Variant Space\$: String

If <number of spaces> evaluates to Null, Space[\$] returns Null.

- **Tip** To generate a string of characters other than spaces, use String[\$].
- **Example** The following example creates a variable containing 10 spaces:

```
Sub Start( )
Dim Msg, Pad, UserFname, UserLname As String
Super::Start( )
UserFname = "Alain"
UserLname = "Colline"
' Create 10-space pad
Pad = Space(10)
Msg = "Notice the 10-space pad between the first "
+ & "and last names. "
ShowFactoryStatus( Msg )
Msg = UserFname & Pad & UserLname
ShowFactoryStatus( Msg )
End sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also String, String\$ functions

## Sqr function

Gives the square root of a number.

Syntax	Sqr( <number>)</number>
Parameters	<number> Number, numeric expression, or Variant of type 8 (String) indicating the number for which you want to determine the square root. If <number> is a String, it is parsed according to the formatting rules of the current run-time locale.</number></number>
Returns	Double

If <number> evaluates to Null, Sqr returns Null.

**Example** The following example generates a number and returns the square root of that number:

```
Sub Start()
Super::Start()
On Error GoTo ErrorHandler
Dim UserTyped, Msg
UserTyped = CInt( Rnd * 100 - 25 )
ShowFactoryStatus ("The square root of " & UserTyped &
+ " is: " & Sqr(UserTyped) )
Exit Sub
ErrorHandler:
Msg = "Sorry! An error occurred. "
+ & UserTyped & " is negative. Please try again."
ShowFactoryStatus( Msg )
Resume Next
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

### Static statement

Declares a static variable or static procedure.

Syntax Static <variable name> [([<subscripts>])] [As <type>] [, <variable name> [([<subscripts>])] [As <type>]]...

Static <procedure name>

- **Description** Static variables retain their values as long as the program is running, unlike local variables, which are cleared when the procedure ends. This does not mean, however, that you can access them from outside the procedure. Every time you execute the procedure in which the static variables are declared, the procedure can make use of and update the most recent values of those variables. Whatever changes it makes are preserved until the next time you call the procedure.
  - To declare a fixed-size array in a nonstatic procedure, you must use Static.
  - Use Static at the procedure level.

### Parameters <variable name>

Name you create for the new variable.

#### <procedure name>

Name you create for the new procedure. When you use the syntax Static <procedure name>, all the variables in <procedure name> are declared to be static.

#### <subscripts>

The dimensions of an array variable.

### Subscript syntax

[<lower> To] <upper>[,[<lower> To] <upper>]...

Rule: <lower> and <upper> can range from -2,147,483,648 to 2,147,483,647, inclusive.

For example, the following two statements have the same effect. Both declare a two-dimensional array that contains 11 elements in its first dimension, and 6 elements in its second.

```
Static MEM(10, 5)
Static MEM(0 TO 10, 0 TO 5)
```

#### As <type>

Clause in which you specify a data type for the new variable.

The default is Variant or the data type specified by the type declaration character appended to <variable name>, if there is one.

# **Tips** To avoid assigning incorrect variable types, use the As clause to declare variable types.

- Use the Option Strict statement to enforce variable typing.
- To retain the values of all the variables in a procedure, you do not have to declare each one individually as Static, or list each one on a Static declaration line. Instead, use Static <procedure name> to declare the procedure itself. Example:

Static Sub MyProcedure

**Example** In the following example, the value of the variable Accumulate is preserved between calls to the Sub procedure. Each time the procedure is called, the example generates a number to add to the Accumulate variable, then displays that value.

```
Sub Start( )
Static Accumulate As Integer
Dim AddTo As Integer
```

```
Super::Start()
AddTo = CInt( Rnd * 100 - 25 )
Accumulate = Accumulate + AddTo
ShowFactoryStatus( "The static variable has changed by "
+ & AddTo & ". It now has the following value: " &
+ Accumulate )
End Sub
```

```
See also Dim statement
Function...End Function statement
Option Strict statement
Sub...End Sub statement
```

## Stop statement

Suspends execution of the running Actuate Basic code.

Syntax Stop

**Description** I Unlike End, Stop does not close files or clear variables.

- To continue a program suspended by Stop, choose Debug→Continue, or press F5.
- **Tip** You can place Stop anywhere in procedures to suspend program execution. It is useful as a debugging aid.
- **Example** In the following example, for each step through the For...Next loop, Stop suspends execution. To resume program execution, choose Debug→Continue.

```
Sub Start( )
   Dim I As Integer
   Super::Start( )
' Begin For...Next loop
   For I = 1 To 10
' Show I
        ShowFactoryStatus ( CStr( I ) )
' Stop each time through loop
        Stop
        Next I
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also End statement

### Str, Str\$ functions

Converts a numeric expression to a String.

**Syntax** Str(<numeric expression>)

Str\$(<numeric expression>)

### Parameters <numeric expression>

Numeric expression to be converted to a String. <numeric expression> is parsed according to the formatting rules of the current run-time locale. For example, the following statements are equivalent. Each returns Customer 55.

```
"Customer" & Str$(55)
"Customer" & Str$(110/2)
```

### Returns Str: Variant

Str\$: String

- If <numeric expression> is positive, Str[\$] precedes the string representation of the number with a leading space.
- If < numeric expression> is negative, Str[\$] precedes the string representation of the number with a minus sign instead of a leading space.
- **Tips** To put the results of a calculation into a message for the user and avoid generating a Data Type Mismatch error, use Str[\$].
  - To convert numeric values you want rendered in currency, date, time, or userdefined formats, use Format[\$] instead of Str[\$].
  - To convert a String to a number, use Val.
- **Example** The following example returns a string representation of the values contained in two variables. Because the numbers are positive, a space precedes the first string character in each case.

```
Sub Start( )
   Dim Dice1, Dice2, Msg
   Super::Start( )
' Generate first value
   Dice1 = Int(6 * Rnd + 1)
' Generate second value
   Dice2 = Int(6 * Rnd + 1)
   Msg = "You first rolled a " & Str$(Dice1)
+ & & " and then a " & Str$(Dice2)
+ & & " for a total of "
+ & & Str$(Dice1 + Dice2) & "."
   ShowFactoryStatus( Msg )
End Sub
```

See also Format, Format\$ functions Val function

### StrComp function

Compares two strings and returns a Variant indicating the result.

Syntax StrComp(<string exprs 1>, <string exprs 2>[,<compare method>])

### Parameters <string exprs 1>, <string exprs 2>

Any string expression. Whether the underlying data type is Variant or String, each expression is first converted to a Variant before the arguments are compared.

#### <compare method>

Numeric expression that indicates whether <compare method> is case-sensitive or -insensitive as shown in Table 6-54.

able 6-54	Case-sensitivity	y indicators for	<compare< th=""><th>method&gt;</th></compare<>	method>
-----------	------------------	------------------	--	---------

Value	String comparison	Similar to
0	Case-sensitive	Option Compare Binary
1	Case-insensitive	Option Compare Text

<compare method> must be 0 or 1. The default is 0, unless you used Option Compare Text in the module.

- When <compare method> is case-sensitive (0), McManus does not match MCMANUS or McMANUS.
- When <compare method> is case-insensitive (1) McManus matches MCMANUS or McMANUS.

### Returns Variant

StrComp returns a Variant that reports the relationship between the two string expressions. Table 6-55 shows each return value and the relationship or condition that value signifies.

Table 6-55	StrComp return values and the corresponding relationships or conditions	
Return value	Relationship or condition	
-1	<string 1="" exprs=""> is less than <string 2="" exprs=""></string></string>	
0	<string 1="" exprs=""> = <string 2="" exprs=""></string></string>	
Table 6-55	StrComp return values and the corresponding relationships or conditions	
--------------	--	--
Return value	Relationship or condition	
1	<string 1="" exprs=""> is greater than <string 2="" exprs=""></string></string>	
Null	<string 1="" exprs=""> = Null or <string 2="" exprs=""> = Null</string></string>	

**Example** The following example prompts the user for the comparison method to use when comparing two strings. Then, it returns the result of the comparison along with a short explanation.

```
Sub Start()
  Dim Comp, CompMethod, Msg As String
  Dim String1 As String, String2 As String, Desc As String
  Super::Start( )
  String1 = "Starlight"
  String2 = "STARLIGHT"
  ' Assume case sensitive. Set CompMethod to 1 for
  ' NOT case sensitive.
  CompMethod = 0
  Desc = "case sensitive."
  Comp = StrComp(String1, String2, CompMethod)
  Select Case Comp
     Case -1
       Msg = String1 & " is less than " & String2
          & " for compare method " & CompMethod
+
          & ", which is " & Desc
+
     Case 0
       Msg = String1 & " is equal to " & String2
          & " for compare method " & CompMethod
+
          & ", which is " & Desc
+
     Case 1
       Msg = String1 & " is greater than " & String2
          & " for compare method " & CompMethod
+
          & ", which is " & Desc
+
  End Select
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Option Compare statement

## String, String\$ functions

Returns a string consisting of a specified character repeated a specified number of times.

**Syntax** String(<number of characters>,<char to repeat>)

String\$(<number of characters>,<character code number>)

### **Parameters** <number of characters> Numeric expression that specifies the number of times the character is to be repeated. The number of times must be between 0 and 65536.

### <char to repeat>

String expression that specifies the character to repeat.

- <char to repeat> must be at least one character long.
- If <char to repeat> is longer than one character, only the first character is repeated.
- If <char to repeat> is a Variant of any numeric type, String[\$] interprets the numeric value as the ANSI code for the repeated character.

#### <character code number>

Numeric expression that specifies the ANSI code for the repeated character.

- <character code number> must not be Null.
- <character code number> must be between 0 and 65535.
- <character code number> is interpreted as a value in the code page corresponding to the current run-time encoding.
- If < character code number> is a Variant of any numeric type, String[\$] interprets the numeric value as the ANSI code for the repeated character.

The following examples show the effect of <character code number>:

```
String$(5, 64)' Returns AAAAA
String$(5,227)' Returns γγγγγ on a Greek locale
String$(5, 227)' Returns äääää on an English locale
```

Returns String: Variant String\$: String

- If any parameter evaluates to Null, String[\$] returns Null.
- **Tips** To generate a string of spaces, use Space[\$].
  - If the run-time encoding is UCS-2, String behaves in the same way as StringW.

**Example** The following example returns a string consisting of 12 number signs (#) followed by a sample check amount. The ANSI code of the number sign character is 35.

```
Sub Start( )
Dim Msg As String
Super::Start( )
Msg = "This generates two strings of 12 number signs, "
+ & "followed by a sample check amount. "
+ & "Both forms of String$ syntax are shown."
+ & "First Form: " & String$(12, "#")
+ & "$100.00"
+ & "Second Form: " & String$(12, 35) & "$100.00"
ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Space, Space\$ functions StringW, StringW\$ functions

### StringW, StringW\$ functions

Returns a string consisting of a specified character in UCS-2 character set repeated a specified number of times.

**Syntax** StringW(<number of characters>,<char to repeat>)

StringW\$(<number of characters>,<character code number>)

### Parameters <number of characters>

Numeric expression that specifies the number of times the character is to be repeated. <number of characters> must be between 0 and 65536.

#### <char to repeat>

String expression that specifies the character to repeat.

- If <char to repeat> is longer than one character, only the first character is repeated.
- If <char to repeat> is a Variant of any numeric type, String[\$] interprets the numeric value as the ANSI code for the repeated character.

### <character code number>

Numeric expression that specifies the UCS-2 code for the repeated character.

- <character code number> must not be Null.
- <character code number> must be between 0 and 65535.

 If <character code number> is a Variant of any numeric type, String[\$] interprets the numeric value as the UCS-2 code for the repeated character.

StringW\$(10,"#")

The following statement returns *үүүүүүү*?

StringW\$(10,947)

Returns String

If any parameter evaluates to Null, StringW[\$] returns Null.

**Example** The following example returns a string consisting of 12 number signs (#) followed by a sample check amount. The ANSI code of the number sign character is 35.

For information about using the code examples, see "Using the code examples," earlier in this chapter.

### StrSubst function

Replaces part of a string with a different substring and returns the result.

**Syntax** StrSubst(<strTarget>, <strToFind>, <strNew>)

### Parameters <strTarget>

String expression. The string within which you want to locate <strToFind>.

### <strToFind>

String expression. The string you want to locate within <strTarget> so you can replace it with <strNew>.

### <strNew>

String expression. The new substring within <strTarget> that you want in place of <strToFind>.

Returns String

<strTarget>, but with substring <strNew> in place of <strToFind>.

If <strToFind> does not exist within <strTarget>, Actuate returns <strTarget> unchanged.

**Example** The following example substitutes a subsection of a string with another string:

```
Sub Start()
  Dim strTarget As String, strToFind As String, strNew As String,
  Msg As String
  ' Set string, value to find, and new value
  strTarget = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
  strToFind = "LMN"
  strNew = "123"
  ' Display original string
  Msg = "Original target string: " & strTarget
  ShowFactoryStatus ( Msg )
  ' Change the string, and display a message containing new
  string
  strTarget = StrSubst(strTarget, strToFind, strNew)
  Msg = "The string " & strTarget &
+ " used to contain " & strToFind &
+ " and now contains " & strNew
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

### See also Format, Format\$ functions Mid, Mid\$ statements

### Sub...End Sub statement

Declares and defines the name, code, and any arguments that constitute an Actuate Basic Sub procedure.

Syntax [Static] Sub <name of procedure> [(<list of arguments>)]

[<statements>]

[Exit Sub]

[<statements>]

End Sub

**Description** Like a Function procedure, a Sub procedure can take arguments, execute a series of statements, and change the values of its arguments. However, unlike a

Function procedure, a Sub procedure does not return a value directly and cannot be used in an expression.

You can use the same name for two or more different procedures, as long as you make sure the procedures take a different number of arguments, or that the arguments are of different data types. For example, you can write two different square root functions—one that operates on integers, and another that operates on doubles. Their respective first lines might look like the following:

Function SquareRoot(intParam As Integer) As Integer Function SquareRoot(dblParam As Double) As Double

Actuate Basic knows which procedure you mean by the type of value you pass when you call the procedure. For instance, if you write a call to SquareRoot(5), the compiler chooses the first SquareRoot function. But if you call SquareRoot(5.1234567), or SquareRoot(5.000), it executes the second one.

- You cannot define a Sub procedure from within another Sub procedure.
- You cannot use a Sub procedure in an expression, as you can a Function procedure.
- You cannot use GoTo to enter or exit a Sub procedure.

### Parameters Static

Keyword that instructs Actuate Basic to preserve the values of the procedure's local variables between calls.

- Cannot use Static to affect variables declared outside the procedure.
- Avoid using Static in recursive Sub procedures.

### <name of procedure>

The name you assign to the procedure.

The following conditions apply to <name of procedure>:

- Subject to the same constraints as variable names, as well as to the following additional constraints:
  - Cannot include a type declaration character.
  - Cannot be the same as any other globally recognized name like that of:
    - Procedure in a declared dynamic-link library (DLL).
    - Any Global variable.
    - Any Global constant.
- Can be overloaded. That is, you can define another function and/or procedure that has the same name, as long as the respective arguments are unique. For example, the following are both permissible in the same program:

```
Sub Potato(intTomato As Integer)
...
End Sub
Sub Potato(dblTomato As Double, strEggplant As String)
...
End Sub
```

### <list of arguments>

The list of variables that Actuate Basic passes as arguments from the calling procedure to the Sub procedure.

#### <statements>

One or more valid Actuate Basic statements. These statements constitute the body of the Sub procedure.

### Exit Sub

Statement that signals Actuate Basic to terminate the Sub procedure and transfer control to the statement following the one that called the procedure. You may use as many Exit Sub statements as you wish, anywhere in a Sub procedure.

list of arguments> has the following syntax:

[**ByVal**] <variable name>[()] [**As** <data type>] [,[**ByVal**] <variable name>[()] [**As** <data type>]...

### ByVal

Keyword that instructs Actuate Basic to pass the argument to the procedure by value rather than by reference. This means that the changes the Sub makes to the argument variable have no effect on its value in the calling procedure.

ByVal cannot be used with a variable of user-defined type, class, or with an array variable.

### <variable name>

Name of the variable to pass as an argument.

If the Sub procedure changes the value of <variable name> internally, then it also changes its value externally—that is, in the procedure that called the Sub.

- You must use the ByVal keyword if you do not want the Sub's changes to an argument variable to affect the variable's value in the calling procedure.
- For array variables, use the parentheses but omit the number of dimensions.

For example, the following statement calls a Sub procedure named SalesTax and passes to it as its argument an array variable previously declared as MyArray(50,72):

```
SalesTax MyArray()
```

### As <data type>

Clause that declares the data type of <variable name>. <data type> can specify any valid Actuate Basic or user-defined data type except a fixed-length string. For example:

SalesTax Customer As String, Amount As Currency

- **Tips** Sub procedures can be recursive. Actuate Basic sets a run-time stack limit of 200. A report using recursion with a large number of iterations might exceed this limit.
  - To evaluate a certain condition and then determine whether or not the Sub procedure should continue, use Exit Sub within a conditional structure such as If Then.
  - See Call for further information on how to call Sub procedures.
- **Example** The following example declares a Sub procedure. This example overrides Start() to generate the data, then calls the new procedure.

```
Sub Start()
  Dim PassThisArq1 As Double, PassThisArq2 As Double
  Super::Start( )
  ' Amount of the sale
  PassThisArg1 = Rnd * 1500 + 100
  ' Sales tax rate, expressed as a decimal
  PassThisArg2 = Rnd * 0.1
  SalesTax( PassThisArq1, PassThisArq2 )
End Sub
Sub SalesTax ( Arg1, Arg2 )
  Dim LocalVar As Double, Fmt As String, Msg As String
  ' Define money formatFmt = "$#,##0.00"
  LocalVar = Arg1 * Arg2
  Msg = "The sales tax on " & Format$(Arg1, Fmt)
     & " at " & Format$( Arg2, "#.00%" ) &
+
     " is: " & Format$( LocalVar, Fmt )
+
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Call statement Dim statement Function...End Function statement Static statement

### **SVGAttr function**

Returns a string for setting attributes in SVG code. The string provides the nonlocale dependent format that SVG requires.

Syntax SVGAttr(<attribute name>, <attribute value>)

Parameters <attribute name>

String expression that specifies the name of the attribute.

#### <attribute value>

Numeric or string expression that specifies the value of the attribute.

- Returns String
- **Example** In the following example, a chart's DrawOnChart() method has been overridden to draw a filled rectangle with rounded corners behind the chart drawing plane:

```
Sub DrawOnChart ( baseLayer As AcChartLayer,
+ overlayLayer As AcChartLayer, studyLayers() As AcChartLayer )
  ' Get the size of the drawing in points
 Dim w As Double
 w = Size.Width / OnePoint
 Dim h As Double
 h = Size.Height / OnePoint
 Dim svq As String
 svg = "<svg version='1.1'"</pre>
+ ' Standard SVG 1.1 namespaces
+ & " xmlns='http://www.w3.org/2000/svg'"
+ & " xmlns:xlink='http://www.w3.org/1999/xlink'"
+ ' Do not collapse whitespace in text
+ & " xml:space='preserve'"
+ ' Scale the SVG to use points as the default units
+ & " viewBox='0 0 " & SVGDbl( w ) & " " & SVGDbl( h ) & "'>"
  ' Draw the background rectangle
 svq = svq
  & "<rect"
+
    & SVGColorAttr( "fill", RGB( 255, 255, 204 ) )
  & SVGColorAttr( "stroke", Black )
  & SVGAttr( "stroke-width", 3.0)
  & SVGAttr( "x", 1.5 )
  & SVGAttr( "y", 1.5 )
  & SVGAttr( "width", w - 3.0 )
  & SVGAttr( "height", h - 3.0 )
+ & SVGAttr( "rx", 9.0 )
   & "/>"
```

+ & "</svg>"
' Insert the background rectangle behind the chart
Dim svgPlane As AcDrawingSVGPlane
Set svgPlane = InsertDrawingPlane( 1, DrawingPlaneTypeSVG )
svgPlane.SetSVG( svg )
End Sub

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also SVGColorAttr function SVGDbl function SVGFontStyle function SVGStr function SVGStyle function

### SVGColorAttr function

Returns a string for setting color attributes in SVG code. The string provides the non-locale dependent format that SVG requires.

**Syntax** SVGColorAttr(<color attribute>, <color value>)

#### Parameters <color attribute>

String expression that specifies the name of the color attribute.

#### <color value>

Numeric expression that specifies the value of the color.

**Returns** String

# **Example** In the following example, a chart's DrawOnChart() method has been overridden to draw a filled rectangle with rounded corners behind the chart drawing plane:

to draw a filled rectangle with rounded corners behind the chart drawing pla Sub DrawOnChart( baseLayer As AcChartLayer, + overlayLayer As AcChartLayer, studyLayers() As AcChartLayer ) ' Get the size of the drawing in points Dim w As Double w = Size.Width / OnePoint Dim h As Double h = Size.Height / OnePoint Dim svg As String svg = "<svg version='1.1'" + ' Standard SVG 1.1 namespaces + & " xmlns='http://www.w3.org/2000/svg'" + & " xmlns:xlink='http://www.w3.org/1999/xlink'" + ' Do not collapse whitespace in text + & " xml:space='preserve'"

```
+ ' Scale the SVG to use points as the default units
+ & " viewBox='0 0 " & SVGDbl( w ) & " " & SVGDbl( h ) & "'>"
  ' Draw the background rectangle
 svq = svq
  & "<rect"
  & SVGColorAttr( "fill", RGB( 255, 255, 204 ) )
  & SVGColorAttr( "stroke", Black )
  & SVGAttr( "stroke-width", 3.0 )
+ & SVGAttr( "x", 1.5 )
  & SVGAttr( "y", 1.5 )
+
  & SVGAttr( "width", w - 3.0 )
  & SVGAttr( "height", h - 3.0 )
+ & SVGAttr( "rx", 9.0 )
  & "/>"
  & "</svq>"
  ' Insert the background rectangle behind the chart
 Dim svgPlane As AcDrawingSVGPlane
 Set svqPlane = InsertDrawingPlane( 1, DrawingPlaneTypeSVG )
  svqPlane.SetSVG( svq )
End Sub
```

See also SVGAttr function SVGDbl function SVGFontStyle function SVGStr function SVGStyle function

## **SVGDbl** function

Returns a string for setting a numeric value in SVG code. The string provides the non-locale dependent format that SVG requires.

Syntax SVGDbl(<numeric value>)

Parameters <numeric value>

Numeric expression that specifies the value of the attribute.

- Returns String
- **Example** In the following example, a chart's DrawOnChart() method has been overridden to draw a filled rectangle with rounded corners behind the chart drawing plane:

Sub DrawOnChart( baseLayer As AcChartLayer, + overlayLayer As AcChartLayer, studyLayers() As AcChartLayer )

```
' Get the size of the drawing in points
 Dim w As Double
 w = Size.Width / OnePoint
 Dim h As Double
 h = Size.Height / OnePoint
 Dim svq As String
 svg = "<svg version='1.1'"</pre>
+ ' Standard SVG 1.1 namespaces
+ & " xmlns='http://www.w3.org/2000/svg'"
+ & " xmlns:xlink='http://www.w3.org/1999/xlink'"
+ ' Do not collapse whitespace in text
+ & " xml:space='preserve'"
+ ' Scale the SVG to use points as the default units
+ & " viewBox='0 0 " & SVGDbl( w ) & " " & SVGDbl( h ) & "'>"
  ' Draw the background rectangle
 svq = svq
   & "<rect"
   & SVGColorAttr( "fill", RGB( 255, 255, 204 ) )
   & SVGColorAttr( "stroke", Black )
+
 & SVGAttr( "stroke-width", 3.0 )
+
  & SVGAttr( "x", 1.5 )
+
 & SVGAttr( "y", 1.5 )
+
  & SVGAttr( "width", w - 3.0 )
+
  & SVGAttr( "height", h - 3.0 )
+
  & SVGAttr( "rx", 9.0 )
+
  & "/>"
   & "</svq>"
 ' Insert the background rectangle behind the chart
 Dim svgPlane As AcDrawingSVGPlane
 Set svqPlane = InsertDrawinqPlane( 1, DrawinqPlaneTypeSVG )
 svqPlane.SetSVG( svq )
End Sub
```

See also SVGAttr function SVGColorAttr function SVGFontStyle function SVGStr function SVGStyle function

### SVGFontStyle function

Returns a string for setting font values in SVG code. The string provides the non-locale dependent format that SVG requires.

Syntax SVGColorAttr(<font style name>, <font params>[,<style overrides>])

### Parameters <font style name>

String expression that specifies the name of the font style.

### <font params>

AcFont object that specifies font parameters.

### <style overrides>

Optional string expression that specifies font style overrides.

- **Returns** String
- **Example** In the following example, a chart's DrawOnChart() method has been overridden to replace the chart with the words "No Data!" if the chart has no data points:

```
Sub DrawOnChart( baseLayer As AcChartLayer,
+ overlayLayer As AcChartLayer, studyLayers() As AcChartLayer )
  ' Check for empty chart
 Dim hasData As Boolean
  Dim numberOfSeries As Integer
 numberOfSeries = baseLayer.GetNumberOfSeries( )
 Dim seriesIndex As Integer
  For seriesIndex = 1 To numberOfSeries
   Dim series As AcChartSeries
    Set series = baseLayer.GetSeries( seriesIndex )
    If Series.GetNumberOfPoints( ) > 0 Then
      hasData = True
      Exit For
    End If
 Next seriesIndex
  If hasData Then
   Exit Sub
 End If
' Hide empty chart
  GetChartDrawingPlane( ).SetHidden( True )
  ' Get the size of the drawing in points
 Dim w As Double
  w = Size.Width / OnePoint
 Dim h As Double
 h = Size.Height / OnePoint
 Dim svg As String
 svg = "<svg version='1.1'"</pre>
+ ' Standard SVG 1.1 namespaces
+ & " xmlns='http://www.w3.org/2000/svg'"
+ & " xmlns:xlink='http://www.w3.org/1999/xlink'"
+ ' Do not collapse whitespace in text
```

### SVGStr function

```
+ & " xml:space='preserve'"
+ ' Scale the SVG to use points as the default units
+ & " viewBox='0 0 " & SVGDbl( w ) & " " & SVGDbl( h ) & "'>"
  ' Define the font style
  Dim messageFont As AcFont
  messageFont.Bold = True
  messageFont.Color = Red
  messageFont.FaceName = "Arial"
  ' Font size is 25% of chart height
  messageFont.Size = h * 0.25
 svg = svg
+ & "<defs>"
+ & SVGFontStyle( "Message", messageFont, "text-anchor:middle;" )
+ & "</defs>"
  ' Draw the text
  svq = svq
+ & "<text class='Message'"
+ ' Center text horizontally and vertically
+ & SVGAttr( "x", w * 0.5 )
+ & SVGAttr( "y", (h * 0.5) + (messageFont.Size * 0.35) )
+ & ">"
+ & <u>SVGStr</u>( "No Data!" )
+ & "</text>"
+ & "</svg>"
  ' Add the text in front of the chart
  Dim svgPlane As AcDrawingSVGPlane
  Set svqPlane = AddDrawingPlane( DrawingPlaneTypeSVG )
  svqPlane.SetSVG( svq )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also SVGAttr function SVGColorAttr function SVGDbl function SVGStr function SVGStyle function

### SVGStr function

Returns a string with escaped characters for those characters that have special meaning to SVG.

Syntax SVGStr(<string value>)

### Parameters <string value>

String expression that specifies the value of the string attribute.

**Returns** String

```
Example In the following example a chart's DrawOnChart() method has been overridden to add some translucent text in front of the chart:
```

```
Sub DrawOnChart ( baseLayer As AcChartLayer,
+ overlayLayer As AcChartLayer, studyLayers() As AcChartLayer )
  ' Get the size of the drawing in points
  Dim w As Double
  w = Size.Width / OnePoint
  Dim h As Double
  h = Size.Height / OnePoint
  ' Create SVG to draw some translucent text
  Dim svq As String
  svg = "<svg version='1.1'"</pre>
+ ' Standard SVG 1.1 namespaces
+ & " xmlns='http://www.w3.org/2000/svg'"
+ & " xmlns:xlink='http://www.w3.org/1999/xlink'"
+ ' Do not collapse whitespace in text
+ & " xml:space='preserve'"
+ ' Scale the SVG to use points as the default units
+ & " viewBox='0 0 " & SVGDbl( w ) & " " & SVGDbl( h ) & "'>"
  ' Define the font style
  Dim sampleFont As AcFont
  sampleFont.Bold = True
  sampleFont.Color = Red
  sampleFont.FaceName = "Arial"
  sampleFont.Size = 80
  svq = svq
+ & "<defs>"
+ & SVGFontStyle( "Sample", sampleFont )
+ & "</defs>"
  ' Draw the text
 svg = svg
+ & "<text class='Sample'"
+ & " transform='translate(60,250) rotate(-30)'"
+ & " fill-opacity='0.35'>"
+ & SVGStr( "SAMPLE" )
+ & "</text>"
+ & "</svq>"
  ' Add the text in front of the chart
  Dim svgPlane As AcDrawingSVGPlane
  Set svgPlane = AddDrawingPlane( DrawingPlaneTypeSVG )
  svgPlane.SetSVG( svg )
End Sub
```

See also SVGAttr function SVGColorAttr function SVGDbl function SVGFontStyle function SVGStyle function

# **SVGStyle function**

	Returns a string for setting a CSS style in SVG code. The string provides the non-locale dependent format that SVG requires.
Syntax	SVGStyle( <styleclass>, <attributes>)</attributes></styleclass>
Parameters	<styleclass> String expression that specifies the name of a CSS style class.</styleclass>
	<attributes> String expression that specifies the attributes for the CSS style.</attributes>
Returns	String
Example	In the following example, a chart's DrawOnChart() method has been overridden to replace the chart with the words "No Data!" if the chart has no data points:
	<pre>Sub DrawOnChart( baseLayer As AcChartLayer, overlayLayer As AcChartLayer, studyLayers() As AcChartLayer ) Super::DrawOnChart( baseLayer, overlayLayer, studyLayers ) ' Check for empty chart Dim hasData As Boolean Dim numberOfSeries As Integer numberOfSeries = baseLayer.GetNumberOfSeries( ) Dim seriesIndex As Integer For seriesIndex = 1 To numberOfSeries Dim series As AcChartSeries Set series = baseLayer.GetSeries( seriesIndex ) If Series.GetNumberOfPoints( ) &gt; 0 Then hasData = True Exit For End If Next seriesIndex If hasData Then Exit Sub End If ' Hide empty chart GetChartDrawingPlane( ).SetHidden( True )</pre>

```
' Get the size of the drawing in points
  Dim w As Double
  w = Size.Width / OnePoint
  Dim h As Double
  h = Size.Height / OnePoint
  Dim svq As String
  svg = "<svg version='1.1'"</pre>
+ ' Standard SVG 1.1 namespaces
+ & " xmlns='http://www.w3.org/2000/svg'"
+ & " xmlns:xlink='http://www.w3.org/1999/xlink'"
+ ' Do not collapse whitespace in text
+ & " xml:space='preserve'"
+ ' Scale the SVG to use points as the default units
+ & " viewBox='0 0 " & SVGDbl( w ) & " " & SVGDbl( h ) & "'>"
  ' Define the font style
  Dim messageFont As AcFont
 messageFont.Bold = True
  messageFont.Color = Red
  messageFont.FaceName = "Arial"
  ' Font size is 25% of chart height
  messageFont.Size = h * 0.25
  svq = svq
+ & "<defs>"
+ & SVGStyle( "Message",
+ "font-family:" & messageFont.FaceName & ";"
+ & "font-size:" & SVGDbl( messageFont.Size ) & "px;"
+ & "font-weight:" & IIf( messageFont.Bold, "bold", "normal" ) &
  " ; "
+ & "font-style:" & IIf( messageFont.Italic, "italic", "normal" )
  & ";"
+ & "text-decoration:" & "none" & ";"
+ & "fill:" & SVGColor( messageFont.Color ) & ";"
+ & "stroke:none;"
+ & "text-anchor:middle;" )
+ & "</defs>"
' Draw the text
  svg = svg
+ & "<text class='Message'"
+ ' Center text horizontally and vertically
+ & SVGAttr( "x", w * 0.5 )
+ & SVGAttr( "y", (h * 0.5) + (messageFont.Size * 0.35) )
+ & ">"
+ & SVGStr( "No Data!" )
+ & "</text>"
+ & "</svg>"
```

' Add the text in front of the chart Dim svgPlane As AcDrawingSVGPlane Set svgPlane = AddDrawingPlane( DrawingPlaneTypeSVG ) svgPlane.SetSVG( svg ) End Sub

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also SVGAttr function SVGColorAttr function SVGDbl function SVGStr function SVGFontStyle function

### **SYD** function

Returns sum-of-years'-digits depreciation of an asset for a specified period.

Syntax SYD(<initial cost>, <salvage value>, <asset lifespan>, <single period>)

### Parameters <initial cost>

Numeric expression that specifies the initial cost of the asset.

### <salvage value>

Numeric expression that specifies the value of the asset at the end of its useful life.

#### <asset lifespan>

Numeric expression that specifies the length of the useful life of the asset.

<asset lifespan> must be given in the same units of measure as <single period>. For example, if <single period> represents a month, then <asset lifespan> must be expressed in months.

#### <single period>

Numeric expression that specifies the period for which you want SYD to calculate the depreciation.

<single period> must be given in the same units of measure as <asset lifespan>. For example, if <asset lifespan> is expressed in months, then <single period> must represent a period of one month.

The following example calculates the depreciation for the first year under the sum-of-years'-digits method for a new machine purchased at \$1400, with a salvage value of \$200, and a useful life estimated at 10 years. The result, \$218.18, is assigned to Year1Deprec. You may wish to note (a) that this result is equivalent to 10/55 \* \$1,200; (b) that 55 = 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1; and (c) that 10 is the 1st (Year 1) term in this series of digits.

YearlDeprec = SYD(1400, 200, 10, 1)

The following example calculates the depreciation of the same asset for the second year of its useful life. The result, \$196.36, is assigned to Year2Deprec. You may wish to note (a) that this result is equivalent to 9/55 \* \$1,200; (b) that 55 = 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1; and (c) that 9 is the 2nd (Year 2) term in this series of digits.

Year2Deprec = SYD(1400, 200, 10, 2)

#### **Returns** Double

Sum-of-years'-digits is an accelerated method of depreciation that results in higher depreciation charges and greater tax savings in the earlier years of the useful life of a fixed asset than are given by the straight-line depreciation method (SLN), where charges are uniform throughout.

The method bases depreciation on an inverted scale of the total of digits for the years of useful life. For instance, if the asset's useful life is 4 years, the digits 4, 3, 2, and 1 are added together to produce 10. SYD for the first year then becomes 4/10ths of the depreciable cost of the asset (cost less salvage value). The rate for the second year becomes 3/10ths, and so on.

Rules:

- <asset lifespan> and <single period> must both be expressed in terms of the same units of time.
- All arguments must be positive numbers.
- **Example** The following example prompts the user for various particulars about an asset, then returns the asset's sum-of-years'-digits depreciation for any single period:

```
Declare
   ' Number of months in a year
Global Const YEARMONTHS = 12
End Declare
Sub Start()
  Dim Fmt As String, InitCost As Double, SalvageVal As Double
  Dim MonthLife As Double, LifeSpan As Double
  Dim PeriodDepr As Double, DepYear As Integer, Msg As String
  Super::Start( )
   ' Define money format
  Fmt = "$#, ##0.00"
   ' Initial cost of the asset
  InitCost = Rnd * 5000 + 2000
   ' Asset's value at the end of its life
  SalvageVal = InitCost * 0.2
   ' The asset's useful life in months
  MonthLife = Rnd * 48 + 24
```

```
' Convert months to years
  LifeSpan = MonthLife / YEARMONTHS
  If LifeSpan <> Int(MonthLife / YEARMONTHS) Then
     ' Round up to nearest year
     LifeSpan = Int( LifeSpan + 1 )
  End If
  ' The year for which to show depreciation
  DepYear = CInt( Rnd * LifeSpan + 1 )
  Do While DepYear > LifeSpan
     DepYear = CInt( Rnd * LifeSpan + 1 )
  Loop
  PeriodDepr = SYD( InitCost, SalvageVal, LifeSpan, DepYear )
  Msg = "The depreciation on " & Format( InitCost, Fmt )
     & " with an end of life value of "
+
     & Format(SalvageVal, Fmt) & " for year " & DepYear
+
     & " is " & Format( PeriodDepr, Fmt ) & "."
  ShowFactoryStatus ( Msg )
End Sub
```

See also DDB function SLN function

### **Tab function**

Specifies the columnar position at which the next character is to print.

Syntax Tab(<column>)

**Description** The first print position on an output line is always 1. When you use Print to print to files, the last print position is the current width of the output file, which you can set using Width. When you use Tabs without separators, the semicolon (;) separator is assumed by default. There is no simple correlation between the number of characters printed and the number of fixed-width columns they occupy. For example, the uppercase letter W occupies more than one fixed-width column.

Table 6-56 describes Tab behavior.

Table	6-56	Tab	behavio
Table	6-56	Tab	behavio

If	Tab
The current print position on the current line is greater than <column>.</column>	Skips to <column> on the next output line and continues printing there.</column>

If	Tab	
<column> is greater than the output- line width (set by Width).</column>	Calculates <print position=""> = <column> Mod <width>.</width></column></print>	
<column> is less than the current print position.</column>	Continues printing on the next line at the calculated print position.	
<column> is less than 1.</column>	Moves the print position to column 1.	
The calculated print position is greater than the current print position.	Continues printing at the calculated print position on the same line, as expected.	

You can use Tab only with the Print statement.

### Parameters <column>

Integer expression. Specifies the column at which printing is to continue.

For example, the following statements open a file and use Tab to move the print position to column 40 in an output data file:

```
Open "Test.fil" For Output As #1
Print #1, "Here is a test of the "; Tab(40); "Tab function."
Close #1
```

- **Tips** To accommodate wider letters, make sure your tabular columns are wide enough.
  - To set the width of the output file, use Width.
- **Example** The following example uses the Tab function to move the print position to column 40 in an output data file:

```
Sub Start( )
Dim Msg
Super::Start( )
'Create sample file
Open "Test.fil" For Output As #1
Print #1, "This is a test of the "; Tab(40); "Tab function."
'Close file
Close #1
'Reopen test file for input
Open "Test.fil" For Input As #1
'Read test data
Input #1, Msg
'Close file
Close #1
ShowFactoryStatus( Msg )
```

```
'Delete test file
Kill "Test.fil"
End Sub
```

See also Print statement

### **Tan function**

Returns the tangent of an angle.

Syntax Tan(<angle>)

### Parameters <angle> Number, numeric expression, or Variant of type 8 (String) that specifies, in

radians, the angle for which you want to find the tangent.

If <angle> is a String, it is parsed according to the formatting rules of the current run-time locale.

- Returns Double
  - **Tips** If <angle> evaluates to Null, Tan returns Null.
    - To find the cotangent of an angle, use: Cotangent = 1/Tan.
    - To convert between radians and degrees, use: radians = degrees \* Pi/180.
- **Example** The following example prompts the user for an angle expressed in radians and returns the tangent of the angle:

```
Sub Start( )
   Dim Angle As Double, Pi As Double
   Super::Start( )
   Pi = 3.14159265358979
   ' The angle in radians
   Angle = Pi * Rnd
   ShowFactoryStatus( "The tangent of " & Angle & " is: "
+   & Tan( Angle ) )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Acos function Asin function Atn function Cos function Sin function

### **Time, Time\$ functions**

Returns the current system time.

Syntax Time

Time\$

The following statement assigns the current system time to a variable:

CurrTime = Time

# Returns Time: Variant Time\$: String

- Time returns a Variant of type 7 (Date) that contains a time stored internally as the fractional part of a double-precision number.
- Time\$ returns an 8-character string of the form hh:nn:ss, where hh represents the hour (00-23), nn the minute (00-59), and ss the second (00-59). The function uses a 24-hour clock, so 8:00 P.M. appears as 20:00:00.
- The return value of Time\$ is equivalent to that of the following statement:

Format\$(Now, "hh:nn:ss")

**Example** The following example selects a lucky number for the user based on the exact time the user calls the routine. This number ranges from 1 to 48.

```
Sub Start( )
Dim CurrTime, Msg, UserUniv, PowerUp, Choice
Super::Start( )
CurrTime = Time
UserUniv = 48
PowerUp = CurrTime * 1000000
Choice = (PowerUp Mod UserUniv) + 1
Msg = "Your lucky number, based on the "
+ & "exact time of day is: " & Choice
ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also CVDate function Date, Date\$ functions Format, Format\$ functions Now function

### Timer function

Returns the number of seconds that elapsed since midnight of the current day.

### Syntax Timer

The following statements use the Timer function to determine how long the procedure, TestRoutine takes to execute:

```
StartTime = Timer
Call TestRoutine
EndTime = Timer
TimeElapsed = EndTime - StartTime
```

### Returns Double

- **Tips** Timer is automatically used with Randomize to generate a seed for the Rnd (random-number) function.
  - You can also use Timer to time programs or parts of programs.
- **Example** The following example generates a number. The computer then counts up to that number using a simple For...Next loop. Actuate Basic times this counting process and reports the total number of elapsed seconds.

```
Sub Start()
  Dim UserNum, StartTime, I, EndTime, TimeElapsed, Msg
  ' The number to count up to
  UserNum = CInt( Rnd * 50 ) * 100000 + 1000000
  StartTime = Timer
  For I = 1 To UserNum
  Next I
  EndTime = Timer
  TimeElapsed = EndTime - StartTime
  Msg = "The time it took your computer to count to "
    & UserNum & " was: " & TimeElapsed
+
     & " seconds."
+
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Randomize statement

### **TimeSerial function**

Returns a date variant based on supplied integer hour, minute, and second arguments.

**Syntax** TimeSerial(<hour>, <minute>, <second>)

### Parameters <hour>

Numeric expression that specifies a particular hour of the day. <hour> must be in the range 0 (12:00 A.M.) through 23 (11:00 P.M.), inclusive.

### <minute>

Numeric expression that specifies a particular minute after <hour>. <minute> must be in the range 0 through 59, inclusive.

#### <second>

Numeric expression that specifies a particular second. <second> must be in the range 0 through 59, inclusive.

For example, the following statements are equivalent. Each stores 05:03:46 to the variable TimeVar.

TimeVar = TimeSerial(05, 03, 46)
TimeVar = TimeSerial(5, 3, 46)

Each of the following statements stores the underlying serial number for 05:03:46, which is about .210949, to the double-precision variable TimeVar#:

```
TimeVar# = CDbl(TimeSerial(5, 3, 46))
TimeVar# = TimeSerial(5, 3, 46) * 1
```

### Returns Date

The value TimeSerial returns usually looks like a time but is stored internally as a double-precision fractional number between 0 and .999999. This number represents a time between 00:00:00 and 23:59:59, or 12:00:00 A.M. and 11:59:59 P.M., inclusive.

In other contexts, this fractional number becomes part of what is known as a date serial number—a number that represents a date and/or time from midnight January 1, 100, through December 31, 9999, inclusive.

The integer component of any date serial number represents the date (day, month, and year) while the decimal or fractional component represents the time of day on that date as a proportion of a whole day—where January 1, 1900, at noon has the date serial number 2.5, and where negative numbers represent dates prior to December 30, 1899 (0).

**Tip** TimeSerial can return a new time based on calculations done on a given time. In the following example, TimeSerial returns a time that is thirty minutes before 11:45:00 A.M. and assigns it to the variable T:

```
T = TimeSerial(11, 45, 00) - ((30/60)/24)
```

**Example** The following example displays the number of hours, minutes, and seconds remaining until midnight:

```
Sub Start()
  Dim HrDiff As Integer, MinDiff As Integer, SecDiff As Integer
  Dim RightNow As Double, Midnight As Double
  Dim TotalDiff As Double, TotalMinDiff As Double
  Dim TotalSecDiff As Double, Msg As String
  Super::Start( )
  Midnight = TimeValue("23:59:59")
' Get current time
  RightNow = Now
  ' Get diffs from midnight
  HrDiff = Hour(Midnight) - Hour(RightNow)
  MinDiff = Minute(Midnight) - Minute(RightNow)
  SecDiff = Second(Midnight) - Second(RightNow) + 1
  ' Restate seconds and minutes if necessary
  If SecDiff = 60 Then
     ' Add 1 to minute
     MinDiff = MinDiff + 1
     ' And set 0 seconds
     SecDiff = 0
  End If
  If MinDiff = 60 Then
     ' Add 1 to hour
     HrDiff = HrDiff + 1
     ' And set 0 minutes
     MinDiff = 0
  End If
  ' Now get totals
  TotalMinDiff = (HrDiff * 60) + MinDiff
  TotalSecDiff = (TotalMinDiff * 60) + SecDiff
  TotalDiff = TimeSerial(HrDiff, MinDiff, SecDiff)
  ' Prepare msg for display
  Msg = "There are a total of " & Format(TotalSecDiff, "#, ##0")
  & " seconds until midnight. That translates to "
+
   & HrDiff & " hours, "
+
+
    & MinDiff & " minutes, and "
   & SecDiff & " seconds. "
+
     & "In standard time notation, it becomes "
  ' Remember not to use "mm" for minutes! m is for month.
  Msg = Msg & Format(TotalDiff, "hh:nn:ss") & "."
  ShowFactoryStatus ( Msg )
End Sub
```

See also DateSerial function DateValue function Hour function Now function Second function TimeValue function

### **TimeValue function**

Returns a Date variant representing a time of day based on a specified string.

**Syntax** TimeValue(<time string>)

### Parameters <time string>

String expression that specifies a time. Can be any string that can be interpreted as a time of day. You can enter valid times using a 12- or 24-hour clock.

- <time string> must be in the range 00:00:00 (12:00:00 A.M.) through 23:59:59 (11:59:59 P.M.).
- If <time string> includes a date, it must be a valid one, even though TimeValue does not return a date.
- <time string> is parsed according to the formatting rules of the current runtime locale.

For examples, the following statements are equivalent. Each stores 1:25:00 P.M. to the variable TimeVar.

```
TimeVar = TimeValue("1:25PM")
TimeVar = TimeValue("1:25 PM")
TimeVar = TimeValue("01:25 pm")
TimeVar = TimeValue("13:25")
```

Each of the following statements stores the underlying serial number for 1:25:00 P.M., which is about .559028, to the double-precision variable TimeVar#:

```
TimeVar# = CDbl(TimeValue("1:25PM"))
TimeVar# = TimeValue("1:25PM") * 1
```

### Returns Date

 TimeValue return value looks like a time but is stored internally as a doubleprecision fractional number between 0 and .99999. This number represents a time between 00:00:00 and 23:59:59, or 12:00:00 A.M. and 11:59:59 P.M., inclusive.

In other contexts, this fractional number becomes part of what is known as a date serial number—a number that represents a date and/or time from midnight January 1, 100, through December 31, 9999, inclusive.

- The integer component of any date serial number represents the date (day, month, and year) while the decimal or fractional component represents the time of day on that date as a proportion of a whole day—where January 1, 1900, at noon has the date serial number 2.5, and where negative numbers represent dates prior to December 30, 1899 (0).
- **Example** The following example displays the number of hours, minutes, and seconds remaining until midnight:

```
Sub Start()
  Dim HrDiff As Integer, MinDiff As Integer, SecDiff As Integer
  Dim RightNow As Double, Midnight As Double
  Dim TotalDiff As Double, TotalMinDiff As Double
  Dim TotalSecDiff As Double, Msg As String
  Super::Start( )
  Midnight = TimeValue("23:59:59")
' Get current time
  RightNow = Now
  ' Get diffs from midnight
  HrDiff = Hour(Midnight) - Hour(RightNow)
  MinDiff = Minute(Midnight) - Minute(RightNow)
  SecDiff = Second(Midnight) - Second(RightNow) + 1
  ' Restate seconds and minutes if necessary
  If SecDiff = 60 Then
     ' Add 1 to minute
     MinDiff = MinDiff + 1
     ' And set 0 seconds
     SecDiff = 0
  End If
  If MinDiff = 60 Then
     ' Add 1 to hour
     HrDiff = HrDiff + 1
     ' And set 0 minutes
     MinDiff = 0
  End If
  ' Now get totals
  TotalMinDiff = (HrDiff * 60) + MinDiff
  TotalSecDiff = (TotalMinDiff * 60) + SecDiff
  TotalDiff = TimeSerial(HrDiff, MinDiff, SecDiff)
  ' Prepare msg for display
  Msg = "There are a total of " & Format(TotalSecDiff, "#,##0")
   & " seconds until midnight. That translates to "
+
   & HrDiff & " hours, "
+
   & MinDiff & " minutes, and "
+
   & SecDiff & " seconds. "
+
   & "In standard time notation, it becomes "
```

' Remember not to use "mm" for minutes! m is for month. Msg = Msg & Format(TotalDiff, "hh:nn:ss") & "." ShowFactoryStatus( Msg ) End Sub

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also DateSerial function DateValue function Hour function Now function Second function TimeSerial function

### Trim, Trim\$ functions

Returns a copy of a string after removing leading and trailing spaces.

Syntax Trim(<string expression>)

Trim\$(<string expression>)

### Parameters <string expression>

String expression from which Trim[\$] strips leading and trailing spaces. Leading spaces are any spaces that occur before the first non-space character in a string. Trailing spaces are any spaces that occur after the last non-space character in a string.

- Returns Trim: Variant Trim\$: String
  - If there are no leading or trailing spaces, Trim[\$] returns the original <string expression>.
  - Trim[\$] has no effect on spaces in the middle of a string.
  - If <string expression> evaluates to Null, Trim[\$] returns Null.
  - **Tips** To strip extraneous spaces a user added in response to a prompt, use Trim[\$].
    - To find other spaces in the middle of a string, use InStr.
- **Example** The following example strips leading and trailing spaces from a string variable. It shows a typical situation in which you use Trim[\$] to remove extraneous spaces that occur in data or when a user enters a value in response to a prompt.

```
Sub Start()
  Dim Msg, NL, CustName, CustName1, LeftSpcs, RightSpcs
Super::Start( )
  CustName = " Customer Name
                                 п
  ' Number of leading spaces
  LeftSpcs = Len( CustName ) - Len(LTrim$( CustName ))
  ' Number of trailing spaces
  RightSpcs = Len( CustName ) - Len(RTrim$( CustName ))
  ' Strip left and right spaces
CustName1 = Trim$( CustName )
  Msg = "The original name " & "'" & CustName & "'"
    & " was " & Len( CustName ) & " characters long. "
+
     & "There were " & LeftSpcs & " leading spaces and "
+
    & RightSpcs & " trailing spaces. "
+
    & "The name after stripping the spaces is: "
+
    & "'" & CustName1 & "'" & " which is only "
+
     & Len( CustName1 ) & " characters."
  ShowFactoryStatus( Msg )
End Sub
```

See also Format, Format\$ functions LTrim, LTrim\$ functions RTrim, RTrim\$ functions

### Type...End Type statement

Declares a user-defined variable or record structure.

Syntax Type <user-defined variable type>

<component variable> [(<subscripts>)] As <data type>

[ <component variable> [(<subscripts>)] As <data type>]

. . .

End Type

# Parameters user-defined variable type> The name of a user-defined data type. <user-defined variable type> must follow

The name of a user-defined data type. <user-defined variable type> must follow standard variable-naming conventions.

### <component variable>

The name of an element of the user-defined data type. <component variable> must follow standard variable-naming conventions.

### <subscripts>

The dimensions of an array element. Each <subscripts> argument specifies the number of elements in a single dimension. The number of <subscripts> arguments for an array variable specifies the number of dimensions in the array.

The syntax of <subscripts> is as follows:

[<lower> To]<upper>[,[<lower> To]<upper>]...

The following conditions apply to <subscripts>:

- <lower> and <upper> can range from -2,147,483,648 to +2,147,483,647 inclusive.
- <lower> must always be less than <upper>.
- <lower> and <upper> cannot be the names of variables. They must be literal numeric constants.
- Each <subscripts> clause can specify up to 60 dimensions.

You cannot use Type to declare dynamic arrays. However, you can declare the dimensions for an existing dynamic array within Type.

#### <data type>

The name of a valid data type.

The following conditions apply to <data type>:

- Can be any valid Actuate Basic data type except an object type, including String, for variable-length strings.
- Can be another user-defined type.

For example, the following three arrays are equivalent if you do not use Option Base:

```
MyArray(7,5)
MyArray(0 TO 7, 0 TO 5)
MyArray(7, 0 TO 5)
```

The following shows the use of array variables in the context of Type:

```
Declare

Type CustomerData

CustName As String

CustAddress As String

ArrayPhones(5) As String

ArrayOther(5 To 10, 31) As Double

End Type

End Declare
```

Rules:

• You cannot use line numbers or line labels in Type.

- When declaring a static array within Type, you must use literal numeric constants rather than variables to declare its dimensions.
- **Tip** To more easily manipulate data records that consist of a number of related elements of different data types, use Type to define your own custom data types.

**Example** The following example creates a user-defined type with three elements. Then, it declares a new variable of the user-defined type and assigns data to each element.

```
Declare
  Type AddressStruct
     StreetAddr As String
     City As String
     State As String
  End Type
End Declare
Function VerifyAddress (NewAddr As AddressStruct) As Integer
  If IsNull(NewAddr.StreetAddr) Or (NewAddr.StreetAddr = "")
+ Or IsNull(NewAddr.City) Or (NewAddr.City = "")
+ Or IsNull(NewAddr.State) Or (NewAddr.State = "") Then
     VerifyAddress = 0
  Else
     VerifyAddress = 1
  End If
End Function
Sub Start()
  Dim Msg As String, NL As String
  Dim Address1 As AddressStruct, Address2 As AddressStruct
  Dim TestResult1 As String, TestResult2 As String
  Super::Start( )
  Address1.StreetAddr = "1024 Lexington Drive"
  Address2.StreetAddr = "40000 New County Road"
  Address2.City = "Redwood Hills"
  Address2.State = "CA"
  TestResult1 = IIf(VerifyAddress(Address1), "", "not ")
  TestResult2 = IIf(VerifyAddress(Address2), "", "not ")
  Msg = "Address 1: " & Address1.StreetAddr & ", "
  & Address1.City & ", " & Address1.State & " is "
+
     & TestResult1 & "a valid address. "
+
  ShowFactoryStatus ( Msg )
  Msg = "Address 2: " & Address2.StreetAddr & ", "
     & Address2.City & ", " & Address2.State & " is "
+
     & TestResult2 & "a valid address."
  ShowFactoryStatus ( Msg )
End Sub
```

See also Class statement Declare statement Type...As statement

### Type...As statement

Declares an alias for an existing data type.

Syntax Type <alias data type name> As <existing data type name>

#### Parameters <alias data type name>

The name you assign as an alias for an existing data type. <alias data type name> must follow standard naming conventions.

#### <existing data type name>

The name of an existing, valid data type. <existing data type name> must follow standard naming conventions.

**Example** The following example shows how to use Type As:

Declare

Type SampleRec
ClientNum As Long
ClientName As String
ClientPurchase As Currency
End Type
Type OtherName As SampleRec
Type LogicalVar As Boolean
End Declare
Sub Start( )
Dim UserVar1 As OtherName, IsValid As LogicalVar
Dim Msg As String, UserAns
Super::Start()
UserVar1.ClientName = "Samita Jain"
Msg = "Client name is: " & UserVar1.ClientName
ShowFactoryStatus ( Msg )
End Sub

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Class statement Declare statement Type...End Type statement

### **UBound function**

Returns the largest available subscript for the given dimension of an array.

Syntax UBound(<array name>[,<dimension>])

### Parameters <array name>

Name of an array variable.

### <dimension>

Numeric expression that specifies the array dimension for which you want to determine the upper bound. The default is 1.

The following conditions apply to <dimension>:

- Use 1 for the first dimension, 2 for the second, and so on.
- Must not be larger than the total number of dimensions in <array name>.
- Must not be zero, negative, or Null.

The following example shows the values UBound returns for an array with these dimensions:

```
Dim MyArray(1 To 55, 0 To 27, -3 To 42)
UBound(MyArray, 1) = 55
UBound(MyArray, 2) = 27
UBound(MyArray, 3) = 42
```

### Returns Integer

If <dimension> is not an Integer, UBound rounds it to the nearest Integer before evaluating it.

- **Tips** To determine the lower bound of an array, use LBound.
  - To determine the total number of elements in a given dynamic array dimension, take the value returned from UBound, subtract the value returned from LBound, then add 1.
- **Example** The following example uses UBound to determine the upper bounds of an array of 3 dimensions. The use of Rnd simulates changes in upper bounds that the user made at run time in response to the program.

```
Sub Start( )
  Dim First As Integer, Sec As Integer, Third As Integer
  Dim Msg As String
  ' Declare array variable
  Dim MyArray()
  Super::Start( )
  ' Seed rnd generator
  Randomize
```

```
' First dimension
  First = Int(14 * Rnd + 3)
  ' Second dimension
  Sec = Int(14 * Rnd + 3)
  ' Third dimension
  Third = Int(14 * Rnd + 3)
  ' Set dimensions
  ReDim MyArray (First, Sec, Third)
  Msg = "MyArray has the following upper bounds: "
     & "Dimension 1 -> " & UBound (MyArray, 1)
+
     & "Dimension 2 -> " & UBound (MyArray, 2)
+
     & "Dimension 3 -> " & UBound (MyArray, 3)
  ShowFactoryStatus ( Msg )
End Sub
```

See also Dim statement LBound function Rnd function

# UCase, UCase\$ functions

Converts all lowercase alphabetic characters in a string to uppercase.

Syntax UCase(<string exprs>)

UCase\$(<string exprs>)

### Parameters <string exprs>

String expression that contains the characters to convert to uppercase.

For example, the following statements are equivalent. Each returns 6 JANE STREET, 2ND FL.

UCase\$("6 jane street, 2nd fl") UCase\$("6 Jane Street, 2nd Fl") UCase\$("6 jAn" & "E sTreeT, 2nD fL")

# **Returns** UCase: Variant

UCase\$: String

- If <string exprs> contains no lowercase alphabetic characters, UCase[\$] returns the original <string exprs>.
- UCase[\$] has no effect on non-alphabetic characters in <string exprs>.
- If <string exprs> evaluates to Null, UCase[\$] returns Null.

- **Tips** To ensure uniformity in the data you get from the user so that, for example, strings like MacManus, Macmanus, and macMaNus are always sorted, compared, or otherwise treated in the same way, use UCase[\$] to convert the data before using it.
  - To convert alphabetic characters in <string exprs> to all lowercase characters, use LCase[\$].
- **Example** The following example uses UCase[\$] to render a customer's name in uppercase letters:

```
Sub Start( )
Dim AnyCase As String, Uppercase As String, Msg As String
Super::Start( )
AnyCase = "CusTomer naME"
' Convert to uppercase
Uppercase = UCase$(AnyCase)
Msg = "UCase$ converts """ & AnyCase & """ to """
+ & Uppercase & """."
ShowFactoryStatus( Msg )
End Sub
```

**See also** LCase, LCase\$ functions

## Val function

Returns the numeric value of a string expression.

Syntax Val(<string exprs>)

### Parameters <string exprs>

String expression from which to extract a numeric value.

For example, the following are equivalent. They both return the value 118.

```
Val("118 36th Street")
Val("118 36th Street, #55")
```

The following returns the value 11836:

```
Val("&H2E3C")
```

Because the first character is non-numeric, the following returns 0:

Val("\$1,275,332.52")

Rules:

 Starts translating a string into a number by starting at the left of the string and proceeding toward the right.
- Stops translating the string as soon as it encounters a non-numeric character or the end of the string, whichever comes first.
- Interprets the prefixes &O and &H as numeric characters representing octal and hexadecimal numbers, respectively.
- Interprets dollar sign as non-numeric.
- Ignores spaces, tabs, thousands separators, and line feeds.
- Stops translating at the appearance of the second decimal point.
- Returns zero if it stopped before it encounters a numeric character.
- Parses data according to the formatting rules of the run-time locale. Table 6-57 shows how Val parses strings in different locales.

String	US English	Greek	French
123.456	123.456	123456	123,456
123,456	123456	123,456	123,456
123.456.789	123.456	123456789	123,456
123,456,789	123456789	123,456	123,456

 Table 6-57
 Comparison of how Val parses strings in different locales

- Interprets the first decimal separator in a numerical string as a decimal separator and stops conversion at the second decimal separator. Both "123.456" and "123.456.789" convert to 123.456 in US English. In the French locale, Val converts "123.456" to 123,456. Similarly, Val converts "123.456.789" in the French locale.
- In a locale that uses a dot as a thousands separator, Val ignores any dots it encounters. In the Greek locale, for example, Val converts "123.456" to 123456 and converts 123.456.789 to 123456789.
- In a locale that uses a comma as a thousands separator, Val ignores the comma.
   For example, in the US English locale, Val converts "123,456" to 123456.
- In a locale that does not use a dot as a thousands separator or decimal separator, Val interprets a dot as a decimal separator.

## Returns Double

If <string exprs> evaluates to Null, Val returns Null.

- **Tips •** To convert a number to a String use Str\$.
  - To avoid using Val, use the Variant data type for variables when it is necessary to convert between strings and numbers.
- **Example** The following example creates a string, then returns the numeric value of that string:

```
Sub Start( )
   Dim Msg As String, Number As Double
   Super::Start( )
   ' A string containing some numbers
   Number = Val( "987,123.654" )
   Msg = "VAL extracted this value from the input: " & Number
   ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Format, Format\$ functions Str, Str\$ functions

# VarType function

Returns a value that indicates how Actuate Basic stores a Variant internally.

**Syntax** VarType(<variant variable>)

## Parameters <variant variable>

String expression that specifies a variant of the Variant data type.

## Returns Integer

Table 6-58 shows the values VarType returns, the Variant data types they indicate, and the names of the corresponding symbolic constants that are stored in Header.bas.

	types and symbolic constants	Table 6-58         VarType return values and the corresponding <variant variable=""> data</variant>
--	------------------------------	---

VarType returns	Data type of <variant variable=""></variant>	Symbolic constant
0	Empty	V_EMPTY
1	Null	V_NULL
2	Integer	V_INTEGER
3	Long	V_LONG
4	Single	V_SINGLE
5	Double	V_DOUBLE
6	Currency	V_CURRENCY
7	Date	V_DATE
8	String	V_STRING

If <variant variable> evaluates to Null, VarType returns Null.

**Example** The following example displays the Variant data type for a date, number, and string:

```
Sub Start()
  Dim TypeData
  Super::Start( )
  TypeData = "This is data"
  ShowFactoryStatus( "The value """ & TypeData &
     """ is of VarType: " & VarType( TypeData ) )
+
  TypeData = 42
  ShowFactoryStatus( "The value """ & TypeData &
     """ is of VarType: " & VarType( TypeData ) )
+
  TypeData = CVDate("12/10/59")
  ShowFactoryStatus ( "The value """ & TypeData &
     """ is of VarType: " & VarType( TypeData ) )
+
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also IsDate function IsEmpty function IsNull function IsNumeric function

# Weekday function

Returns an integer between 1 (for Sunday) and 7 (for Saturday) that represents the day of the week for a specified date argument.

Syntax Weekday(<date exprs>)

## Parameters <date exprs>

Date expression, or any numeric or string expression that can be interpreted as a date, or both a date and a time:

- Can be a string such as November 12, 1982 8:30 PM, Nov. 12, 1982 08:30 PM, 11/12/82 8:30pm, or any other string that can be interpreted as a date or both a date and a time in the valid range.
- Can be a date serial number such as 30267.854, which corresponds to November 12, 1982 at 8:30 P.M., or any other number that can be mapped to a date or both a date and a time in the valid range.
- For date serial numbers, the integer component represents the date itself while the fractional component represents the time of day on that date, where January 1, 1900, at noon has the date serial number 2.5, and negative numbers represent dates prior to December 30, 1899 (0).

The following conditions apply to <date exprs>:

- If <date exprs> includes a time, it must be a valid time, even though Weekday does not return a time. A valid time is one that is in the range 00:00:00 (12:00:00 A.M.) through 23:59:59 (11:59:59 P.M.), in either 12- or 24-hour format.
- If <date exprs> is a numeric expression, it must be in the range -657434 to +2958465, inclusive.
- If <date exprs> is a variable containing a date serial number, the variable must be explicitly declared as one of the numeric types.
- <date exprs> is parsed according to the formatting rules of the current run-time locale.

The following assumes that the AC\_CENTURY\_BREAK property is set to the default value of 30. For details on how to set AC\_CENTURY\_BREAK to some other value, please see *Accessing Data using e.Report Designer Professional*.

- For abridged date expressions ending in 00 through 29 that do not specify the century, relevant date functions estimate the unabridged year by adding 2000. For example, while Year(#4/30/1910#) returns 1910, the abridged expression Year(#4/30/10#) returns 2010 (10 + 2000).
- For abridged date expressions ending in 30 through 99 that do not specify the century, relevant date functions estimate the unabridged year by adding 1900. For example, while Year(#11/12/2082#) returns 2082, the abridged expression Year(#11/12/82#) returns 1982 (82 + 1900).

Note that a date can contain an abridged month, such as Nov. 12, 1982 as opposed to November 12, 1982.

For example, the following statements are equivalent. Each assigns 1 (which maps to Sunday) to the variable DayOfWeek.

```
DayOfWeek = Weekday("6/7/64 2:35pm")
DayOfWeek = Weekday("June 7, 1964 2:35 PM")
DayOfWeek = Weekday("Jun " & 5+2 & ", 1964")
DayOfWeek = Weekday(23535.6076)
DayOfWeek = Weekday(0.6076) - 6
```

## Returns Integer

- If <date exprs> is Null, Weekday returns Null.
- If Weekday returns 1, the day of the week is Sunday. If Weekday returns 2, the day if the week is Monday, and so on.
- If <date exprs> cannot be evaluated to a date, Weekday returns Null. Example:

```
Weekday("This is not a date.") returns Null
```

 If <date exprs> fails to include all date components (day, month, and year), Weekday returns Null.
 Examples:

```
Weekday("Nov 12, 1982") returns 6, but
Weekday("Nov 1982") returns Null
```

- **Tip** If you use a date expression in one locale, it might be misinterpreted in another locale. For instance, in the United States, 1/2/2005 means January 2nd, 2005, but in France, it means February 1st, 2005. To avoid such ambiguities, use DateSerial to specify all your dates.
- **Example** The following example displays a message indicating whether today is a weekend day or a weekday:

```
Sub Start( )
Dim DayNum As Integer, Msg As String, FullDesc as String
Super::Start( )
' Get current day of week
DayNum = Weekday( Now )
FullDesc = Format( Now, "dddd, mmmm dd, yyyy" )
' Is it a weekend day?
If DayNum = 1 Or DayNum = 7 Then
Msg = "Today is " & FullDesc & ". It is a weekend day."
'... or is it a weekday?
Else
Msg = "Today is " & FullDesc & ". It is a weekday."
End If
ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Day function Hour function Minute function Month function Now function Second function Year function

# While...Wend statement

Repeats a block of instructions each time a specified condition is True.

Syntax While <expression to reevaluate>

#### <statements to repeat>

#### Wend

**Description** First, <expression to reevaluate> is tested. If it is True, the <statements to repeat> in the body of the loop are executed. Then <expression to reevaluate> is tested again. If it is still True, <statements to repeat> are executed again. This process repeats until <expression to reevaluate> is False. In that case, Actuate Basic skips all intervening <statements to repeat> and passes control to the statement immediately following Wend.

Do not branch into the middle of a While...Wend statement. Execute the entire statement instead, starting at the line with the While keyword.

#### Parameters <expression to reevaluate>

Any valid numeric or string expression that evaluates to True (1) or False (0 or Null).

#### <statements to repeat>

One or more valid Actuate Basic statements; the statements that, collectively taken, are referred to as the loop.

For example, assuming the variable Counter starts at 1 and is incremented by 1 each time through the loop, the following statement causes its associated loop to be executed nine times:

While Counter < 10

- **Tips** To take advantage of its greater flexibility, use Do…Loop instead of While Wend.
  - To be sure the loop executes at least once, explicitly set <expression to reevaluate> to True in a statement that closely or immediately precedes the While...Wend statement.
  - It is good programming practice to evaluate Boolean variables by using the keywords True or False instead of by inspecting their content for a nonzero (True) or zero (False) numeric value.

#### **Example** The following example sorts an array:

```
Sub Start( )
Dim SwapThese, Position As Integer, TempVar
Static Cities(5)
Super::Start( )
'Now assign array data out of alphabetical order
'so we can see sort illustrated.
Cities(1) = "New York"
Cities(2) = "San Francisco"
Cities(3) = "Paris"
Cities(4) = "Boston"
Cities(5) = "Seattle"
```

```
' Make sure we loop at least once
  SwapThese = True
   ' Sort while there still are elements we need to swap
  While SwapThese
   ' Now compare array elements by pairs. When two are swapped,
   ' ensure another pass thru loop by setting SwapThese to TRUE:
     SwapThese = False
     For Position = 2 To 5
        If Cities (Position - 1) > Cities (Position) Then
          SwapThese = True
          ' Now do the swap:
          TempVar = Cities (Position)
          Cities (Position) = Cities (Position - 1)
          Cities(Position - 1) = TempVar
        End If
     Next Position
  Wend
  ShowFactoryStatus("New, sorted order: ")
  For Position = 1 To 5
     ShowFactoryStatus( Cities( Position ) )
  Next Position
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Do...Loop statement If...Then...Else statement

## Width statement

Defines the width of the output line of a sequential file.

- Syntax Width #<open file number>, <new line width>
- **Description** When you print to an ASCII text file, you might need to define a maximum output length for each line. Use the Width statement for this operation.

#### Parameters <open file number>

Numeric expression that is the file number for a file opened with the Open statement. The file number must refer to a file that is currently open.

#### <new line width>

Numeric expression that is the new width for the line. Width is measured as the number of characters Actuate Basic prints on a line before it begins a new line. If <new line width> is specified as 0, there is no limit to the length of the line. <new line width> must be in the range of 0–255, inclusive. The default is 0.

For example, the following statement sets the output line width to 75 columns for open file #1:

Width #1, 75

**Example** The following example creates a test file and writes some ASCII characters to it using the default output width of unlimited. Then, it changes the width and appends the same ASCII characters to the file. Finally, it reads the entire file back and displays it with both sets of ASCII characters, then deletes the file.

```
Sub Start()
  Dim I As Integer, Msg As String, TextLines As String
Super::Start( )
  ' Create a sample text file
  Open "Testfile.txt" For Output As #1
  ' Print ASCII characters . . . 0-9 all on same line
  Width #1, 10
  Print #1, "Width 10:"
  For I = 0 To 9
     Print #1, Chr$(48 + I)
  Next I
' Start a new line
  Print #1,
  ' Change the line width to 4
  Width #1, 4
  ' Print ASCII characters . . . 4 characters to a line
  For I = 0 To 9
     Print #1, Chr$(48 + I)
  Next I
  Close #1
  ShowFactoryStatus( "The effect of WIDTH is as displayed: " )
  ' Reopen test file for input
  Open "Testfile.txt" For Input As #1
  Do While Not EOF(1)
' Get contents of each line
     Input #1, TextLines
     ShowFactoryStatus ( TextLines )
  Loop
  Close #1
ShowFactoryStatus( "Test file will now be deleted.")
  Kill "Testfile.txt"
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Open statement Print statement Space, Space\$ functions Tab function

# Write statement

Formats and writes data to a sequential file.

Syntax Write #<file number> [, <exprs 1>] [, <exprs 2>] [, <exprs 3>]...

**Description** When Write writes to the file it inserts commas between items and places quotation marks around strings. You do not need to put explicit delimiters in the list. Write also inserts a newline character after it writes the final <exprs> to the file.

Write usually writes Variant data to a file the same way it writes data of any other Actuate Basic data type. Table 6-59, however, notes certain exceptions.

 Table 6-59
 Exceptions to how Write writes Variant data to a file

Data	Write writes this to the file
One Variant of type V_EMPTY (Empty)	Nothing at all
More than one Variant of type V_EMPTY	Delimiting commas
Variant of type V_NULL (Null)	Literal #NULL#
Variant of type V_DATE (Date)	The date, using the fixed date format: #yyyy-mm-dd hh:nn:ss#
Variant of type V_DATE, but with either the date or time component missing or zero	Only the part of the date provided

- <file number> must match the number of a currently open file.
- The file corresponding to <file number> must be open under either the Output or Append mode.
- You must have write access to the open file. That is, the file must not have been opened using a Lock Write or Lock Read Write clause.
- The data cannot be an object variable, user-defined data type structure, handle to a class, CPointer, or OLE object.

## Parameters <file number>

Numeric expression that is the file number of a sequential file that is Open for Output or Append.

For example, the following statements work together, although they can be separated from one another by lines of code. The first statement opens a file called Mynotes.fil, the second one writes the contents of four variables to that file, and the third, by closing it, turns the file over to the operating system to complete the process of writing it to the disk.

```
Open "C:\Myfiles\Mynotes.fil" FOR OUTPUT AS #2
Write #2, A$, B%, C!, D#
Close #2
```

The following statement writes the specified variables to the file opened by a previous Open statement under number 1. Actuate Basic places double quotation marks around the value in StringVar\$, and it inserts commas between all the variables to delimit them. Finally, after the value in DoubleVar#, it writes a newline character.

Write #1, StringVar\$, IntVar%, SingleVar!, DoubleVar#

#### <exprs n>

Numeric and/or string expression that specifies the data that Write places in the file. There can be any number of these. The default is the newline character.

- **Tip** Many database programs let you import data when the data is structured in the way that Write structures it. Such programs interpret each newline character as delimiting the end of a complete record, and treat each comma on a line as delimiting the end of a field within that record. This means you can use Write to export information from Actuate Basic to almost any database program.
- **Example** The following example creates a name and age. It then writes this information to a test file, reads the file back, places the data in a message, and displays the message to the user. After the user responds, the example deletes the file.

```
Sub Start()
  Dim Msg As String
  Dim UsersAge As Integer, UsersName As String
  Super::Start( )
  ' Now create a sample data file:
  ' Open the file for output
  Open "Test.fil" For Output As #1
  UsersName = "Xudong Ho"
  UsersAge = CInt(Rnd * 25) + 20
  ' Write data to test file
  Write #1, UsersName, UsersAge
  ' Close file
  Close #1
  ' Now read back the sample data file:
   ' Open test file for input
  Open "Test.fil" For Input As #1
   ' Read the data in it
  Input #1, UsersName, UsersAge
```

```
' Close file
Close #1
Msg = "The name '" & UsersName & "' was read from the file. "
+ & "'" & UsersAge & "' is the age."
+ & "The test file will now be deleted."
ShowFactoryStatus( Msg )
' Delete file from disk
Kill "Test.fil"
End Function
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Open statement Print statement

# Year function

Returns an integer between 100 and 9999, inclusive, that represents the year of a specified date argument.

Syntax Year(<date exprs>)

## Parameters <date exprs>

Date expression, or any numeric or string expression that can evaluate to a date. Specifies a date and/or time.

- Can be a string such as November 12, 1982, Nov. 12, 1982, 11/12/82, 11-12-82, or any other string that can be interpreted as a date in the valid range.
- Can be a date serial number such as 30267.854, which corresponds to November 12, 1982 at 8:30 P.M., or any other number that can be mapped to a date in the valid range.
- For date serial numbers, the integer component represents the date itself while the fractional component represents the time of day on that date, where January 1, 1900 at noon has the date serial number 2.5, and negative numbers represent dates prior to December 30, 1899 (0).
- If <date exprs> is a string expression, must specify a date in the range January 1, 100 through December 31, 9999, inclusive.
- If <date exprs> is a numeric expression, must be in the range -657434 to +2958465, inclusive.
- If <date exprs> is a variable containing a date serial number, the variable must be explicitly declared as one of the numeric types.
- <date exprs> is parsed according to the run-time locale formatting rules.

The following assumes that the AC\_CENTURY\_BREAK property is set to the default value of 30. For information about using AC\_CENTURY\_BREAK, please see *Accessing Data using e.Report Designer Professional*.

- For abridged date expressions ending in 00 through 29 that do not specify the century, relevant date functions estimate the unabridged year by adding 2000. For example, while Year(#4/30/1910#) returns 1910, the abridged expression Year(#4/30/10#) returns 2010 (10 + 2000).
- For abridged date expressions ending in 30 through 99 that do not specify the century, relevant date functions estimate the unabridged year by adding 1900. For example, while Year(#11/12/2082#) returns 2082, the abridged expression Year(#11/12/82#) returns 1982 (82 + 1900).

Note that a date can contain an abridged month, such as Nov. 12, 1982 as opposed to November 12, 1982.

For example, the following statements are equivalent. Each assigns 1964 to the variable UserYear.

```
UserYear = Year("6/7/64")
UserYear = Year("June 7, 1964 2:35 PM")
UserYear = Year("Jun 7, 1964")
UserYear = Year(23535)
UserYear = Year(4707*5)
```

## Returns Integer

- If <date exprs> is Null, Year returns Null.
- If <date exprs> cannot be evaluated to a date, Year returns Null. For example:

Year("This is not a date.") returns Null

 If <date exprs> fails to include all date components (day, month, and year), Year returns Null. For example:

```
Year("Nov 12, 1982") returns 1982, but
Year("Nov 1982") returns Null
```

- **Tip** If you use a date expression in one locale, it might be misinterpreted in another locale. For instance, in the United States, 1/2/2005 means January 2nd, 2005, but in France, it means February 1st, 2005. To avoid such ambiguities, use DateSerial to specify all your dates.
- **Example** The following example generates a date. Then, it uses various date functions to display the year, month, day, and weekday of the that date. Finally, it gives the date's serial number.

' This function tells us what suffix (1st, 2nd, 3rd) to use ' with a number Function Suffix (DateNum) As String

```
Select Case DateNum
     Case 1, 21, 31
       Suffix = "st"
     Case 2, 22
       Suffix = "nd"
     Case 3, 23
       Suffix = "rd"
     Case Else
       Suffix = "th"
  End Select
End Function
Sub Start()
  Dim Date1 As Date, UserYear As Integer, UserMonth As Integer
  Dim UserDay As Integer, UserDOW As Integer, DOWName As String
  Dim Msg As String, LeapYear As String
  Super::Start( )
  ' Get a date
  Date1 = Dateserial(Rnd * 120 + 1899, 1, 1) + CInt(Rnd * 366)
  ' Calculate year
  UserYear = Year(Date1)
  ' Calculate month
  UserMonth = Month(Date1)
  ' Calculate day
  UserDay = Day(Date1)
  ' Calculate day of week
  UserDOW = Weekday(Date1)
  ' Convert to name of day
  DOWName = Format$ (Date1, "dddd")
  ' Determine if the year is a leap year or a centesimal
  If UserYear Mod 4 = 0 And UserYear Mod 100 = 0 Then
     ' Evenly divisible by 400?
     If UserYear Mod 400 = 0 Then
       LeapYear = "is a leap year."
     ' Not evenly divisible
     Else
       LeapYear = "is a centesimal but not a leap year."
     End If
  ElseIf UserYear Mod 4 = 0 Then
     LeapYear = "is a leap year."
  Else
     LeapYear = "is not a leap year."
  End If
```

' Display results for the user after the pattern: ' The given year 1982 is not a leap year. The weekday number ' for the 12th day of the 11th month in 1982 is 6, which ' means that day was a Friday. ' The date serial number for 11/12/82 is: 30267 Msg = "The given year, " & UserYear & ", " & LeapYear ShowFactoryStatus ( Msg ) Msg = "The weekday number for the " & UserDay & Suffix(UserDay) & " day of the " + & UserMonth & Suffix(UserMonth) & " month in " + + & UserYear & " is " & UserDOW & ", which means " & "that day is a " & DOWName & ". " + & "The date serial number for " & Date1 & " is: " + & CDbl(DateSerial(UserYear, UserMonth, UserDay)) + ShowFactoryStatus( Msg ) End Sub

For information about using the code examples, see "Using the code examples," earlier in this chapter.

See also Date, Date\$ functions Day function Hour function Minute function Month function Now function Second function Weekday function

## Appendix

# Operators

This appendix contains information about operators Actuate Basic supports.

When several operations occur in an expression, each part is evaluated and resolved in a predetermined order, known as operator precedence. You can use parentheses to override the order of precedence and force parts of an expression to be evaluated before others. Operations within parentheses are always performed before those outside. Within parentheses, normal operator precedence is maintained.

When expressions contain operators from more than one category, arithmetic operators are evaluated first, comparison operators are evaluated next, and logical operators are evaluated last. Within individual categories, operators are evaluated in the order in which they appear in Table A-1.

Arithmetic	Comparison	Logical	Bitwise
Exponentiation(^)	Equality(=)	Not	BNot
Negation(-)	Inequality(<>)	And	BAnd
Multiplication and Division(* and /)	Less than(<)	Or	BOr
Integer division( $\)$	Greater than(>)	XOr	
Modulo arithmetic (Mod)	Less than or Equal to(<=)	Eqv	

 Table A-1
 Order of evaluation of arithmetic, comparison, logical, and bitwise operators

(continues)

operators	(continued)		
Arithmetic	Comparison	Logical	Bitwise
Addition and Subtraction (+ and -)	Greater than or Equal to(>=)	Imp	
String concatenation(&)	Like	Is	

**Table A-1**Order of evaluation of arithmetic, comparison, logical, and bitwise<br/>operators (continued)

All comparison operators have equal precedence, that is, they are evaluated in the left-to-right order in which they appear.

When multiplication and division or addition and subtraction occur together in an expression, each operation is evaluated as it occurs from left to right.

In precedence, the string concatenation operator (&) falls after all arithmetic operators and before all comparison operators. The Like operator, while equal in precedence to all comparison operators, is actually a pattern-matching operator.

# \* operator

Multiplies two numbers.

```
Syntax <result> = <operand1> * <operand2>
```

**Description** The operands can be any numeric expression.

The data type of <result> is usually the same as that of the most precise operand. The order of precision, from least to most precise, is Integer, Long, Single, Double, Currency. The following are exceptions to this order:

- When multiplication involves a Single and a Long, the data type of <result> is converted to a Double.
- When the data type of <result> is a Variant of type 3 (Long), type 4 (Single), or type 7 (Date) that overflows its legal range, <result> is converted to a Variant of type 5 (Double).
- When the data type of <result> is a Variant of type 2 (Integer) that overflows its legal range, <result> is converted to a Variant of type 3 (Long).

If one or both operands are Null expressions, <result> is a Null. Any operand that is empty (VarType0) is treated as 0.

**Example** The following example determines tax on a value using the \* operator to multiply the user-supplied value by the tax rate of 15%:

```
Super::Start()
' Get value
N = CInt(Rnd * 125000 + 20000)
' Calculate tax
Tax = .15 * N
' Display results
Msg = "Tax on " & N & " is " & Tax
ShowFactoryStatus( Msg )
End Sub
```

See also VarType function

## + operator

Sums two numbers.

Syntax <result> = <operand1> + <operand 2>

**Description** Use the & operator for concatenation to eliminate ambiguity and provide self-documenting code.

When you use the + operator, you may not be able to determine whether addition or string concatenation is to occur. If at least one operand is not a Variant, the rules in Table A-2 apply.

Table A-2	Conditions and	the correspo	nding resu	ults for the +	operator
-----------	----------------	--------------	------------	----------------	----------

Condition	Result
Both expressions are numeric data types (Integer, Long, Single, Double, or Currency).	Add
Both expressions are strings.	Concatenate
One expression is a numeric data type and the other is a Variant (other than a Null).	Add
One expression is a String and the other is a Variant (other than a Null).	Concatenate
One expression is a Variant containing Empty.	Return the remaining operand unchanged as <result></result>
One expression is a numeric data type and the other is a String.	A Type Mismatch error occurs

If either operand is a Null (Variant type 1), <result> is a Null.

If both operands are Variant expressions, the VarType of the operands determines the behavior of the + operator in the way that is shown in Table A-3.

Table A-3	Conditions and results for the + operator when both operands are
	Variant expressions

Condition	Result
Both Variant expressions are of type 2-7 (numeric data types).	Add
Both Variant expressions are of type 8 (String).	Concatenate
One Variant expression is of type 2-7 (a numeric data type) and the other is of type 8 (String).	Add

For simple arithmetic addition involving only operands of numeric data types, the data type of <result> is usually the same as the most precise operand.

The order of precision, from least to most precise, is Integer, Long, Single, Double, Currency. The following exceptions are to this order:

- When a Single and a Long are added together, the data type of <result> is converted to a Double.
- When the data type of <result> is a Variant of type 3 (Long), type 4 (Single), or type 7 (Date) that overflows its legal range, <result> is converted to a Variant of type 5 (Double).

If one or both of the expressions are Null expressions, <result> is a Null. If both operands are Empty, <result> is an Integer. If only one operand is Empty, the other operand is returned unchanged as <result>.

**Example 1** The following example adds a number to itself and displays the result:

```
Sub Start()
Dim N As Double, Res As Double, Fmt As String, Msg As String
Super::Start()
Fmt = "#,##0.00"
' Get a number
N = CInt(Rnd * 10000000) / 100)
'Add numbers
Res = N + N
Msg = Format$(N, Fmt) & " plus " & Format$(N, Fmt) & " is "
Msg = Msg & Format$(Res, Fmt)
' Display result
ShowFactoryStatus( Msg )
End Sub
```

**Example 2** The following example uses the + operator to concatenate strings:

```
Sub Start()
  Dim Msg As String
  Super::Start()
  Msg = "Actuate " + "e.Report " + "Designer Professional"
  ' Display result
  ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples" in Chapter 6, "Statements and functions."

See also VarType function

## - operator

Finds the difference between two numbers or indicates the negative value of an operand.

## Syntax 1

<result> = <operand1> - <operand2>

## Syntax 2

-number

**Description** In Syntax 1, the - operator is the arithmetic subtraction operator used to find the difference between two numbers. The operands can be any numeric expression.

The data type of <result> is usually the same as the most precise operand. The order of precision, from least to most precise, is Integer, Long, Single, Double, Currency. The following exceptions are to this order:

- When subtraction involves a String and a Long, the data type of <result> is converted to a Double.
- When the data type of <result> is a Variant of type 3 (Long), type 4 (Single), or type 7 (Date) that overflows its legal range, <result> is converted to a Variant of type 5 (Double).

If one or both operands are Null expressions, <result> is a Null. If an operand is Empty (VarType 0), it is treated as if it were 0.

**Example 1** The following example subtracts a number from 1000 and displays the result:

```
Sub Start()
Dim N As Double, Result As Double, Msg As String
```

```
Super::Start()
' Get a number
N = CInt(Rnd * 200000) / 100
Result = 1000 - N
' Display result
Msg = "1000 minus " & N & " is " & Result
ShowFactoryStatus( Msg )
End Sub
```

**Example 2** The following example uses the - operator to indicate a negative value of a number:

```
Sub Start()
Dim N As Double, MinusN As Double, Msg As String
Super::Start()
' Get a number
N = CInt(Rnd * 200000) / 100
' Use negation operator
MinusN = -N
' Display result
Msg = "Negative " & N & " is " & MinusN
ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples" in Chapter 6, "Statements and functions."

See also VarType function

# / operator

Divides two numbers and returns a floating point result. Syntax <result> = <operand1> / <operand2> Description The operands can be any numeric expression. The data type of <result> is always a Double or a Variant of type 5 (Double). If one or both operands are Null expressions, <result> is a Null. If an operand is Empty (Variant type 0), it is treated as if it were 0. The following example divides 100,000 by 7 and returns 14,285.7142857143: Example Sub Start() ' Declare variable Dim Result As Double, Msg As String Super::Start( ) ' Divide numbers Result = 100000 / 7

```
' Display results
Msg = "100000 divided by 7 is " & Result
ShowFactoryStatus( Msg )
End Sub
```

See also VarType function

## \ operator

Divides two numbers and returns an integer result.

```
Syntax <result> = <operand1> \ <operand2>
```

**Description** The operands can be any numeric expression.

Before division is performed, the operands are rounded to Integer or Long expressions.

The data type <result> is an Integer. Any fractional portion is truncated. However, if any operand is a Null, <result> is also a Null. Any operand that is Empty (Variant type 0) is treated as 0.

**Example** The following example divides 100,000 by 7 using integer division and returns 14,285:

```
Sub Start()
   ' Declare variables
   Dim Result As Double, Msg As String
   Super::Start()
   ' Divide numbers
   Result = 100000 \ 7
   ' Display results
   Msg = "100000 divided by 7 is " & Result
   ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples" in Chapter 6, "Statements and functions."

## ^ operator

Raises a number to the power of an exponent.

Syntax <result> = <number> ^ <exponent>

#### & operator

**Description** The <number> and <exponent> operands can be any numeric expression. However, the <number> operand can be negative only if <exponent> is an integer value. When more than one exponentiation is performed in a single expression, the ^ operator is evaluated as it is encountered from left to right.

Usually, the data type of <result> is a Double or a Variant (type 5). However, if either <number> or <exponent> is a Null expression, <result> is also a Null.

**Example** The following example squares  $(n^2)$  a number:

```
Sub Start()
Dim N As Double, Result As Double, Msg As String
Super::Start()
' Get number
N = CInt(Rnd * 200) - 100
' Square number
Result = N ^ 2
' Display result
Msg = N & " squared is " & Result
ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples" in Chapter 6, "Statements and functions."

**See also** VarType function

## & operator

Forces string concatenation of two operands.

```
Syntax
             <result> = <operand1> & <operand2>
Description
             Whenever an operand is a number, it is converted to a Variant of type 8 (String).
             The data type of <result> is String if both operands are String expressions;
             otherwise, <result> is a Variant of type 8 (String). If both operand are Null
             (type 1), <result> is also Null. However, if only one operand is Null, that operand
             is treated as a zero-length string when concatenated with the other operand. Any
             operand that is Empty (type 0) is also treated as a zero-length string.
  Example
             The following example uses the & operator to concatenate a numeric variable
             (Var1), a string literal containing a space, and a string variable (Var2) that contains
             trombones:
             Sub Start()
                 ' Declare variables
                Dim Var1 As Integer, Var2 As String, Msg As String
                 Super::Start( )
                 Var1 = 76
```

Var2 = "trombones"

```
' Concatenate and display
Msg = Var1 & " " & Var2
ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples," earlier in this chapter.

## And operator

Performs logical conjunction on two expressions.

Syntax <result> = <expr1> And <expr2>

**Description** If, and only if, both expressions evaluate True, <result> is True. If either expression evaluates False, <result> is False. Table A-4 illustrates how <result> is determined.

<expr1></expr1>	And <expr2></expr2>	<result></result>	
True	True	True	
True	False	False	
True	Null	Null	
False	True	False	
False	False	False	
False	Null	False	
Null	True	Null	
Null	False	False	
Null	Null	Null	

 Table A-4
 Results of expressions that use the And operator

**Example** The following example displays a message that depends on the value of variables A, B, and C, assuming that no variable is a Null. If A = 10, B = 8, and C = 6, both expressions evaluate True. Because both expressions are True, the And expression is also True.

```
Sub Start()
    ' Declare variables
    Dim A, B, C, Msg
    Super::Start()
    ' Assign values
    A = 10: B = 8: C = 6
```

```
' Evaluate expressions
If A > B And B > C Then
Msg = "Both expressions are True."
Else
Msg = "One or both expressions are False."
End If
' Display results
ShowFactoryStatus( Msg )
End Sub
```

# **BAnd operator**

Performs a bitwise comparison of identically positioned bits in two numeric expressions.

Syntax <result> = <expr1> BAnd <expr2>

**Description** If, and only if, both expressions evaluate True, <result> is True. If either expression evaluates False, <result> is False. The BAnd operator sets the corresponding bit in <result> according to the truth table, Table A-5.

Bit in <expr1></expr1>	Bit in <expr2></expr2>	<result></result>	
0	0	0	
0	1	0	
1	0	0	
1	1	1	

**Table A-5**Results for the BAnd operator

**Example** The following example displays a message that depends on the value of variables A, B, and C, assuming that no variable is a Null. If A = 10, B = 8, and C = 6, both expressions evaluate True. Because both expressions are True, the BAnd expression is also True.

```
Sub Start()
   ' Declare variables
   Dim A, B, C, Msg
   Super::Start()
   ' Assign values
   A = 10: B = 8: C = 6
   ' Evaluate expressions
   If A > B BAnd B > C Then
        Msg = "Both expressions are True."
```

```
Else
    Msg = "One or both expressions are False."
  End If
  ' Display results
   ShowFactoryStatus( Msg )
End Sub
```

# **BNot operator**

Inverts the bit values of any variable.

Syntax <result> = BNot <expr>

**Description** The BNot operator sets the corresponding bit in <result> according to the truth table, Table A-6.

 Table A-6
 Results for the BNot operator

Bit in <expr></expr>	Bit in <result></result>
0	1
1	0

If an integer variable has the value 0 (False), the variable becomes 1 (True); if it has the value of 1, it becomes 0.

**Example** The following example displays a message that depends on the value of variables A, B, and C, assuming that no variable is a Null. If A = 10, B = 8, and C = 6, the BNot expression evaluates True because A is not equal to B.

```
Sub Start()
    ' Declare variables
    Dim A, B, Msg
    Super::Start()
    ' Assign values
    A = 10: B = 8
    ' Evaluate expression
    If BNot A = B Then
        Msg = "A and B aren't equal."
    Else
        Msg = "A and B are equal."
    End If
    ' Display results
    ShowFactoryStatus( Msg )
End Sub
```

## **BOr operator**

Performs a bitwise comparison of identically positioned bits in two numeric expressions.

Syntax <result> = <expr1> BOr <expr2>

**Description** The BOr operator sets the corresponding bit in <result> according to the truth table, Table A-7.

Bit in <expr1></expr1>	Bit in <expr2></expr2>	<result></result>
0	0	0
0	1	1
1	0	1
1	1	1

 Table A-7
 Results for the BOr operator

**Example** The following example displays a message that depends on the value of variables A, B, and C, assuming that no variable is a Null. If A = 10, B = 8, and C= 6, the left expression is True and the right expression is False. Because at least one comparison expression is True, the BOr expression evaluates to True.

```
Sub Start()
    ' Declare variables
    Dim A, B, C, Msg
    Super::Start()
    ' Assign values
    A = 10: B = 8: C = 11
    ' Evaluate expression
    If A > B BOr B > C Then
        Msg = "One or both comparison expressions are True."
    Else
        Msg = "Both comparison expressions are False."
    End If
    ' Display results
    ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples" in Chapter 6, "Statements and functions."

# **Comparison operators**

Compares two expressions.

Syntax <result> = <expr1> <operator> <expr2>

**Description** Comparison operators, also known as relational operators, compare two expressions. Table A-8 contains a list of the comparison operators and the conditions that determine whether <result> is True, False, or Null.

	-	•		
	Meaning	True if	False if	Null if
<	Less than	expr1 < expr2	expr1 >= expr2	expr1 or expr2= Null
<=	Less than or equal to	expr1 <= expr2	expr1 > expr2	expr1 or expr2= Null
>	Greater than	expr1 > expr2	expr1 <= expr2	expr1 or expr2= Null
>=	Greater than or equal to	expr1 >= expr2	expr1 < expr 2	expr1 or expr2= Null
=	Equal to	expr1 = expr2	expr1 <> expr 2	expr1 or expr2= Null
<>	Not equal to	expr1 <> expr2	expr1 = expr2	expr1 or expr2= Null

 Table A-8
 Comparison operators

When comparing two expressions, you may not be able to easily determine whether the expressions are being compared as numbers or as strings. Table A-9 shows how the expressions are compared or what results when either expression is not a Variant.

Table A-9Comparison results when either expression is not a<br/>Variant

If	Then
Both expressions are numeric data types (Integer, Long, Single, Double, or Currency).	Perform a numeric comparison.
Both expressions are String.	Perform a string comparison.
One expression is a numeric data type and the other is a Variant of type 2-7 (a numeric data type) or type 8 (String) that can be converted to a number.	Perform a numeric comparison.

(continues)

If	Then
One expression is a numeric data type and the other is a Variant of type 8 (String) that cannot be converted to a number.	A Type Mismatch error occurs.
One expression is a String and the other is a Variant of any type except Null.	Perform a string comparison.
One expression is Empty and the other is a numeric data type.	Perform a numeric comparison, using 0 as the Empty expression.
One expression is Empty and the other is a String.	Perform a string comparison, using a zero-length string as the Empty expression.
One expression is Empty and the other is a Variant of type 8 (String).	A Type Mismatch error occurs.

Table A-9Comparison results when either expression is not a<br/>Variant (continued)

If <expr1> and <expr2> are both Variant expressions, their type determines how they are compared. Table A-10 shows how the expressions are compared or what results from the comparison, depending on the type of the Variant.

Table A-10	Comparison	results when	either exp	ression is n	ot a Variant
------------	------------	--------------	------------	--------------	--------------

If	Then
Both Variant expressions are of type 2-7 (numeric data types).	Perform a numeric comparison.
Both Variant expressions are of type 8 (String).	Perform a string comparison.
One Variant expression is of type 2-7 (a numeric data type) and the other is of type 8 (String).	The numeric expression is less than the String expression.
One Variant expression is Empty and the other is of type 2-7 (a numeric data type).	Perform a numeric comparison, using 0 as the Empty expression.
One Variant expression is Empty and the other is of type 8 (String).	Perform a string comparison, using a zero-length string as the Empty expression.
Both Variant expressions are Empty.	The expressions are equal.

If a Currency is compared with a Single or Double, the Single or Double is converted to a Currency. This causes any fractional part of the Single or Double value less than 0.000000001 to be lost and might cause two values to compare as

equal when they are not. Such a conversion can also cause an Overflow error if the magnitude of the Single or Double is too large.

**Example** The following example shows the typical use of a comparison operator to evaluate the relationship between variables A and B. The example displays an appropriate message depending on whether A < B, A = B, or A > B. The other comparison operators can be used in a similar way.

```
Sub Start()
  Dim A As Integer, B As Integer, Msg As String
  Super::Start()
  ' Get first variable
  A = Rnd * 1000
  ' Get second variable
  B = Rnd * 1000
  ' Evaluate relationship
  If A < B Then
     ' Create correct message
     Msg = " is less than "
  ElseIf A = B Then
     Msg = " is equal to "
  Else
     Msg = " is greater than "
  End If
  ' Display results
  ShowFactoryStatus ( A & Msg & B )
End Sub
```

For information about using the code examples, see "Using the code examples" in Chapter 6, "Statements and functions."

See also VarType function

# **Eqv** operator

Performs a logical equivalence on both expressions.

Syntax	<result> = <expr2> Eqv <expr2></expr2></expr2></result>			
Description	If either expression is a Null, <result> is also Null. When neither expression is a Null, result is determined according to Table A-11.</result>			
	Table A-11         Results for the Eqv operator when neither expression is a Null			
	<expr1> <expr2> <result></result></expr2></expr1>			
	True	True	True	

(continues)

Table A-11	Results for the Eqv operator v	when neither expression is a Null
True	False	False
False	True	False
False	False	True

Example The following example displays a message that depends on the value of variables A, B, and C, assuming that no variable is a Null. If A = 10, B = 8, and C = 6, both expressions evaluate True. As a result, the Eqv expression also evaluates to True.

```
Sub Start()
  ' Declare variables
  Dim A, B, C, Msg
  Super::Start( )
  ' Assign values
  A = 10: B = 8: C = 6
  ' Evaluate expressions
  If A > B Eqv B > C Then
     Msg = "Both expressions are True or both are False."
  Else
     Msg ="One expression is True and one is False."
  End If
  ' Display results
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples" in Chapter 6, "Statements and functions."

# Imp operator

Performs a logical implication on two expressions.

Syntax <result> = <expr1> lmp <expr2>

Description Table A-12 illustrates how <result> is determined.

<expr1></expr1>	<expr2></expr2>	<result></result>
True	True	True
True	False	False
True	Null	Null
False	True	True
False	False	True

Table A-12 Results for the Imp operator

	• •		
<expr1></expr1>	<expr2></expr2>	<result></result>	
False	Null	True	
Null	True	True	
Null	False	Null	
Null	Null	Null	

 Table A-12
 Results for the Imp operator

**Example** The following example displays a message that depends on the value of variables A, B, and C, assuming that no variable is a Null. If A = 10, B = 8, and C = 6, both expressions evaluate True. Because both are True, the Imp expression is also True.

```
Sub Start()
   ' Declare variables
   Dim A, B, C, Msg
   Super::Start()
   ' Assign values
   A = 10: B = 8: C = 6
   If A > B Imp B > C Then
        Msg = "The left expression implies the right expression."
   Else
        Msg = "The left expression doesn't imply the right
        expression."
   End If
   ' Display results
   ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples" in Chapter 6, "Statements and functions."

## Is operator

Compares two object reference variables.

 Syntax
 <result> = <objref1> ls <objref2>

 Description
 If <objref1> and <objref2> both refer to the same object, <result> is True; if they do not, <result> is False. Two variables can be made to refer to the same object in several ways.

 In the following example. A has been set to refer to the same object as Rieles in the same object in the same object in the same object is refer to the same object in the same object is refer to the same object in the same object is refer to the same object in the same object is refer to the same object is refer to the same object in the same object is refer to the same object is refer to the same object in the same object is refer to the same object is refer to the same object in the same object is refer to the sam

In the following example, A has been set to refer to the same object as B:

Set A = B

The following example makes A and B refer to the same object as C:

```
Set A = C
Set B = C
```

**Example** The following example uses the Is operator to evaluate whether two object reference values refer to the same control:

```
Sub Start()
  Dim MLabel1 As AcLabelControl
  Dim MLabel2 As AcLabelControl
  Dim Msg As String, LoopCount As Integer
  Super::Start( )
  'Instantiate two new, separate objects
  Set MLabel1 = New AcLabelControl
  Set MLabel2 = New AcLabelControl
  'Loop twice through the Is evaluation
  For LoopCount = 1 to 2
     If MLabel1 Is MLabel2 Then
       Msq = "MLabel1, MLabel2 DO refer to the same object."
       ShowFactoryStatus ( Msg )
     Else
       Msg = "MLabel1, MLabel2 DON'T refer to the same object."
        ShowFactoryStatus ( Msg )
        'Before the next Is evaluation loop, force identity.
        Set MLabel1 = MLabel2
     End If
  Next LoopCount
```

#### End Sub

For information about using the code examples, see "Using the code examples" in Chapter 6, "Statements and functions."

See also Set statement

## Like operator

Compares two string expressions.

Syntax <result> = <exprs> Like <pattern>

**Description** If <exprs> matches <pattern>, <result> is True; if there is no match, <result> is False; and if either <exprs> or <pattern> is a Null, <result> is also a Null. The case sensitivity and character sort order of the Like operator depend on the setting of the Option Compare statement.

Unless other wise specified, the default string-comparison method for each module is Option Compare Binary; that is, string comparisons are case-sensitive.

Built-in pattern matching provides a versatile tool for string comparisons. The pattern-matching feature allows you to use wildcard characters, such as those recognized by the operating system, to match strings. The wildcard characters and what they match are shown in Table A-13.

Character(s) in <pattern></pattern>	Matches in <exprs></exprs>
?	Any single character
*	Zero or more characters
#	Any single digit (0-9)
[character list]	Any single character in <character list=""></character>
[^character list]	Any single character not in <character list=""></character>

Table A-13 Wildcard characters

A group of one or more characters <character list> enclosed in brackets ([]) can be used to match any single character in <exprs> and can include almost any characters in the ANSI character set, including digits. The special characters left bracket ([), question mark (?), number sign (#), and asterisk (\*) can be used to match themselves directly only by enclosing them in brackets. The right bracket (]) cannot be used within a group to match itself, but it can be used outside a group as an individual character.

In addition to a simple list of characters enclosed in brackets, <character list> can specify a range of characters by using a hyphen (-) to separate the upper and lower bounds of the range. For example, [A-Z] in <pattern> results in a match if the corresponding character position in <exprs> contains any of the uppercase letters in the range A through Z. Multiple ranges are included within the brackets without any delimiting. For example, [a-zA-Z0-9] matches any alphanumeric character.

Other important rules for pattern matching include the following:

- The hyphen (-) can appear either at the beginning (after an exclamation mark if one is used) or at the end of <character list> to match itself. In any other location, the hyphen is used to identify a range of ANSI characters.
- When a range of characters is specified, they must appear in ascending order (low to high).
- The character sequence [] is ignored; it is evaluated as a zero-length string.
- The special character exclamation point (!) is ignored.

# **Example** The following example shows the result of mixing and matching various wildcard characters in conjunction with the Like operator:

```
Sub Start()
  Super::Start( )
  'Returns True (1)
  ShowFactoryStatus("PB123" Like "P[A-F]###")
  'Returns False (0)
  ShowFactoryStatus("PG123" Like "P[A-F]###")
  'Returns False
  ShowFactoryStatus("PG123" Like "P[^A-F]###")
  'Returns False
  ShowFactoryStatus("PB12x" Like "P[A-F]###")
  'Returns True
  ShowFactoryStatus("PB123" Like "?B???")
  'Returns False
  ShowFactoryStatus("PB123" Like "?B??")
  'Returns True
  ShowFactoryStatus("PB123" Like "P*")
  'Returns False
  ShowFactoryStatus("PB123" Like "*P")
  'Returns True
  ShowFactoryStatus("PB123" Like "*P*")
  'Returns True
  ShowFactoryStatus("F" Like "[^A-Z] *")
  'Returns True
  ShowFactoryStatus("F" Like "[A-Z]*")
  'Returns True
  ShowFactoryStatus("abracadabra122%$a" Like "a*a")
  'Returns False
  ShowFactoryStatus("CAT123khq" Like "B?T*")
End Sub
```

See also Format, Format\$ functions StrComp function Option Compare statement

# Mod operator

Divides two numbers and returns only the remainder.

Syntax <result> = <operand1> Mod <operand2>

**Description** The modulus, or remainder, operator divides <operand1> by <operand2> (rounding floating point numbers to integers) and returns only the remainder as the <result>. For example, in the expression A = 19 Mod 6.7, A (which is <result>) equals 5. The operands can be any numeric expression.

Usually the data type of <result> is an Integer. However, <result> is a Null (Variant type 1) if one or both operands are Null expressions. Any operand that is Empty (Variant type 0) is treated as 0.

**Example** The following example uses the Mod operator to determine if a 4-digit year is a leap year:

```
Sub Start()
  Dim TestYr As Integer, LeapStatus As String
  Super::Start( )
  TestYr = Rnd * 120 + 1899
  ' Divisible by 4?
  If TestYr Mod 4 = 0 And TestYr Mod 100 = 0 Then
     ' Divisible by 400?
     If TestYr Mod 400 = 0 Then
       LeapStatus = " is a leap year."
     Else
        LeapStatus = " is centesimal, but not a leap year."
     End If
  ElseIf TestYr Mod 4 = 0 Then
     LeapStatus = " is a leap year."
  Else
     LeapStatus = " is not a leap year."
  End If
  ' Display results
  ShowFactoryStatus ( TestYr & LeapStatus )
End Sub
```

For information about using the code examples, see "Using the code examples" in Chapter 6, "Statements and functions."

**See also** VarType function

## Not operator

Performs logical negation on an expression.

Syntax <result> = Not <expr>

**Description** Table A-14 illustrates how <result> is determined.

Table A-14         Results for the Not ope
--

<expr></expr>	<result></result>
True	False

(continues)

----

Table A-14	Results for the Not operator (continued)	
False	True	
Null	Null	

In addition, the Not operator inverts the bit values of any variable and sets the corresponding bit in <result> according to the truth table, Table A-15.

Table A-15 Bitwise results for the Not operator

Bit in <expr></expr>	Bit in <result></result>
0	1
1	0

If an integer variable has the value 0 (False), the variable becomes 1 (True); if it has the value of 1 it becomes 0.

Example The following example displays a message that depends on the value of variables A, B, and C, assuming that no variable is a Null. If A = 10, B = 8, and C = 6, the Not expression evaluates True because A is not equal to B.

```
Sub Start()
  ' Declare variables
  Dim A, B, Msg
  Super::Start( )
  ' Assign values
  A = 10: B = 8
  ' Evaluate expression
  If Not A = B Then
     Msg = "A and B aren't equal."
  Else
     Msg = "A and B are equal."
  End If
  ' Display results
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples" in Chapter 6, "Statements and functions."

## Or operator

Performs a logical disjunction on two expressions.

Syntax <result> = <expr1> Or <expr2>
**Description** If either or both expressions evaluate True, <result> is True. Table A-16 illustrates how <result> is determined.

<expr1></expr1>	<expr2></expr2>	<result></result>
True	True	True
True	False	True
True	Null	True
False	True	True
False	False	False
False	Null	Null
Null	True	True
Null	False	Null
Null	Null	Null

**Table A-16**Results for expressions that use the Or operator when either or both<br/>expressions evaluate True

The Or operator performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in <result> according to the truth table, Table A-17.

Bit in <expr1></expr1>	Bit in <expr2></expr2>	<result></result>	
0	0	0	
0	1	1	
1	0	1	
1	1	1	

**Table A-17**Results for expressions that use the Or operator to compare two<br/>identically positioned bits in two numeric expressions

**Example** The following example displays a message that depends on the value of variables A, B, and C, assuming that no variable is a Null. If A = 10, B = 8, and C = 6, the left expression is True and the right expression is False. Because at least one comparison expression is True, the Or expression evaluates to True.

```
Sub Start( )
    'Declare variables
    Dim A, B, C, Msg
    Super::Start( )
    ' Assign values
    A = 10: B = 8: C = 11
    ' Evaluate expression
    If A > B Or B > C Then
        Msg = "One or both comparison expressions are True."
    Else
        Msg = "Both comparison expressions are False."
    End If
    ' Display results
    ShowFactoryStatus( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples" in Chapter 6, "Statements and functions."

#### **XOr operator**

Performs a logical exclusion on two expressions.

Syntax <result> = <expr1> Xor <expr2>

**Description** If only one of the expressions evaluates True, <result> is True. If either expression is a Null <result> is also a Null. When neither expression is a Null, <result> is determined according to Table A-18.

 Table A-18
 Results for the XOr operator when neither expression is a Null

<expr1></expr1>	<expr2></expr2>	<result></result>
True	True	False
True	False	True
False	True	True
False	False	False

**Example** The following example displays a message that depends on the value of variables A, B, and C, assuming that no variable is a Null. If A = 6, B = 8, and C = 10, both expressions evaluate False. Because both are False, the XOr expression is also False.

```
Sub Start()
  ' Declare variables
  Dim A, B, C, Msg
  Super::Start( )
  ' Assign values
  A = 6: B = 8: C = 10
  ' Evaluate expression
  If A > B Xor B = C Then
     Msg = "Only one comparison expression is True, not Both."
  Else
     Msg = "Both comparison expressions are True or both are
       False."
  End If
  ' Display results
  ShowFactoryStatus ( Msg )
End Sub
```

For information about using the code examples, see "Using the code examples" in Chapter 6, "Statements and functions."

XOr operator

#### Appendix

# B

## **Keywords**

Table B-1 lists words Actuate Basic uses as keywords. These words include data types, operators, statements, and other words that the Actuate compiler recognizes. Although Actuate Basic allows using names that are the same as function names, avoid using these names.

-	
Keyword	Reference
Access	Open statement
Alias	Declare statement
And	And operator
Any	Any data type
AnyClass	SetBinding function
AppActivate	Reserved
Append	Open statement
As	Class statement, Declare statement, Dim statement, Open statement, TypeAs statement
Assert	Assert statement
BAnd	BAnd operator
Base	Option Base statement
Веер	Beep statement
Binary	Option Compare statement

#### Table B-1 Keywords and the corresponding references

Keyword	Reference
BNot	BNot operator
BOr	BOr operator
ByVal	Declare statement, FunctionEnd Function statement, SubEnd Sub statement
Call	Call statement
Case	Select Case statement
ChDir	ChDir statement
ChDrive	ChDrive statement
Class	Class statement
Clipboard	Reserved
Close	Close statement
Compare	Option Compare statement
Const	Const statement
CPointer	CPointer data type
Currency	Currency data type
Date	Date data type
Debug	Reserved
Declare	Declare statement
DefCur	Reserved
DefDbl	Reserved
DefInt	Reserved
DefLng	Reserved
DefSng	Reserved
DefStr	Reserved
Dim	Dim statement
Do	DoLoop statement
Double	Double data type
Else	IfThenElse statement, Select Case statement
ElseIf	IfThenElse statement
End	End statement
Eqv	Eqv operator
Erase	Erase statement

**Table B-1**Keywords and the corresponding references (continued)

Keyword	Reference
Err	Err statement
Error	Error statement
Exit	Exit statement
Explicit	Reserved
FileCopy	FileCopy statement
For	ForNext statement
FreeLocks	Reserved
Function	FunctionEnd Function statement
Get	Get statement
Global	Global statement
GoSub	Reserved
GoTo	GoTo statement
Group	Reserved
If	IfThenElse statement
Imp	Imp operator
Input	Input statement, Line Input statement
Integer	Integer data type
Internal	Reserved
Is	Is operator, Select Case statement
Kill	Kill statement
Let	Let statement
Lib	Declare statement
Like	Like operator
Line	Line Input statement
Load	Reserved
Local	Reserved
Lock	LockUnlock statement, Open statement
Long	Long data type
Loop	DoLoop statement
LSet	LSet statement

 Table B-1
 Keywords and the corresponding references (continued)

Keyword	Reference
Mid	Mid, Mid\$ statements
MidB	MidB, MidB\$ statements
MkDir	MkDir statement
Mod	Mod operator
MsgBox	MsgBox statement
Name	Name statement
New	Set statement
Next	ForNext statement, Resume statement
Not	Not operator
Nothing	Set statement
Null	Working with Variant data
Object	Dim statement
Of	Class statement
Off	Option Strict statement
On	On Error statement, Option Strict statement
Open	Open statement
Option	Option Base statement, Option Compare statement, Option Strict statement
Or	Or operator
Output	Open statement
Persistent	Dim statement
Preserve	ReDim statement
Print	Print statement
Put	Put statement
Random	Open statement
Randomize	Randomize statement
Read	Open statement
ReDim	ReDim statement
Rem	Rem statement
Reset	Reset statement
Resume	Resume statement
Return	Reserved

**Table B-1**Keywords and the corresponding references (continued)

Keyword	Reference
RmDir	RmDir statement
RSet	RSet statement
SavePicture	Reserved
Seek	Seek statement
Select	Select Case statement
Set	Set statement
SetAttr	SetAttr statement
Shared	Open statement
Single	Single data type
Static	Class statement, FunctionEnd Function statement, Static statement, SubEnd Sub statement
Step	ForNext statement
Stop	Stop statement
Strict	Option Strict statement
String	String data type
Sub	SubEnd Sub statement
Subclass	Class statement
Text	Option Compare statement
Then	IfThenElse statement
Time	Reserved
То	ForNext statement, LockUnlock statement, Select Case statement
Transient	Dim statement
Туре	TypeEnd Type statement, TypeAs statement
Unload	Reserved
Unlock	LockUnlock statement
Until	DoLoop statement
Variant	Variant data type
Verify	Reserved
Volatile	Dim statement
Wend	WhileWend statement

**Table B-1**Keywords and the corresponding references (continued)

Keyword	Reference
While	WhileWend statement
Width	Width statement
Write	Write statement
XOr	XOr operator

**Table B-1**Keywords and the corresponding references (continued)

## **Trigonometric identities**

This appendix contains information about trigonometric formulas that Actuate Basic supports. Actuate Basic supports only the Sin, Cos, Tan, Asin, Acos, and Atn built-in trigonometric functions. To use other trigonometric functions, you need to derive and write the functions yourself. To do this, use Table C-1, which shows the most commonly used trigonometric identities.

Name of formula	Formula
Secant	Sec(X) = 1 / Cos(X)
Cosecant	$\operatorname{Cosec}(X) = 1 / \operatorname{Sin}(X)$
Cotangent	Cotan(X) = 1 / Tan(X)
Inverse Cotangent	$\operatorname{Arccotan}(X) = -\operatorname{Atn}(X) + 2 * \operatorname{Atn}(1)$
Hyperbolic Sine	HSin(X) = (Exp(X) - Exp(-X)) / 2
Hyperbolic Cosine	HCos(X) = (Exp(X) + Exp(-X)) / 2
Hyperbolic Tangent	HTan(X) = (Exp(X) - Exp(-X)) / (Exp(X) + Exp(-X))
Hyperbolic Secant	HSec(X) = 2 / (Exp(X) + Exp(-X))
Hyperbolic Cosecant	HCosec(X) = 2 / (Exp(X) - Exp(-X))
Hyperbolic Cotangent	HCotan(X) = (Exp(X) + Exp(-X)) / (Exp(X) - Exp(-X))
Inverse Hyperbolic Sine	HArcsin(X) = Log(X + Sqr(X * X + 1))
Inverse Hyperbolic Cosine	HArccos(X) = Log(X + Sqr(X * X - 1))

#### Table C-1 Supported trigonometric formulas

Name of formula	Formula
Inverse Hyperbolic Tangent	HArctan(X) = Log((1 + X) / (1 - X)) / 2
Inverse Hyperbolic Secant	HArcsec(X) = Log((Sqr(-X * X + 1) + 1) / X)
Inverse Hyperbolic Cosecant	HArccosec(X) = Log((Sgn(X) * Sqr(X * X + 1) + 1) / X)
Inverse Hyperbolic Cotangent	HArccotan(X) = Log((X + 1) / (X - 1)) / 2
Logarithm to base N	LogN(X) = Log(X) / Log(N)

 Table C-1
 Supported trigonometric formulas (continued)

# Index

### Symbols

` (back quotation mark) character 30 ^ operator 7,475 , (comma) character as format symbol 188, 192 as printer code 356 ; (semicolon) character numeric formats and 189 printer codes and 356, 438 : format symbol 194 ! format symbol 196 ! icon 319, 322 ! type declaration symbol 21 ? (question mark) character as format symbol 345 as wildcard 487 ? icon 319, 322 . (dot) format symbol 188 . (dot) operator 34 ' (apostrophe) character 11, 371 (single quotation mark) character 11, 371 " (double quotation mark) character date formats and 191 literal strings and 29 numeric formats and 187 print messages and 96 () parentheses characters in expressions 469 in procedure declarations 41 numeric formats and 189 {} (braces) characters 30 @ (at-sign) character as format symbol 195 as type declaration symbol 21, 31 \* operator 8, 470 \* wildcard character 487 / operator 8, 474 \ (backslash) characters 187, 191  $\land$  operator 8,475 & (ampersand) character 111 & format symbol 195 & operator 10, 476

& type declaration symbol 21 &H prefix characters 249 &O prefix characters 333 # (number sign) character as numeric format symbol 187 as type declaration symbol 21 as wildcard 487 in date and time values 31, 32 % format symbol 188 % type declaration symbol 21 + (plus sign) character as line continuation symbol 11 commenting code and 11 numeric formats and 189 + operator 8, 471 < format symbol 196 < operator 8, 481 <= operator 8, 481 <> operator 9, 481 = operator assignment 5,9 comparisons 8, 481 > format symbol 196 > operator 8, 481 >= operator 8, 481 - (minus sign) character in strings 27 - (hyphen) character as wildcard 487 numeric formats and 189 - operator 8, 473 \$ format symbol 189 \$ type declaration symbol 21

#### Numerics

0 format symbol 187 0 keyword 373

## A

A/P format symbols 195 a/p format symbols 195 Abort buttons 319, 320, 322 abridged dates 316 Abs function 74 absolute paths 181 absolute values 74 AC\_CENTURY\_BREAK property 88, 466 AC\_LOCALE\_CURRENCY\_RADIX value 226 AC\_LOCALE\_ MONTHS\_LONG value 226 AC\_LOCALE\_ MONTHS\_SHORT value 226 AC LOCALE WEEKDAYS LONG value 226 AC\_LOCALE\_WEEKDAYS\_SHORT value 226 AC\_LOCALE\_CURRENCY value 226 AC\_LOCALE\_CURRENCY\_FORMAT value 226 AC\_LOCALE\_CURRENCY\_THOUSAND\_ SEPARATOR value 226 AC LOCALE DATE LONG value 226 AC\_LOCALE\_DATE\_SEPARATOR value 226 AC LOCALE DATE SHORT value 226 AC\_LOCALE\_NUM\_RADIX value 226 AC\_LOCALE\_NUM\_THOUSAND\_ SEPARATOR value 226 AC\_LOCALE\_TIME\_AM\_STRING value 226 AC LOCALE TIME FORMAT value 226 AC\_LOCALE\_TIME\_PM\_STRING value 226 AC\_LOCALE\_TIME\_SEPARATOR value 226 accelerated depreciation 437 accented characters 25 access permissions 76, 337, 338 access restrictions 293, 338 accessing code examples 12,72 external functions 4, 46, 48 files 336 Java classes 53, 114, 115 procedures 5, 38 subprocedures 39 variables 14,400 AcCleanup function 46 accounts receivable 128 AcCurrency structure 50, 51 AcFont type 220 Acos function 75 AcTextPlacement type 220

Actuate Basic accessing files and 336 accessing Java objects and 53 binary data and 25 building arrays and 17 coding conventions for 10–12 coding examples for 12, 72 compatibility with Visual Basic 4, 18 declaring arrays and 17 declaring variables and 14 developing with 4,5 encoding scheme for 25 extending functionality of 38, 46 function and statement reference for 71 generating code for 5 naming rules for 11 programming tasks categorized 61 recursive statements and 6 supported data types for 18 unassigned data types and 20 verifying C function arguments 51 Actuate Basic reports. See reports Actuate Foundation Class Library 211 Actuate Foundation Classes 5, 20 Actuate Viewer. See DHTML Viewer AddBurstReportPrivileges function 76 adding arguments to functions 40, 201 arguments to procedures 41–42, 84 arguments to subprocedures 39, 425 C functions 46, 47, 48, 49 class variables 14, 16, 100 comments to code 11, 371 formatting patterns and strings 184, 187, 195 Java classes 53, 114, 115 line breaks to statements 11 search indexes 76, 78 source code to designs 5, 40, 72 type-declaration characters 7, 21, 144, 149 addition operator (+) 8, 471 AddValueIndex function 76 AFC. See Actuate Foundation Classes Age\_ArchiveBeforeDelete constant 403 Age DeleteDependencies constant 404 Age\_NoOptions constant 403 aging policy 403

aging properties 404 aging rules 403 *See also* archives alarms 83 alerts 83 Alias keyword 47, 49, 144, 145 aliases assigning to data types 20, 33, 451 assigning to variables 400 calling C functions and 47, 49 defining for external procedures 145, 146 alignment 300, 384 allocating memory 148, 243 alphanumeric characters 487 alternate names. See aliases AM/PM format symbols 195 am/pm format symbols 195 ampersand (&) character as concatenation operator 10, 476 as format symbol 195 as type declaration symbol 21 XML data converter and 111 AMPM format symbols 195 ANALYSIS format 231, 278 And operator 9,477 angles calculating hyperbolic sin for 171 getting arc cosine 75 getting arc sine 81 getting arctangents 82 getting cosine 113 getting sine 409 getting tangent 440 annuities defined 267 getting future value 205 getting interest payments 266 getting interest rates 366 getting number of periods 328 getting present value 361 getting principal 351 getting total payments 349 programming functions listed 67 ANSI character codes 79 ANSI characters getting 95, 97 matching 487

programming functions for 69 storing binary data and 25 Any data type 18 any pointer type 52 AnyClass type 51 apostrophe (') character 11 apostrophe (′) character 371 Append mode 173, 338 applications accessing C functions and 46 converting date values and 346 copying text from 30 defining variables global to 15 developing 5 getting currently running 212, 229 getting run-time errors for 162, 163, 164 running cross-platform 4 running from command prompt 107, 408 arc cosine 75 arc sine 81 archives 215, 399, 403 arctangents 82 area 109 argument lists See also arguments function calls and 201 procedure calls and 41, 42, 84 subprocedure calls and 425 arguments See also command-line arguments; parameters adding to functions 40, 201 adding to procedures 41-42, 84 adding to subprocedures 39, 425 calling C functions and 48, 49, 51 checking for invalid 51, 78 converting for DLLs 85 disabling type checking for 18 overloading procedures and 201, 424 passing by reference 41, 48, 84 passing by value 42, 48, 84, 145 removing type restrictions for 49 returning null values 7 arithmetic functions and statements. See math functions and statements arithmetic operations. See mathematical operations

arithmetic operators See also specific described 7 order of evaluation for 469, 470 variants and 23, 24 array elements (defined) 16 array functions and statements 18, 62, 63 array names 161, 368 array variables Dim statements 148 Function...End Function statements 203 Global statements 245 ReDim statements 368 Sub...End Sub statements 425 Type...End Type statements 449 arrays adding multiple dimensions 17 associating with classes 100 calculating values and 291 changing at runtime 281 converting Java 55 converting strings to 290 creating user-defined variables and 449 de-allocating memory for 160 debugging Java 57 declaring 16-18 defining dimensions 148, 244 defining dynamic 149, 244 defining fixed-size 414 defining global 243 defining static 450 determining contents 199 getting lower bounds 280, 281 getting number of elements in 281 getting upper bounds 452 initializing 18, 368 overriding 341 passing to C functions 49, 51 passing to procedures 84, 203, 414 passing to subprocedures 425 preserving data in 17 reinitializing fixed 160 resizing 17, 368, 370 returning literal values 450 setting contents of 199 setting lower bounds 340 sorting 460

storing binary data and 26 As Any keyword 49 As keyword in Class statements 101 in Declare statements 144, 146 in Dim statements 21, 149 in Function statements 40, 203 in Global statements 244 in Name statements 324 in Open statements 337 in ReDim statements 369 in Static statements 415 in Sub statements 426 in type declarations 33 Asc function 79 ASCII characters getting 96, 289 naming rules and 11 programming functions for 69 ASCII text files. See text files AscW function 80 Asin function 81 Assert statement 81 assets See also annuities getting depreciation of 142, 411, 436 getting net present values 330 getting rate of return 268, 311 programming tasks for 67 assigning data types to constants 108 to functions 146, 203, 204 to procedures 41, 43, 426 to variables 6, 20-21, 24, 146 assigning values to data types 18, 30, 287 to Java objects 53 to variables 5, 15, 34, 249, 287, 414 to variants 23, 30, 288 assigning variables to classes 14, 16, 100 to object references 397 to variant data 121, 455 assignment operator (=) 5, 9 assignment statements 5,55 asterisk (\*) character as multiplier 8, 470

as wildcard 487 Atn function 82 at-sign (@) character as format symbol 195 as type declaration symbol 21, 31 ATTR\_ARCHIVE constant 215, 399 ATTR\_DIRECTORY constant 215 ATTR\_HIDDEN constant 214, 399 ATTR\_NORMAL constant 214, 399 ATTR\_READONLY constant 214, 399 ATTR\_SYSTEM constant 215, 399 ATTR VOLUME constant 215 attributes *See also* properties defining object 34 getting current 213 getting file type 172 getting locale-specific 226 getting system 214 setting file type 398

#### В

back quotation mark (`) character 30 backslashes ( $\)$  187, 191 backspace characters 96 BAnd operator 8, 215, 478 .bas files. See source files Basic. See Actuate Basic Basic reports. See reports Beep statement 83 beeps 83 binary data assigning to strings 26 comparing 28 decoding 110 encoding 110 storing 25 binary files accessing 336 converting strings to 110 converting to strings 110 getting bytes from 261 getting current position 292 getting current position in 393 locking 294 opening 338, 359

reading from 208, 210, 259 setting byte position for 391 testing input position in 159 writing to 110, 359 binary images 63 Binary keyword 342 Binary mode 173, 210, 338, 359 binary string functions 26 binding column names to data rows 400 bitwise And operator 8, 215, 478 bitwise comparisons 215, 478, 480, 491 bitwise Not operator 479 bitwise Or operator 8, 480 blank lines 256 blank space characters. *See* space characters BNot operator 479 Boolean data type 54, 55 boolean type 54 Boolean values assigning numeric values for 19, 43 comparing 9, 396 evaluating 43, 177 getting 187 Boolean variables 177 BOr operator 8,480 braces ({ }) characters 30 brackets ([]) characters 487 branching enum types and 156 program comments and 371 restrictions for 460 run-time errors and 334 to specified line 246 branching statements 62 browser code 230 browser scripting controls 230 browsers. See web browsers buffers allocating 336 flushing 105 reading 372 setting size 338 writing to 337 built-in functions 40 built-in methods 5 burst reports 76 bursting 402

button style constants 319, 322 buttons adding to dialog boxes 318, 321 defining as default 319, 322 selecting at run time 320 byte copy operations 285, 305, 380 byte functions 69 byte locks 294 byte type 54 ByVal keyword arrays and 52 C functions and 48, 51 in Declare statements 145 in Function statements 202 in Sub statements 425 procedures and 42, 84

#### С

C function names 48, 49 C functions accessing 46 adding 46 aliasing 49 allocating data in 18 assigning data types to 48, 49, 51, 52 calling 46, 48, 49, 51 declaring 46, 47–51 passing arguments to 48, 49, 51 passing arrays to 51 passing null pointers to 52 passing object reference variables to 52 passing strings to 49, 51 removing type restrictions for 49 renaming 49 returning values 52, 55 unloading libraries for 46 calculations arithmetic operators for 7 arrays and 291 currency values and 31 date and time values in 32 functions for 68 local variables and 16 procedures performing 5, 6, 38, 41 string variables and 25 calculator 407

calendar 33 Call keyword 84 Call statement 42, 84 Cancel buttons 319, 320, 322 capitalization 28, 196, 282, 453 carriage return characters 96, 260, 261 cascading style sheets 242, 434 case conversions 28, 196, 282, 453 Case Else keyword 396 Case keyword 395 case statements 62, 394 case-insensitive comparisons 28, 396, 418 case-sensitive comparisons 9, 28, 342, 396, 418 cash flow. See annuities; assets CCur function 35,85 CDate function 33, 35, 87 CDbl function 35,90 CF\_LINK constant 219, 401 CF\_TEXT constant 219, 401 changing capitalization 28, 196, 282, 453 current drive and directory 92, 93, 94 file names 323 headlines 402 strings 304, 307, 422 values 41, 42 char type C functions and 50, 52 Java objects and 54 character arrays 25 character codes entering in strings 25 getting 26, 79, 80 translating 95, 97 character conversion functions and statements 63 character lists 487 character placeholders 195, 196 character sets 25 character strings 25, 26 See also strings character strings. See strings characters See also space characters; special characters adding type-declaration 21, 144 changing case 28, 196, 282, 453

comparing 28, 342, 418, 481, 486 converting to XML 111 converting UCS-2 codes to 97 copying files and 174 deleting files and 279 encoding 25,80 entering in strings 25, 29 getting ANSI 95 getting escape 111, 432 getting number of 26 getting specified 26 getting width of 223 naming conventions and 11, 49 null-terminated strings and 51 repeating 420, 421 chart drawing functions getting escaped characters for 433 setting CSS styles for 434 setting font values for 431 setting numeric values for 429 setting SVG attributes for 427, 428 ChDir statement 92 ChDrive statement 93 Chr function 95 Chr<sup>\$</sup> function 95 ChrW function 97 ChrW\$ function 97 CInt function compared with Fix and Int 183 described 35,98 circumference 109 class arrays 100 class definitions 100, 154 class functions and statements 62 class IDs 216, 217, 218 class names assigning to classes 100 determining fully qualified 77 getting 217, 218 class scope 14, 38 Class statement 100, 154 class variables accessing 62 adding 14, 16, 100 declaring 149, 244 initializing 245

classes See also Actuate Foundation Classes accessing Java 53, 114, 115 aliasing 400 calling methods for 38, 101 creating objects for 34 creating persistent 326 declaring 100, 101, 154 defining procedures in 38 defining user-defined types as 34 developing with 5,62 getting instances of 238 instantiating 62, 115, 325, 326, 397 nesting 101 testing for objects in 273 testing for specific 218 CLASSPATH variable 53, 157 cleanup functions 46 ClearClipboard function 102 client applications 212 Clipboard clearing 102 copying to 401 getting contents 219 Clipboard formats 401 CLng function 35, 103 clock See also time formatting 32, 191, 194 getting current time 441 getting specific time for 443 setting 32, 88, 133 specifying start of day for 33 Close statement 105 closing disk files 105, 372 code See also Actuate Basic accessing Java objects and 54 adding comments to 11, 371 adding external functions to 46, 49 adding Java classes 53, 114, 115 controlling statement flow in 44 disabling while debugging 371 generating 5 implicit declarations and 14 naming conventions for 11 pasting text blocks to 30

code (continued) referencing specific objects in 39 simplifying programming tasks in 38 unassigned data types and 20 writing 5, 10, 11 code editors 46 code example window 72 code examples 12,72 code points 80 colon (:) character in format patterns 194 color constants 364 color values getting 376 setting 20, 364, 378, 428 translating 363 column names 400 column names. See column headings columns 400, 438 comma (,) character as format symbol 188, 192 as printer code 356 comma separated values 231, 277 Command function 107 command prompt 407 Command<sup>\$</sup> function 107 command-line arguments 107 comments adding to code 11, 371 displaying 64 comparison functions 69 comparison operators described 8, 481 Is keyword and 396 order of evaluation for 469, 470 comparisons among a range of values 395 bitwise 478, 480, 491 currency 482 expressions 8, 394, 481 logical values 9,396 null values and 28 numeric values 264 object reference variables 9, 485 string values 28, 342, 418, 486 variants 482 compiling 73, 107, 235 completion requests 224, 402

components adding to designs 5 getting class information for 77, 217, 218 computed values. See expressions concatenation 470, 471 concatenation operator 10, 476 conditional expressions 44 conditions creating loops and 43 evaluating 199, 204 returning specific values 255 setting for control structures 150, 253, 459 testing 65, 253 trapping errors and 82 conjunction operator 9,477 Const statement 23, 108 constants adding to arrays 450 assigning data types to 7, 20, 108 associating with database columns 400 declaring global 146 declaring symbolic 108, 401 defining enum types and 22 embedding special characters in 30 naming 11, 108 parsing string 283, 285, 379, 380 programming statements for 70 setting values for 23 container objects 220 context names 212, 229 context-sensitive strings 166 continuation character (+) 11 control codes 96 control structures *See also* structures branching 246, 460 creating 43 executing indefinitely 150 executing with conditions 253, 459 exiting 44, 168 nesting 44, 152, 253 repeating statement blocks 197, 459 stepping through 416 controlling program execution 416 controlling program flow 43, 62, 84 controlling report generation 5, 43

controls adding to frames 101 getting size of 224 scaling 230 testing references to 486 conversion functions and statements 35, 63, 68,69 conversions binary images 110 C function types 49 case 28, 196, 282, 453 dates to strings 344 dates to variants 122 decimal to hexadecimal 248 decimal to octal 332 expressions to dates 87 expressions to strings 118 Java arrays 55 Java types 54–56 locale-specific values 35 numbers to currency 85 numbers to dates 271 numbers to doubles 90 numbers to integers 98 numbers to longs 103 numbers to singles 116 numbers to strings 27, 31, 184, 248, 332, 417 numbers to variants 121 numeric types to dates 32 radians to degrees 82 strings to arrays 290 strings to dates 344 strings to numbers 116, 183, 264, 348 testing status 270, 275 UCS-2 codes to characters 97 values to specific types 34 variant data types 23 XML characters 111 ConvertBFileToString function 110 ConvertStringToBFile function 110 ConvertToXML function 111 Copy command (DOS) 174 copying binary data 26 bytes 285, 305, 380 class variables 62

code examples 72,73 data 103, 219, 401 files 174 specific characters 283, 302, 379 strings 29, 301, 401, 447 variables 112 CopyInstance statement 112 copyright symbols 96 corrupted data 210 Cos function 113 cosecant 501 cosine 113 cotangent 501 counter variables 198 counters incrementing 151, 197, 460 setting 197, 198 CPointer data type assigning to variables 288 assigning values 18, 287 initialization values for 245 mapping to C functions 50 returning from C functions 52 CreateJavaClassHandle function 114 CreateJavaObject function 53, 115 creating control structures 43 data structures 33 expressions 6 indexed searches 76, 78 instance variables 100 Java classes 114, 115 Java objects 53–54, 115 persistent files 326 procedures 5, 38, 40 report archives 403 report files 402 reports 5 source files 72 statements 5, 11, 61 static variables 100 temporary files 106, 159, 173 user-defined types 20, 33-34, 450 criteria. See conditions; parameters criteria. See search conditions cross-platform applications 4 CSng function 35, 116

CSS formats 242, 434 CStr function 35, 118 CSV formats 231, 277 cumulative totals 16 CurDir function 119 CurDir\$ function 119 Currency data type assigning 18, 31 C functions and 50 converting to 85 currency symbols 25, 189, 349 currency type-declaration symbol 21, 31 currency values See also Currency data type comparing 482 defining as constant 23 defining as variants 24 maintaining precision for 31 CurrencyDecimal parameter 35 CurrencyGrouping parameter 35 current date 32 current drive/directory 120 custom browser code 230 customizing data types 33–34 reports 238 CVar function 35, 121 CVDate function 35, 122

#### D

d format character 345 data See also values assigning to variables 256, 450 copying 103, 219, 401 corrupting 210 displaying 356, 439 exporting/importing 346, 464 formatting 184, 463 misreading 210 printing unformatted 354 reading from disk 208, 256, 259, 289 returning from user input 454 searching for 77 storing 6, 14, 18 structuring 33

writing to disk files 354, 357, 372, 463 data filters 112 data rows 17, 400 See also records data streams 112, 347 data structures 450 See also structures data types *See also* specific type aliasing 451 assigning to variables 6, 20–21, 24, 146 assigning values to 18, 287 building arrays and 16, 17 building Java objects and 54, 54–56 calling C functions and 48, 49, 51, 52 converting values to specific 34 creating 20, 33–34, 450 declaring in procedures 41, 43, 426 defining constant values and 7, 23, 108 defining enums and 156 defining variant data and 23-24 determining 236 mismatched 286 overview 18-20 passing multiple 49 passing to procedures and DLLs 86 reading sequential files and 256 restrictions for 23 returning from expressions 6 returning from function calls 40, 146, 203, 204returning null values and 52, 55 specifying 21 suppressing type checking for 18 data typing conventions 4 databases automating frequent operations for 38 exporting data to 464 mapping data rows to columns in 400 Date data type 19, 31, 50, 52 See also dates date expressions creating 127, 133, 457 entering numbers in 32, 88 entering time values in 88, 129, 132 estimating year from 88 evaluating 270

getting number of weeks from 131 getting time values from 250, 308, 387 locale-specific format characters in 345 mathematical operations in 7, 32 Date function 32, 125 date functions 346 date functions and statements 32, 63 date separators 192 date serial numbers assigning to variables 31, 327 converting dates to 32, 63, 135 converting to dates 88 tormatting 190 returning 124 time values in 250, 308, 388, 443, 445 date stamps file creation 175, 179 file modification 175, 179 date variables 32 Date\$ function 125 DateAdd function 126 DateDiff function 128 DatePart function 132 dates adding specified time intervals to 126 assigning to variables 125 assigning values 19, 31 calculating values 7, 32, 126, 128 checking conversion status 270 converting expressions to 87, 271, 344 converting to strings 344 converting to variants 122 converting values to 32, 33 defining variants as 24, 27, 347 displaying 131, 137 entering literal 31, 32 extracting days from 32 extracting month from 32, 315 extracting year from 32, 465 formatting 32–33, 125, 184, 190 getting current 32 getting day of month 140 getting difference between specified 128 getting for specified string 138 getting number of weeks 130 getting system 125, 327 getting weekday 32, 154, 226, 457

importing 346 parsing 132, 344, 347 redefining formats for 191 returning from serial numbers 136 specifying for report archiving 404 specifying locale specific 89, 125, 190, 344 valid ranges for 270 DateSerial function 31, 32, 135 DateValue function 138 day format characters 192 Day function 32, 140 DDB function 142 de-allocating memory 46, 160 debugger 57 debugging branching and 198, 247 creating Java objects and 57 creating timers for 354 disabling source code while 371 initializing global variables and 245 invalid arguments and 78 nesting user-defined types and 34 repeating random numbers and 365 resuming program execution and 373 simulating errors and 167, 374 suspending program execution and 416 decimal numbers converting to hexadecimal 248 converting to octal 332 defining colors with 364, 377 defining currency values and 31 formatting 186 Decimal parameter 35 decimal placeholder 188, 348 decimal values. See decimal numbers declaration statements 146 declarations arrays 16-18, 149, 243, 450 class 100, 101, 154 constants 23, 108 enum types 22, 155 functions 40, 144, 200 indefinite loops 150 procedures 414 restrictions for 14, 17, 24 subprocedures 39, 423 typing 343

declarations (continued) user-defined types 33 variables 15, 16, 148, 243, 414, 448 caution for 149 Declare statement See also declarations C functions and 46, 47, 48, 49 described 143 exiting 154 Declare...End Declare statement 146 decoding binary data 110 decoding function 67 default data type 23 default date and time values 88 default directories copy operations and 174 delete operations and 279, 381 rename operations and 324 returning current 119 default paths 15 default values 21 degrees 82 DEL command (DOS) 279 deleting directories 381 disk files 279, 280, 404 leading spaces 29, 184, 301, 447 trailing spaces 29, 385, 447 delimiters 463 depreciation 142, 411, 436 derived classes 100 design tools 5 designers. See e.Report Designer; e.Report **Designer** Professional designs adding source code to 5, 40, 72 declaring string data and 25 developing 5 embedding images in 110 developers 5 developing applications 5 developing reports 4,5 development languages 4 development tasks 38, 53, 61 DHTML formats 242, 278 DHTML reports formatting functions for 66

getting viewing formats for 241 scaling 230 searching 231, 277 DHTMLLong formats 242 DHTMLRaw formats 242 dialog boxes adding buttons 318, 321 adding icons 319, 322 adding title bars 320, 323 displaying dynamically 320 displaying user-defined messages 318, 321 Dim keyword 100 Dim statements arrays and 17 described 148 Java objects and 53 variables and 14, 16, 21, 25 variant data and 24 directories accessing files in 15 adding to search paths 171, 182 changing 92, 93 creating 313 deleting 381 getting attributes 213 getting default 119 moving 324 moving files between 323 removing files from 279 renaming 323 testing existence of 177 directory labels 215 directory paths *See also* search paths calling C functions and 48 copying files and 174 creating Java classes and 115 default for copy operations 174 default for delete operations 279 deleting files and 279 getting absolute 181 setting system specific 176 specifying partial 93 storing default 15 disjunction operator 9,490 disk drives changing 92, 93, 94

entering in string expressions 120 getting default directory for 119 specifying default 94 display formats 241, 278 displaying class names 217, 218 Clipboard content 401 code examples 72 custom browser output 230 data 356, 439 date and time values 32, 131, 137 error messages 81, 334 headlines 223, 224, 225, 403 messages 199, 260, 318, 321 reports 241 search results 231 string values 25, 28, 220 displays 220 divide by zero errors displaying messages for 167 example of 162 preventing 387 division operations 387 division operators 8, 474, 475, 488 DLLs accessing Java objects and 53 accessing procedures in 84, 143 calling C functions in 46, 48 calling subprocedures from 84 disabling type checking for 18 passing incorrect types to 86 specifying 48, 144 Do While keyword 43 Do...Loop statement 43, 150, 168 document files. See report object instance files documentation xi documents See also reports documents. See reports dollar sign (\$) character as currency symbol 189 as type declaration symbol 21 DOS command processor 408 DOS command window 107 DOS file handles 173 DOS shell 407 dot notation 34

Double data type See also double values assigning values to 19, 30 converting to 90 mapping C functions to 50, 52 mapping to Java types 54 returning from C functions 52 double quotation mark (") character date formats and 191 literal strings and 29 numeric formats and 187 print messages and 96 double type 54 double type-declaration symbol 21 double values See also Double data type; floating point numbers assigning to integers 6 defining as constant 23 defining as variants 24 returning 90 valid ranges for 19 double-declining depreciation 142 double-precision numbers. See double values download formats 231, 277 drawing functions getting escaped characters for 433 setting CSS styles for 434 setting font values for 431 setting numeric values for 429 setting SVG attributes for 427, 428 DrawOnChart method 427, 433, 434 drives changing 92, 93, 94 entering in string expressions 120 getting default directory for 119 specifying default 94 duplicate names 42, 54, 424 DWBContext value 213 dynamic arrays clearing 160 declaring 17, 149, 244 defining list of values and 291 getting number of elements in 281 resizing 368, 370 dynamic text controls 224 dynamic-link libraries. See DLLs

#### Ε

E- or e- format symbols 188 E\_JAVAEXCEPTIONOCCURRED error 56 E JVMCLASSNOTFOUND error 56 E\_JVMCLASSPATHNOTFOUND error 56 E\_JVMCREATEJVMFAILED error 56 E\_JVMCREATEOBJECTFAILED error 56 E\_JVMINVALIDJAVAHANDLE error 56 E IVMMETHODFIELDACCESSFAILED error 56 E\_JVMTYPECONVERSIONFAILED error 56 e.Analysis reports 231 e.Report Designer 6 e.Report Designer Professional copying code examples to 72 creating source files with 72 developing designs and 5 extending functionality of 4 getting command-line arguments for 107 launching from command prompt 107 running recursive statements and 6 e.Reporting Server. See iServer e.reports. See reports E+ or e+ format symbols 188 editors 46 Else keyword 44, 254 Elself keyword 44, 254 embedding images 110 Empty keyword 23, 272, 274 empty strings 52 empty values 23, 274 encoding 25, 80, 110, 340 encoding function 110 Encyclopedia volumes 242 End If keyword 254 End keyword 154 End statement described 154 Exit statements vs. 169 end-of-file indicator 158 Enum statement 22, 155 enum type 22, 155 enumerations (enums) 19, 20, 22 Environ function 157 Environ<sup>\$</sup> function 157 environment functions and statements 64

environment variables getting specific settings 157 loading C libraries and 48 EOF function 158 equal sign (=) character as assignment operator 5,9 as comparison operator 8, 481 equivalence operator 10, 483 Eqv operator 10, 483 ERASE command (DOS) 279 Erase statement 160 Erl function 162 Err function 56, 163 Err statement 164 error codes creating user-defined 164, 167 getting 163 returning messages for specific 166 setting 164 Error function 166 error messages creating user-defined 164, 167 displaying 81, 334 getting 166 error numbers. See error codes Error statement 167 error status functions 65 error trapping functions and statements 65 Error\$ function 166 errors accessing run-time 162, 163, 164 calling Java methods and 53 creating Java objects and 56, 225 creating routines for 167, 374 defining currency values and 31 defining variants and 24 disabling handlers for 334 displaying messages for 81, 334 generating user-defined 164, 167 getting for Java classes 114, 115 handling run-time 333 preventing divide by zero 387 recovering from 373 running recursive statements and 6 setting screen attributes and 221 simulating 167 structuring reports and 33

trapping 82 escape characters 111, 432 events 5,410 example code segments 12,72 exceptions 56, 225 exclamation point (!) character as format symbol 196 as type declaration symbol 21 Exclamation Point icon 319, 322 exclusion operator 9, 492 executable files generating 107 getting Factory version for 222 getting version number for 230 running from DOS 407 executing applications 4, 107, 408 executing programs 373, 408 Exit Do keyword 152 Exit Do statement 44 Exit For keyword 197 Exit For statement 44 Exit Function keyword 202 Exit Function statement 44 Exit statement 168 Exit Sub keyword 425 Exit Sub statement 44 Exp function 170 expiration age 404 exponentiation 30, 170, 188, 299, 476 exponentiation operator 7,475 exporting data 464 expressions adding functions to 40, 42 assigning to constants 108 assigning values to variables and 6 bitwise comparisons in 478, 480, 491 calculating time values in 129 calling procedures from 42 checking for null values in 274 comparing 8, 394, 481 computing values of 6,7 controlling statement flow and 44 converting to dates 87, 270, 345 converting to numbers 91, 98, 103, 116 converting to strings 118 converting to variants 121, 122 creating 6

entering literal strings in 29 entering specific dates in 89 entering time values in 88, 136 estimating year from 88 evaluating 255 formatting date values and 190, 191 formatting numeric data and 187, 189 formatting string values and 27 formatting time values and 194 getting substrings in 302, 305 inserting blank spaces and 356 mixing data types in 6 mixing type formats in 185 operator precedence in 469 printing output and 356 returning absolute values 74 returning logical values 9 returning null values 7 returning time values from 132, 250, 308, 387 setting SVG values and 429, 433, 434 testing equality 483 translating character codes 79, 80, 95, 97 trigonometric functions in 501 ExtendSearchPath function 171 external functions 4, 46, 48 See also C functions external libraries 4, 46 external objects 57 external procedures 42, 143 external programs 408

#### F

Factory service 222, 228, 409 FactoryReportContext value 229 False keyword 19, 187 false values 8, 481 *See also* Boolean values fatal errors 6, 334 fields creating Java objects and 53 importing data and 464 Java objects and 54 mapping to variables 400 writing to disk files 356 file access types 336 file buffers allocating 336 flushing 105 reading 372 setting size 338 writing to 337 file descriptors 158 file handles 173 file I/O functions and statements 66 file I/O operations 286, 287, 335 file IDs 77 file information functions and statements 66 file management statements 66 file mode. See file type attributes file modes 172, 336 file names changing 323 getting 228, 374 resolving relative 181 specifying 336 storing 15 file numbers getting unused 199 reusing 105 setting 257, 337 file pointers 391, 392 file type attributes getting 172, 213 setting 398 FileAttr function 172 FileCopy statement 174 FileDateTime function 175 FileExists function 177 FileLen function 178 files *See also* specific type accessing 336 adding search index to 76 archiving report 403 assigning permissions 337, 338 checking input position in 158 closing 105, 372 copying 174 creating multiple report 402 creating persistent 326 creating temporary 106, 159, 173 deleting 279, 280, 404

determining if changed 176 encoding binary 110 error handling for 165 getting current position in 291, 392 getting date/time stamps for 175, 179 getting end-of-file indicator 158 getting number of bytes from 261 getting objects in 227 getting size 178, 298 locking/unlocking 293, 337, 339 moving 323 opening 165, 199, 292, 336 printing to disk 438, 461 reading 208, 256, 259, 289 renaming 323 running from DOS 407 searching for 171, 181 setting position in 390 testing existence of 177 writing to 354, 357, 372, 463 FileTimeStamp function 179 filters 112 financial functions and statements 67 FindFile function 181 fiscal quarters defining formats for 193 getting difference between 129 getting specified 132 Fix function compared to Int 265 described 182 Fixed format 186 Fixed keyword 186 fixed point numbers 31 fixed-size arrays 17, 160, 414 fixed-width fonts 356, 438 float data types 54 float type 54 floating point numbers assigning values 30 defining currency values and 31 returning 8, 90, 474 storing date variables as 31 storing double values as 19 storing time as 443, 445 fontFace parameter 233 fonts 25, 220, 223, 356, 430

For...Next statement 44, 169, 197 format characters creating user-defined formats with 187, 192, 194, 195 entering in expressions 184 locale-specific date conversions 345 format expressions creating 184 date and time values in 190, 191, 194 numeric values in 187, 189 string values in 195 Format function 27, 184 format patterns 27, 184 See also format characters format symbols 346 format symbols. See format characters Format<sup>\$</sup> function 27, 184 formats 346 automating 38 creating user-defined 187, 191, 195 described 191 displaying string values and 28 getting DHTML viewing 241 getting search 231 placing restrictions on 20 specifying locale-specific 28, 226 testing search 277 formatting dates 32-33, 125, 184, 190 numbers 184, 185 numeric values 27 output 463 strings 27-28, 184, 195 time values 88, 191, 194 variant data 27 fractions as special characters 25 date and time values and 32 floating point numbers and 30 removing 98, 182 frames 101 FreeFile function 199 freeing resources 398 function categories 61 Function keyword 144 function names 48, 49, 201, 203

Function statement 40 *See also* functions; procedures Function...End Function statement 200 functions See also C functions; procedures accessing external 4, 46, 48 adding arguments to 40, 201 alphabetical reference for 71 assigning data types to 146, 203, 204 building arrays and 18 calling 40, 42 comparing string values with 28 converting data types with 35 converting string values with 31 declaring 40, 144, 200 defining constants in 108 defining variant data and 24 exiting 44, 154, 169, 202 formatting date or time values with 32 formatting string values with 27 manipulating strings with 26 naming 201, 203 nesting 204 overloading 201 performing case conversions with 28 removing leading or trailing spaces 29 returning binary data 26 returning null dates 346 returning null values 7 running from reports 73 with no parameters 204 fundamental data types. See data types future dates 127 future values 205, 266 FV function 205

#### G

General date format 191 General number format 185 general purpose libraries 238 generating executable files 107 reports 5, 43, 409 source code 5 Get statement 208 GetAFCROXVersion function 211 GetAppContext function 212, 229 GetAttr function 213 GetClassID function 216 GetClassName function 217 GetClipboardText function 219 GetDisplayHeight function 220 GetFactoryVersion function 222 GetFontAverageCharWidth function 223 GetHeadline function 224 GetJavaException function 56, 225 GetLocaleAttribute function 226 GetLocaleName function 227 GetOSUserName function 228 GetPId function 228 GetReportContext function 229 GetReportScalingFactor function 230 GetROXVersion function 230 GetSearchFormats function 231 GetServerName function 232 GetServerUserName function 233 GetTextWidth function 233 GetUserAgentString function 234 GetValue function 234 GetValueType function 236 GetVariableCount function 238 GetVariableName function 240 GetViewPageFormats function 241 GetVolumeName function 242 GetWindowsDirectory procedure 146 global constants 146 Global keyword 108 global procedures 38, 39, 40 global scope 14, 38 Global statement 14, 15, 243 global variables assigning values to 15 declaring 15, 146, 148, 243 caution for 245 initializing 15, 245 GoTo keyword 334 GoTo statement 246 graphics files converting binary 110 searching 171 graphics functions and statements 67 greater than operator (>) 8, 481 greater than or equal to operator ( $\geq$ =) 8, 481 Grouping parameter 35

#### Η

h or hh format symbols 194 Header.bas 214 Headline parameter 224 headlines 224, 402 Hex function 248 Hex\$ function 248 hexadecimal notation 249 hexadecimal numbers assigning to variables 249 converting decimal numbers to 248, 249 defining colors with 20, 364, 377 getting values 454 hidden files 214, 399 hour counting 129 formatting 194 getting minutes in 308 getting seconds in 354, 387 returning 250 Hour function 250 HTML elements 230 HTTP requests 234 hyperbolic cosecant 501 hyperbolic cosine 501 hyperbolic cotangent 501 hyperbolic secant 501 hyperbolic sin 171 hyperbolic sine 501 hyperbolic tangent 501 hyperlinks executing 78 searching for targets 77 hyphen (-) character as numeric operator 8, 473 as wildcard 487 numeric formats and 189

#### 

I/O functions and statements 66 I/O operations 286, 287, 335 icon style constants 319, 322 icons 319, 322 IDABORT constant 320 **IDCANCEL** constant 320 identifiers 49, 57 **IDIGNORE** constant 320 IDNO constant 320 IDOK constant 320 **IDRETRY constant 320 IDYES constant** 320 If keyword 253 If...Then...Else statement 44, 154, 253 Ignore buttons 319, 320, 322 IIf function 255 images 67, 110, 171 Imp operator 10, 484 implication operator 10, 484 implicit date conversions 346 implicit declarations 14 importing data 346, 464 incorrect variable types 244 index numbers (arrays) 16 indexed searches 76, 78, 235 inequality 9,481 Information icon 319, 322 initializing arrays 18, 368 Java objects 115 variables 15, 21, 148, 245 input assigning to variables 209, 289 closing files for 105 creating alerts for invalid 83 defining error messages for 166 getting record size for 286, 287 opening files for 335 reading from sequential files 289 testing end-of-file position 158 input filters 112 Input function 259 input functions and statements. See I/O functions and statements Input mode 173, 338 Input statement 256 Input<sup>\$</sup> function 259 InputB function 261 InputB\$ function 261 instance methods 54 instance variables aliasing 400

counting 238 creating 100 getting values 234 instantiating classes 62, 115, 325, 326, 397 objects 16, 34, 326, 327 InStr function described 261 RevInStr compared with 375 InStrB function 26, 263 Int function described 264 Fix function vs. 183 int type C functions and 49, 52 Java objects and 54, 55 Integer data type See also integers assigning 19, 30 converting to 98 mapping C functions to 49 mapping to Java types 54, 55 returning from C functions 52 integer type-declaration symbol 21 integers See also numbers assigning values 19,30 calculating date or time values and 32, 136, 194 checking conversion status 275 comparing 481 converting to dates 88, 89, 122, 124 converting to doubles 90 converting to longs 103 converting to singles 116 converting to variants 121 defining as constant value 23 defining as variants 24 expressing as exponents 171, 475 getting largest 264 getting square root of 202, 204, 413 returning 8, 98, 182, 475 rounding 6 testing 406 interest rates 266, 366 internal rate of return 311 international alphabets 25

invalid arguments 51,78 invalid data types 256 invalid identifiers 49 invalid names 49 inverse cotangent 501 inverse hyperbolic cosecant 502 inverse hyperbolic cosine 501 inverse hyperbolic cotangent 502 inverse hyperbolic secant 502 inverse hyperbolic sine 501 inverse hyperbolic tangent 502 investments See also annuities getting depreciation for 142, 411, 436 getting interest rates 366 getting net present values 330 getting principal payments 351 getting rate of return 268, 311 IPmt function 266 IRR function 268 Is keyword 396 Is operator 9,485 IsDate function 33, 270 IsEmpty function 272 IsKindOf function 273 IsNull function 274 IsNumeric function 275 ISO date format 346 IsPersistent function 276 IsSearchFormatSupported function 277 IsViewPageFormatSupported function 278 iterations 6

#### J

Java classes 53, 114, 115 Java Development Kit 53 Java exceptions 225 Java objects accessing 53 converting arrays for 55 converting data types for 54–56 creating 53–54, 115 debugging 57 error handling for 56 initializing 115 Java Runtime Environment 53 Java Virtual Machine 53 JDK software 53 job status files 409 jobs. *See* requests JRE software 53 JVMs 53

#### Κ

keywords Actuate compiler and 495 C functions and 49 formatting functions 191 numeric formats and 185 operators as 7 restrictions for 12 Kill statement 279

#### L

l format character 345 language support (programming) 4 launching report applications 107, 408 LBound function 280 LCase function 29, 282 LCase\$ function 29,282 LD\_LIBRARY\_PATH variable 48 leading spaces adding to strings 27, 184 defined 301 reading 260, 261 removing 29, 301, 447 leap years 489 Left function 283 Left\$ function 283 LeftB function 26, 284 LeftB\$ function 284 Len function 285 Len keyword 337 LenB function 286 less than operator (<) 8, 481 less than or equal to operator ( $\leq$  ) 8, 481 Let keyword 287 Let statement 287 Lib keyword 48, 144 LIBPATH variable 48 libraries accessing functions in 4, 46, 48

creating general purpose 238 unloading external 46 library name argument 48 Library Organizer 46,72 Like operator 9, 470, 486 line breaks 96, 320 line continuation character 11 Line Input statement 289 line numbers getting 162 setting 247 line widths 461 linefeed characters character code for 96 reading 260, 261 ListToArray function 290 literal characters date expressions and 191 numeric expressions and 187 numeric formats and 189 literal strings adding to expressions 29, 187, 191 copying portions of 302 getting characters in 283, 285, 379, 380 replacing portions of 304, 307 returning 25 trimming 386 literal values dates as 31, 32 null as 463 static arrays and 450 symbolic constants as 108 time as 32 Loc function 291 local scope 14 local variables clearing 414 compared to static 414 declaring 16 getting number of 239 preserving 201, 424 locale maps 35 locale names 227 locales formatting currency values for 189 formatting data for specific 185 formatting string values and 27

getting attributes for 226 getting current run-time 227 specifying dates for 31, 89, 125, 190, 344 specifying numeric values for 33, 35 specifying string data for 25, 27 supporting information for 35 Lock Read keyword 339 Lock Read Write keyword 339 Lock Write keyword 339 Lock...Unlock statement 293 locks 293, 337, 339 LOF function 298 Log function 299 logarithms 299, 502 *See also* exponentiation logical expressions 9 logical operators described 9 order of evaluation for 469 Select Case statements and 396 logical values assigning numeric values for 19,43 comparing 9,396 evaluating 177 getting 187 login names 228, 233 Long data type See also long values assigning 19, 30 converting to 103 mapping C functions to 50 returning from C functions 52 long date formats applying 191, 192 converting 87, 124 Long time format 191 long type C functions and 50, 52 Java objects and 54 long type-declaration symbol 21 long values 19, 24, 30 Loop While keyword 43 loops branching 246, 460 building arrays for 17 controlling program flow and 43 creating 43

loops (*continued*) defined 460 defining indefinite 150 ending immediately 197 executing variables and 16, 238 executing with conditions 253, 459 exiting 44, 168 interrupting 152 nesting 44, 152, 198 programming constructs for 62 repeating specified number of times 197 repeating statement blocks 459 stepping through 416 temporarily suspending report execution and 410 lower bounds (arrays) 17 lowercase characters 11, 29 lowercase conversions 196, 282, 453 LSet statement 300 LTrim function 29, 301 LTrim<sup>\$</sup> function 301

#### Μ

m format character 345 Make3Files procedure 106 MakeDataFile procedure 159 mantissa 30 See also scientific notation mapping column names to data rows 400 math functions and statements 68 mathematical operations arithmetic operators for 7, 469 concatenation and 471 date or time values and 32 operator precedence in 469, 470 preventing divide by zero errors 387 restrictions for 25 variants and 23, 24 MB\_ABORTRETRYIGNORE constant 319, 322 MB\_APPLMODAL constant 319, 322 MB\_DEFBUTTON1 constant 319, 322 MB\_DEFBUTTON2 constant 319, 322 MB\_DEFBUTTON3 constant 319, 322 MB\_ICONEXCLAMATION constant 319, 322

MB\_ICONINFORMATION constant 319, 322 MB\_ICONQUESTION constant 319, 322 MB ICONSTOP constant 319, 322 MB\_OK constant 319, 322 MB\_OKCANCEL constant 319, 322 MB\_RETRYCANCEL constant 319, 322 MB\_SYSTEMMODAL constant 319, 322 MB\_YESNO constant 319, 322 MB\_YESNOCANCEL constant 319, 322 Me keyword 39 Medium date format 191 Medium time format 191 memory allocating 148, 243 building arrays and 17 de-allocating 46, 160 storing values in 19 memory cleanup function 46 message boxes adding title bars 320, 323 displaying dynamically 320 displaying text in 199, 260, 318, 321 showing current directory 120 MessageBeep procedure 85 messages *See also* error messages adding line breaks 320 displaying 199, 260, 318, 321 drawing charts and 431 entering invalid input and 83 generating reports and 409 inserting quotation marks in 96 printing 96 Method Editor 72 methods See also functions; procedures calling class-specific 38, 101 creating procedures and 5, 38 defining variables local to 16 invoking Java 53, 54 overriding 5 pasting text blocks to 30 referencing specific objects in 39 Mid function 302 Mid statement 304 Mid\$ function 302 Mid\$ statement 304

MidB function 26, 305 MidB statement 307 MidB\$ function 305 MidB\$ statement 307 military time getting 191, 194, 441 specifying 88, 133 minus sign (-) character in strings 27 Minute function 308 minutes counting 129 formatting 194 getting seconds in 354, 387 returning 308 MIRR function 311 mismatched type errors 4 mixing colors 364 MkDir command (DOS) 314 MkDir statement 313 Mod operator 8, 488 modal buttons 322 modal dialog boxes 319, 322 See also message boxes modality constants 319, 322 modified rate of return 311 modifying. See changing modules accessing procedures in 39, 144 user-defined types and 34 modulus 8, 488 monetary values 18, 31 *See also* currency values monospaced characters 356, 438 month entering in expressions 136, 316 getting day of 140 getting difference in 129 getting specific 132, 315 month format characters 193 Month function 32, 315 moving files 323 MsgBox function 318 MsgBox statement 321 multibyte characters 27, 79 multidimensional arrays associating with classes 100 calling C functions and 51

changing at runtime 281 creating user-defined variables and 449 declaring 17 defining dimensions 148, 244 determining contents 199 getting lower bounds 280, 281 getting upper bounds 452 setting contents of 199 setting lower bounds 340 multiline statements 11, 147, 154 multiplication operator 8, 470

#### Ν

n or nn format characters 194 name conflicts 204 Name property 39 Name statement 323 names aliasing non-standard 49 arguments as 41 C function calls and 47, 48 calling procedures and 41, 42 determining fully qualified 77 getting Encyclopedia 242 getting parameter 41 implicit declarations and 14 Java methods and 54 reserved words as 495 reusing variable 16 special characters in 11 naming constants 11, 108 functions 201, 203 Java fields 53 procedures 11, 43, 415 source files 72 subprocedures 424 temporary files 228 variables 11, 414 naming conventions 11, 49 negation operator 9, 473, 489 negative exponents 7 negative numbers comparing 265 displaying 188 financial transactions and 266

negative numbers (continued) formatting currency values and 186 formatting string values and 27 representing dates 32, 88, 127 rounding 183 specifying 473 testing for 406 valid ranges for 19 nested classes 101, 218 nesting control structures 44, 152, 253 functions 204 loops 44, 152, 198 statements 169 structures 20 subprocedures 424 user-defined types 34 net present value 67, 330 New keyword 397 new line characters 30, 463 new lines inserting 463 setting line breaks for 96, 320 setting line widths for 461 NewInstance function 325 NewPersistentInstance function 326 Next keyword 198, 373 No buttons 319, 320, 322 No keyword 187 nonprintable characters 96 non-standard names 49 not equal to operator (<>) 9, 481 Not operator 9, 489 Nothing keyword 398 Now function 327 NPer function 328 NPV function 330 null character 51 null digit placeholder 187 Null keyword 52, 272, 274 null pointers 52 null values assigning 55, 287 comparing strings and 28 concatenating strings and 10 defining variants as 23, 463 empty values compared to 23

format specifier for 187 passing to C functions 52 returning 7,272 testing for 274, 481 writing as literals 463 null-terminated strings 51 number sign (#) character as type declaration symbol 21 as wildcard 487 date and time formats and 31, 32 numeric formats and 187 numbers assigning to variables 255, 286, 287, 288 assigning values 30 calculating difference between 473 checking conversion status 275 comparing 264, 481 computing values 7 conversion functions for 63 converting for specific locales 33, 35 converting strings to 31, 116, 183, 264, 348 converting to currency 85 converting to dates 32, 88, 89, 122, 124, 271 converting to doubles 90 converting to integers 98 converting to longs 103 converting to singles 116 converting to strings 27, 184, 248, 332, 417 converting to variants 121 declaring enum types for 22 defining as constant value. See constants defining date and time values as 32, 127, 136, 194 defining variants as 23, 24, 287, 358 dividing 8, 474, 475, 488 entering in strings 25 expressing as exponents 171, 475 extracting from strings 276, 454 financial transactions and 266 formatting 27, 184, 185, 187 generating random 104, 169, 364, 383 getting absolute values 74 getting hexadecimal 248, 249 getting number of digits in 254 getting octal 332 getting square root 202, 204, 413 getting values 454
inputting as error code 164, 166 parsing 348 reading from sequential files 256 removing fractional part 98, 182 removing leading spaces 184 representing date and time values 32 returning Boolean values 19, 43 rounding 6, 30, 31, 98, 99, 183 storing 19 testing 406 valid ranges for 19 numeric conditions 43 numeric constants. See constants numeric data types 19, 30, 31, 32 See also specific type numeric expressions adding arithmetic operators 7 bitwise comparisons for 478, 480, 491 conditionally executing statements and 43 converting data types and 35 converting decimal/hexadecimal values 248 entering time values in 136 evaluating 275 formatting data and 187, 189 inserting blank spaces and 356 printing output and 356 returning absolute values 74 returning character codes 95, 97 returning line widths 461 setting SVG values and 429 numeric variables 30, 31, 245

# 0

object handles 53, 57 object IDs 57, 227 Object Linking and Embedding. *See* OLE object reference variables comparing 9, 485 creating Java objects and 53 defining 397 getting 235 initialization values for 245 passing to C functions 52 Object type 51, 54 object types 57, 217, 276

object-oriented programming 4 objects See also Java objects accessing procedures for 38 building arrays of 16–18 calling class-specific methods for 38 checking for given type 217 copying variables between 112 creating 34 customizing aging settings for 404 defining attributes of 16, 34 determining if persistent or transient 276 discontinuing associations 398 freeing resources for 398 getting class information for 216, 217, 273 instantiating 16, 326, 327 referencing 34, 39, 397, 485 Oct function 332 Oct\$ function 332 octal notation 332 octal numbers 332, 455 Off keyword 187 OK buttons 319, 320, 322 OLE applications 65 On Error statement 333 On keyword 187 On/Off format 187 online documentation xi online help 72 Open statement 335 opening code example window 72 code examples 72 disk files 165, 199, 292, 336 Library Organizer 72 operands 6, 473 operating environments See also UNIX systems; Windows systems getting login names for 228 getting system attributes for 214 getting system dates/time 125, 327, 441 getting variables for 157 locking files and 293 setting file type attributes for 398 storing information for 157 task-specific functions and statements 64

operating systems. See operating environments operators assignment statements and 5 categorized 68 described 469 expressions and 6 naming conventions and 12 null values and 7 order of evaluation for 469 types described 7-10 variants and 23, 24 Option Base statement 340 Option Compare statement 28, 342, 487 Option Strict statement 20, 343 Or operator 9, 490 order of evaluation 469 output adding spaces to 356 controlling date conversions for 347 defining line widths for 461 formatting 463 getting record size for 286, 287 printing 355, 356, 438 scaling 230 sending to disk files 105, 335, 354, 357 output formats 28 output functions and statements. See I/O functions and statements Output mode 173, 338 overflow errors currency conversions 85 numeric conversions 90, 98, 103, 116 variant types 24, 121 overloading functions 201 procedures 42, 424 overriding methods 5 overriding operator precedence 469

#### Ρ

p format symbol 345 parameter lists. *See* argument lists parameter names 41 parameters *See also* arguments

checking values 38 defining locale-specific rules and 35 parentheses () characters in expressions 469 in procedure declarations 41 numeric formats and 189 ParseDate function 344 ParseNumeric function 348 parsing dates 132, 344, 347 numbers 348 strings 226, 374 passing arguments by reference 41, 48, 84 passing arguments by value 42, 48, 84, 145 pasting text blocks 30 path names 93 PATH variable 48, 157 paths See also search paths calling C functions and 48 copying files and 174 creating Java classes and 115 default for copy operations 174 default for delete operations 279 deleting files and 279 getting absolute 181 setting system specific 176 specifying partial 93 storing default 15 pattern matching 487 pattern-matching operator 9, 486 patterns date or time formats 192, 194 string comparisons 9, 486 user-defined formats 187, 195 payment due 128 PDF formats 242, 278 percent (%) character as type declaration symbol 21 numeric formats and 188 Percent format 186 Percent keyword 186 percentage placeholder 188 percentages 186, 188 periodic cash flows 268, 311, 330 permissions. See privileges persistent classes 326

Persistent keyword 149 persistent objects 77, 276 persistent variables 149 phone numbers 25 plus sign (+) character as line continuation symbol 11 as numeric operator 8, 471 commenting code and 11 numeric formats and 189 Pmt function 349 pointers passing to C functions 18, 50, 52 repositioning file 391, 392 portability 4 positive exponents 7 positive numbers See also integers; numbers converting to strings 27, 184 displaying 188 financial transactions and 266 representing dates 127 testing for 406 valid ranges for 19 pound sign (#) character as type declaration symbol 21 as wildcard 487 in date and time values 31, 32 in numeric formats 187 power. See exponentiation PPmt function 351 precedence 469 PreciseTimer function 354 predefined functions 40 predefined methods 5 present value 206, 266, 361 Preserve keyword 17, 368 principal 351 print position 439 Print statement 354 print zones 356 printers getting text height for 220 sending control codes to 96 printing functions and statements 67 printing output 355, 356, 438, 461 printing unformatted data 354 PrintReportContext value 229

privileges 76, 337, 338 procedural functions and statements 68 procedure names 11, 41, 42 procedures See also subprocedures accessing 5, 38 adding arguments 41–42, 84 adding C functions to 46, 47, 48, 49 adding comments 11, 371 assigning data types to 41, 43 calling 39, 42, 42, 143 creating 5, 38, 40 declaring as static 414 declaring functions as 40 declaring global 38, 39, 40 defining variables in 14, 16, 148 disabling type checking for 18 error handling in 334 executing specific tasks with 5 exiting 44, 169, 202, 425 naming 11, 43, 415 overloading 42, 424 passing values to 41, 42, 48, 84 preserving variables in 201, 415, 424 returning data types from 144 returning values from 38 scoping 38 suspending program execution from 416 unconditional branching in 246 process IDs 228 programmers 5 programming languages 4 programming tasks 38, 53, 61 programs accessing procedures in 38 adding comments to 371 controlling flow of 62, 84 debugging. See debugging developing 5, 34, 38 exiting 62, 106 getting available variables for 78 handling run-time errors in 333 incrementing counters for 199 pausing 62 running at specified place 373 running from DOS shell 408 running with conditions 81

programs (continued) stopping execution 82, 154, 416 timing 442 prompts 447 properties archiving reports and 404 creating data types and 33 creating reports and 5 naming variables and 78 referencing specific objects for 39 proportionally spaced characters 356 pseudo-random sequences 364, 383 Put statement 357 PV function compared to NPV function 331 described 361

### Q

q format character 193 QBColor function 363 qualified names 77 quarter defining formats for 193 getting difference in 129 getting specified 132 Query Builder 400 question mark (?) character as format symbol 345 as wildcard 487 Question Mark icon 319, 322 quotation mark characters date formats and 191 in code 371 literal strings and 29 numeric formats and 187 print messages and 96 reading 260, 261

## R

radians 82 radical prefixes 249, 333 radix characters 348 random files accessing 336 getting current position 292 getting current position in 393

getting record size 286, 287 locking 294 opening 338, 357 reading from 208, 209 setting position in 391 testing input position 159 writing to 359 Random mode 173, 209, 338, 357 random number generator 364 random numbers converting 104 creating text files for 159 defining seed values for 365, 442 generating 104, 169, 364, 383 specifying range for 384 random-access file mode 338 Randomize statement described 364 Rnd statement and 384 Timer function and 442 range of values array subscripts 369 character lists 487 color codes 377 comparing against 395 hexadecimal numbers 249 octal numbers 333 overflow errors and 24 random numbers 384 type definitions and 18, 20 variant declarations and 24 Rate function 366 rate of return 67, 268, 311 Read keyword 338 Read Write keyword 338 read-only files 214, 399 read-only permissions 338 real types. See Double data type; Single data type record numbers 208, 359 record structures 448, 449 records determining size 286, 287 importing data and 464 locking 294 misreading 210 reading from disk 208

specifying length 338 writing to disk 357 rectangle controls 40 recursive statements 6, 101, 204, 426 ReDim statement described 368 Dim statement and 148 Erase statement and 161 references assigning to variables 397 closing files and 105 creating objects and 34, 39, 397 defining multiple 485 passing arguments by 41, 48, 84 relational operators 481 See also comparison operators; comparisons relative file names 181 Rem keyword 11 Rem statement 371 remainders 8,488 removing type restrictions 49 Rename command (DOS) 323 renaming C functions 49 directories 323 files 323 repeating blank spaces 356 repeating characters 420, 421 report aging constants 403 report bursting 76, 402 report components adding to designs 5 getting class information for 77, 217, 218 report designers. See e.Report Designer; e.Report Designer Professional report designs adding source code to 5, 40, 72 declaring string data and 25 developing 5 embedding images in 110 Report Encyclopedia. See Encyclopedia volumes report files *See also* specific type adding search index to 76 archiving 403

assigning permissions 337, 338 closing 105, 372 copying 174 creating multiple 402 creating persistent 326 deleting 279, 280, 404 error handling for 165 getting current position in 291, 392 getting date/time stamps for 175, 179 getting end-of-file indicator 158 getting objects in 227 getting size 178, 298 locking/unlocking 293, 337, 339 moving 323 opening 165, 199, 292, 336 renaming 323 searching for 171, 181 setting position in 390 specifying default 66 testing existence of 177 writing to 355, 357, 372, 463 report object design files 181 report object executable files generating 107 getting Factory version for 222 getting version number for 230 report object instance files 76, 227, 326, 402 report object web files 403 report sections 101 report servers. See iServer; servers ReportContext variable 229 reports creating 5 customizing 238 designing. See report designs developing 4,5 displaying 241 embedding images in 110 generating 5, 43, 409 getting contexts 229 opening as read-only 338 restricting access to 293 retrieving data for 76, 208, 256, 289 retrieving specified number of characters for 259 running functions from 73 specifying default 402

reports (continued) structuring related data in 33 temporarily suspending 410 requests creating headlines for 402 getting headlines for 224 getting user agent for 234 reserved characters date formats 192 numeric formats 187 string formats 195 time formats 194 reserved words Actuate compiler and 495 C functions and 49 formatting functions 191 numeric formats and 185 operators as 7 restrictions for 12 Reset statement 372 resizing arrays 17, 368, 370 resources 398 restricting access to reports 293, 338 Resume Next keyword 334 Resume Next statement 373 Resume statement 373 resuming program execution 373 Retry buttons 319, 320, 322 return (cash flow) 268, 311 return values assigning to procedures 40 calling C functions and 47, 52, 55 creating subprocedures and 38 declaring functions for 40, 202 setting conditions for 255 typing 40 RevInStr function 374 RGB color values 364, 376 RGB function 376 Right function 379 Right\$ function 379 RightB function 26, 380 RightB\$ function 380 RmDir command (DOS) 382 RmDir statement 381 Rnd function 383 .rod files. See report object design files .roi files. See report object instance files roles 76 rounding currency values and 31 numeric values and 6, 30, 98, 99, 183 variant data and 264 rounding errors 31 routines converting arguments for 85 creating user-defined errors in 164 defining error handling 167, 374 .row files. See report object web files rows See also records declaring multidimensional arrays and 17 mapping to database columns 400 .rox files. See report object executable files RSet statement 384 RTrim function 29, 385 RTrim\$ function 385 running applications 4, 107, 408 running programs at specified place 373 from DOS shell 408 with conditions 81 run-time errors disabling handlers for 334 getting error codes for 163 getting line numbers for 162 getting messages for 166 handling 333 setting error codes for 164 run-time stack limit 6 RWCString type 50

#### S

s or ss format characters 194 SafeDivide function 387 sample code segments 12, 72 saving source files 72 Scalable Vector Graphics. *See* SVG attributes scaling controls 230 scaling factor 230 scheduling archiving operations 404 Scientific format 186 Scientific keyword 186

scientific notation 170, 171, 186, 188, 476 See also exponentiation scope procedure calls and 38, 42 variables and 14, 16, 245 screen attributes 220 scripting controls 230 search conditions See also search expressions search formats 231, 277 search indexes 76, 78, 235 search paths adding directories to 182 calling C functions and 48 specifying 171 UNIX systems and 120 search results 231 Searchable property 77 searching DHTML reports 231, 277 for data 77 for disk files 171, 181 for special characters 487 graphics files 171 secant 501 Second function 387 seconds counting 129 formatting 194 returning 354, 387 sections 101 security roles 76 seed values 365, 442 Seek statement 390 Seek2 function 392 Select Case statement 154, 394 selection criteria. See search conditions semicolon (;) character numeric formats and 189 printer codes and 356, 438 separators date values 192 locale-specific data and 35 numeric values 349 tab characters 438 time values 194

sequential files accessing 336 getting current position 292 getting number of bytes from 261 getting specified characters in 259 locking 294 opening 338, 339 printing to 354, 438, 461 reading 256, 259, 289 testing input position 159 writing to 354, 463 sequential input mode 338 sequential output mode 338 serial numbers assigning to variables 31, 327 converting dates to 32, 63, 135 converting time values to 64 converting to dates 88 converting to time values 64 formatting 190 getting 32, 135, 442 programming functions for 63, 64 storing 445 server context values 213, 229 server login names 228, 233 server names 232 ServerContext value 213, 229 Set statement 397 SetAttr statement 398 SetBinding function 400 SetBurstReportPrivileges function 76 SetClipboardText function 401 SetDefaultPOSMFile function 402 SetHeadline statement 402 SetStructuredFileExpiration function 403 SetValue function 405 Sgn function 406 share.exe 293 shared files 339 Shared keyword 339 shared libraries 46, 48 Shell function 407 shelling out to DOS 407 SHLIB PATH variable 48 short date formats 32, 191, 192 Short time format 191 short type 55

ShowFactoryStatus statement 409 Sin function 409 sine 409 Single data type See also single values assigning 19, 30 converting to 116 mapping C functions to 50 mapping to Java types 54 returning from C functions 52 single quotation mark (') character 11, 371 single type-declaration symbol 21 single values 19, 23, 24, 30 See also numbers single-byte characters 25 single-dimension arrays 55 Sleep statement 410 SLN function 411 sorting array elements 460 sounds 83 source code See also Actuate Basic accessing examples for 12,72 accessing Java objects and 54 adding comments to 11, 371 adding external functions to 46, 49 adding Java classes 53, 114, 115 controlling statement flow in 44 disabling while debugging 371 generating 5 implicit declarations and 14 naming conventions for 11 pasting text blocks to 30 referencing specific objects in 39 simplifying programming tasks in 38 unassigned data types and 20 writing 5, 10, 11 source files accessing C functions in 46 adding to designs 5 creating 72 declaring procedures in 38, 39, 40 pasting code examples to 73 saving 72 space characters adding to output 356 formatting numeric values and 189

removing from strings 29 repeating 356 returning in strings 412 stripping extra 447 Space function 412 Space<sup>\$</sup> function 412 Spc keyword 356 special characters embedding in strings 25, 29 naming conventions and 11 returning 96 searching for 487 spreadsheets See also e.Spreadsheet reports; Excel spreadsheets Sqr function 413 square roots 43, 202, 204, 413 SquareRoot procedure 204 stack 6 standard data types 18 See also data types Standard format 186 Standard keyword 186 start of day 33 statement block 197 statements See also procedures adding C functions 46, 47, 48, 49 adding comments to 11 adding line breaks to 11 alphabetical reference for 71 branching to 246 building arrays and 16, 17, 18 calling 6 changing order of 43 comparing string values and 28 controlling type definitions and 20, 21 creating 5, 11, 61 declaring enum types in 22, 155 declaring functions in 40, 144, 202 defining as recursive 101, 204, 426 entering multiline 11, 147, 154 executing 39 nesting 169 repeating indefinitely 43 repeating specified number of times 44 static arrays 450

static fields 54 Static keyword 100, 201, 424 static methods 54 static procedures 414 Static statement 14, 16, 414 static variables creating 100 declaring 201, 414, 424 Step keyword 197 Stop icon 319, 322 Stop statement 416 stopping program execution 82, 154, 416 storing values 16 Str function 27, 417 Str\$ function described 27, 417 Format\$ function vs. 184 straight-line depreciation double-declining vs. 142 returning 411 StrComp function 28, 418 string comparison operator 9, 486 string constants copying portions of 302 delimiting 30 embedding special characters in 30 getting first characters 283, 285 getting last characters 379, 380 replacing portions 304 setting values for 23 trimming 386 String data type assigning 25, 288 character limits for 19 converting to 118 mapping to Java types 55 returning 27, 52 string expressions 25, 29 String function 420 string functions and statements 26, 28, 29, 63, 69 String objects 55 string type-declaration symbol 21 String\$ function 420 strings See also String data type adding literal characters to 29

aligning characters in 300, 384 as format patterns 184, 192, 194, 195 assigning to variables 288 building 19,25 changing 304, 307, 422 changing capitalization of 28, 196, 282, 453 comparing 28, 342, 418, 481, 486 concatenating 10, 471, 476 converting dates to 344 converting numbers to 27, 184, 248, 332, 417 converting to arrays 290 converting to currency 85 converting to dates 87, 89, 122, 124, 344 converting to doubles 90 converting to integers 98 converting to longs 103 converting to numbers 31, 116, 183, 264, 348 converting to XML 111 copying 29, 301, 401, 447 copying specific characters in 283, 302, 379 counting characters 285 creating user-defined formats and 187, 191 debugging Java 57 declaring C functions and 49, 51 defining variants as 23 determining height 220 displaying 25, 28, 220 encoding binary images as 110 extracting only numbers from 276 formatting 27-28, 184, 195 getting character codes for 26, 79, 80 getting dates for specified 138 getting first characters 283 getting last characters 374, 379 getting length 285 getting number of bytes in 26, 286 getting number of characters in 26 getting numeric values of 454 getting specific characters 259 getting specified portion of 302, 305 getting starting byte for 263 getting starting position for 261 getting substrings in 26 initializing variable-length 245 parsing 226, 374

strings (continued) reading 210, 257 removing leading spaces 29, 184, 301, 447 removing trailing spaces 29, 385, 447 replacing portions of 304, 307, 422 representing dates 125 returning ANSI characters in 95 returning binary data in 284, 380 returning empty 52 returning from Clipboard 219 returning from external procedures 42 returning literal 25 returning repeating characters in 420, 421 returning with specified number of spaces 412 space characters replacing content 166 storing binary data and 25, 26 trimming 301, 385, 447 truncating 300, 385 writing 358 StringW function 421 StringW\$ function 421 StrSubst function 422 structure members 20 structures 20, 33, 43 See also control structures style sheets 242, 434 Sub keyword 144 Sub statement 39, 47 *See also* subprocedures Sub...End Sub statement 423 subclasses declaring 100 testing for instances of 273 subdirectories See also directories; directory paths adding to search paths 182 creating 313 deleting 381 determining if exists 382 getting names 214 removing files from 279 subprocedures accessing 39 adding arguments 425 assigning data types to 426 calling 42

declaring 39, 423 defining C functions in 47 defining constants in 108 exiting 44, 154, 169, 425 initializing variables and 15 naming 424 nesting 424 passing incorrect types to 86 returning values from 38 transferring control to 84 subroutines 248 subscripts. See arrays; multidimensional arrays substrings combining 10, 476 getting first character 283 getting last character 379 getting starting byte for 263 getting starting position for 261, 374 replacing 304, 307, 422 returning binary data in 284, 380 returning first occurrence of 26 returning specified 302, 305 returning starting position of 374 subtraction operations 7 subtraction operator (–) 8, 473 summary values 471 sum-of-years'-digits depreciation 436 superclasses 100 SVG attributes 427, 428 SVGAttr function 427 SVGColorAttr function 428 SVGDbl function 429 SVGFontStyle function 430 SVGStr function 432 SVGStyle function 434 SYD function 436 symbolic constants 108, 401 symbols 7 syntax (programming). See declarations system attributes 214 system clock See also time formatting 32, 191, 194 getting current time 441 getting specific time for 443 setting 32, 88, 133

specifying start of day for 33 system dates displaying 32 getting current 125, 327 system files 215, 399 system variables 157 *See also* environment variables

### Т

t format character 345 tab characters character code for 96 embedding in strings 30 setting position 356, 438 Tab function 438 Tab keyword 356 tab separated values 231, 278 Tan function 440 tangent 440 telephone numbers 25 temporary addresses 84 temporary calculations 16 temporary files creating 106, 159, 173 determining size 298 getting current position in 292 getting unused file numbers from 200 locking 295 naming 228 reading from 210, 258, 290, 391 writing to 210, 357, 372, 391, 462 testing conditional statements 43 programs. See debugging TestRoutine procedure 442 text adding to headlines 402 aligning 300, 384, 385 changing capitalization of 28, 196, 282, 453 copying 301, 401, 447 defining string variables for 25, 29 determining headline 224 displaying in dialog boxes 199, 260, 318, 321 displaying in title bars 320, 323 getting height 220

getting width 224, 233 pasting 30 text editors 46 text files accessing 336 reading in random mode 209 reading one line at a time 289 reading sequentially 289 reading specified characters from 259, 261 writing to 356, 461 Text keyword 342 text strings. See strings Then keyword 253 third-party libraries 46 thousands separator 188, 349 Time function 441 time functions 346 time functions and statements 63 time separators 194 time serial numbers converting time values to 64 converting to time values 64 formatting 190 getting 442 storing 445 time stamps file creation 175, 179 file modification 175, 179 time values as date interval 88, 126, 129, 132 assigning 19 calculating 32 converting to variants 122 displaying 32 entering in date expressions 88, 133 entering literal values for 32 formatting 88, 191, 194 getting current 445 getting hour 250 getting minutes 308 getting seconds 354, 387 getting seconds past midnight 442 getting seconds to midnight 310, 389, 443, 446 getting specific 443 returning system 327, 441 Time\$ function 441

Timer function 442 TimeSerial function 442 TimeValue function 445 title bars 320, 323 To keyword 294, 395 totals 16 trailing spaces defined 447 removing 29, 385, 447 Transient keyword 149 transient objects 276 transient variables 149 trapping errors 82 trigonometric formulas 501 trigonometric functions 68, 501 Trim function 29, 447 Trim\$ function 447 trimming strings 301, 385, 447 True keyword 19, 187 true values 8, 481 See also Boolean values True/False format 187 truncating numbers 182 truncating strings 300, 385 TSV formats 231, 278 ttttt format characters 195 two-dimensional arrays 17 type checking 18 type definitions (typedefs) 20 Type keyword 19 type mismatch errors 24, 286, 482 Type statement 450 Type...As statement 451 Type...End Type statement 448 type-declaration characters adding to expressions 7 adding to procedures 144 defining currency values and 31 defining variables and 21, 149 listed 21 Typedef keyword 33 types. See data types; specific type

#### U

UBound function 452 UCase function 29, 453 UCase\$ function 29, 453 UCS-2 character sets 421 UCS-2 encoding 80,97 unassigned data types 20 unformatted data 354 Unicode characters 79 uniform annual depreciation 412 uninitialized variables 272 union filters 112 union type 50 UNIX systems calling C functions for 46, 48 checking for files on 177 copying files 174 creating directories 313 deleting directories 381 deleting files 279 getting current default directory for 120 getting date/time stamps for 176 getting file size 178 getting system attributes for 214 renaming files 324 setting file type attributes for 399 unknown object types 57 UnknownReportContext value 229 unloading external libraries 46 unlocking files 293 Until keyword 151 upper bounds (arrays) 17 uppercase characters 11, 29 uppercase conversions 196, 282, 453 URLs 242 user agent string 234 user names getting login 228, 233 getting system information for 64 returning strings as 282 user-defined errors creating messages for 167 defining error codes for 164, 167 generating 167 user-defined formats 187, 191, 195 user-defined functions 42 user-defined messages 318, 321 user-defined types assigning to variables 101, 287, 288, 448 assigning values 19

creating 33–34, 450 declaring enums as 156 nesting 34 passing to C functions 52 user-defined variables assigning data types to 449 assigning values to 450 avoiding type mismatch for 286 declaring 34, 146, 448 initializing 245 users getting login information for 228, 233 placing access restrictions for 76, 293 user-supplied values. *See* input user-triggered events 5

#### V

V\_CURRENCY constant 456 V\_DATE constant 456 V\_DOUBLE constant 456 V\_EMPTY constant 456 V\_INTEGER constant 456 V\_LONG constant 456 V NULL constant 456 V SINGLE constant 456 V\_STRING constant 456 Val function 31, 454, 455 values assigning null. See null values assigning to data types 18 assigning to Java objects 53 assigning to variables 5, 15, 34, 249, 287, 414 assigning to variants 23 changing 41, 42 checking parameter 38 comparing 395 converting to specific type 34 defining constant 20, 23, 108 defining date or time 31 defining numeric data and 30 determining if assigned 272 getting absolute 74 getting numeric 454 passing to procedures 38, 41, 42, 48, 84 placing limitations on 20

retrieving from variables 34 returning from expressions 6,7returning from functions. See return values returning from strings 417 selecting 19 setting conditions for 255 setting dynamically 405 specifying default 21 storing 16 supplying to random number generator 365, 442 variable lists 258 variable-length arrays. See dynamic arrays variable-length strings C functions and 51 copying portions of 302 declaring 25 getting characters in 283, 285, 379, 380 getting length 285 initializing 245 reading 209, 210 replacing portions of 304, 307 trimming 386 writing 358 variables accessing 62, 400 adding type-declaration characters to 21 aligning strings in 300, 384 assigning data types to 6, 20–21, 24, 146 assigning numbers to 255, 286, 287 assigning strings to 288 assigning to classes 14, 16, 34, 100 assigning to object references 397 assigning values to 5, 15, 34, 249, 287, 414 assigning variant types to 121, 455 avoiding incorrect assignments 149, 415 avoiding name conflicts 204 avoiding type mismatch for 286 calling from external functions 46 calling from structures 34 changing values 41, 42 checking for uninitialized 272 comparing 9, 483, 485 containing empty values 274 containing null values 7, 272 copying values 62, 112 creating global 15, 146, 148, 243

variables (continued) creating Java objects and 53 creating local 16 creating static 201, 414, 424 creating user-defined 146, 448 declaring 14–16, 20, 148 caution for 149 defining dates as 31, 125 defining numbers as 30, 31 defining scope of 14, 16, 245 defining strings as 25, 29 defining variants as 23–24 describing single items 20 determining data type for 236 determining if exists 237 determining if initialized 272 evaluating boolean 177 getting available 78 getting names 240 getting number of 238 getting values in 34, 234, 417 initializing 15, 21, 148, 245 inverting bit values 479, 490 mapping column names to 400 naming 11, 414 passing by reference 41, 84 passing by value 42, 84, 145 passing to functions 201 passing to procedures 203, 425 preserving values 201, 415, 424 programming tasks for 70 providing aliases for 400 reading data into 208 retaining values of 16 reusing names for 16 setting dynamically 405 specifying indexed searches and 77 storing data in 6, 14, 18 storing return values in 40 testing for null values in 274 testing value assignments for 272 values frequently changing 149 writing to disk files 464 variant data See also Variant data type assigning to variables 23-24, 121, 455 building arrays and 17

checking for null 274 comparing 482 converting to currency 85 converting to doubles 90 converting to integers 98 converting to longs 103 converting to singles 116 formatting 27, 184 getting values 235 reading 210, 257, 358 returning first byte for 284 returning first segment of 283 returning from function calls 27, 28 returning from procedures 41 returning last byte for 380 returning last segment of 379 storing as strings 28, 286, 287 storing internally 456 writing to disk files 355, 463 Variant data type assigning values 23, 30, 288 converting to 121, 122 copying as String 302 declaring 20, 24 initializing 245 limitations 23, 24, 108 mapping C functions to 50 Variant symbolic constants 456 variant type-declaration symbol 21 Variant variables 70 VarType function 23, 456 varying payments 331 vector graphic images 427, 428, 429, 431, 433, 434 Verify function 78 version numbers 211 versions, preserving 404 view formats 241, 278 ViewerReportContext value 229 viewing class names 217, 218 Clipboard content 401 code examples 72 custom browser output 230 data 356, 439 date and time values 32, 131, 137 error messages 81, 334

headlines 223, 224, 225, 403 messages 199, 260, 318, 321 reports 241 search results 231 string values 25, 28, 220 Visual Basic 4, 18, 25 void type 50, 55 Volatile keyword 149 volatile variables 149 volume attributes 214 volume labels 215 volume names 242 *See also* Encyclopedia volumes volumes (file system) 213 volumes. *See* Encyclopedia volumes

#### W

w format character 345 warning beeps 83 wchar\_t type 50 web browsers developing content for 4, 5, 230 displaying output for 241 getting user agent for 234 returning scaling factor for 230 weekday counting 129 entering in expressions 131, 136 getting 32, 132, 154, 226, 457 weekday format characters 192 Weekday function 32, 457 While keyword 151 While...Wend statement 459 white space characters. See space characters whole numbers See also integers; numbers as enumerated values 19 getting 98, 182 specifying 30 Width statement 461 wildcard characters 174, 487 window style constants 408 Windows predefined colors 364 Windows systems calling C functions for 46, 48 changing directories for 92

checking for files on 177 copying files 174 creating directories 313 declaring string variables for 25 deleting files 279 enabling DOS shell for 407 getting date/time stamps for 176 getting default directory for 120, 146 getting file size 178 getting PATH variable for 157 getting system attributes for 214, 215 removing directories 381 renaming files 324 setting file type attributes for 399 working directories changing 92, 93 setting 93, 120 Write keyword 338 Write statement 463 write-only permissions 338

# X

XML character conversions 111 XMLCompressedDisplay formats 242 XMLDisplay formats 231, 242, 277, 278 XMLStyle formats 242 Xor operator 9, 492

## Y

y format characters 193, 345 year determining if leap 489 entering in expressions 136 estimating 88 formatting 193, 345 getting 32, 132, 465 getting difference in 129 getting month of 315 Year function 32, 465 Yes buttons 319, 320, 322 Yes keyword 187 Yes/No format 187

#### Ζ

zero values 406 zero-digit placeholder 187 zero-length strings 23, 257 zip codes 25 zooming 230