VectorScript Language Guide

VectorScript Language Guide

The VectorScript Language Guide was written and illustrated by Alexandra Duffy, Teresa Heaps, and Susan Collins.

Contributions

Thank you to Craig Hollinshead and Vladislav Stanev.

Registration and Updates

If you have not already done so, please register your copy of the Vectorworks software with Nemetschek Vectorworks at http://register.nemetschek.net.

If you would like to receive automatic notification of Vectorworks software updates, you can select to automatically check for updates on a weekly or monthly basis from the Session tab of Vectorworks preferences.

Vectorworks License Agreement

The license agreement binding the use of this software can be found in the Vectorworks ReleaseNotes directory or by clicking “License” in the About Vectorworks dialog box.

Table of Contents

1 Introduction to VectorScript 1

Some Background On VectorScript 1

2 Lexical Structures of VectorScript 5

Case Sensitivity 5

Symbols 5

Delimiters 5

Comments 6

Literals 6

Identifiers 8

Reserved Words 8

Special Symbols 9

3 Variables, Constants, Data Types, and Formats 11

Variables 11

Constants 12

VectorScript Data Types 13

Numeric and Data Formats 16

4 Arrays in VectorScript 21

Static Arrays 21

Dynamic Arrays 22

5 Structures 31

Creating Structures 31

Accessing Values in a Structure 32

6 Expressions 35

Simple Expressions 35

Complex Expressions 35

Operator Precedence 36

Operator Associativity 36

Arithmetic Operators 36
Comparison Operators 38
Logical Operators 39
Other Operators 40

7 Statements 43
Assignment Statements 43
Compound Statements 46
Procedure Statements 46
GOTO Statements 47
Repetition Statements 47
Conditional Statements 50

8 User Defined Functions 57
User-Defined Procedures 57
User-Defined Functions 59
Parameters 62
Program Blocks and Block Scope 63

9 Compiler Directives 67
{$INCLUDE} 67
{$DEBUG} 67
{$NAMES} 68
{$STRICT} 68
{$VER} 68
Conditional Compile Directives 68

Index 71

Introduction to VectorScript 1

VectorScript is a scripting language component of the Vectorworks® Fundamentals software package. It is a lightweight programming language which syntactically resembles Pascal, incorporating many of the programming constructs of that language. VectorScript is actually a "superset" of the Pascal language, extending basic Pascal capabilities with a number of APIs (application programming interfaces) which provide access to the features and functionality of the Vectorworks CAD engine.

This guide provides an introduction to using the VectorScript language. The VectorScript Function Reference, along with other sources of developer-oriented documentation related to scripting in Vectorworks, can be found online at http://developer.vectorworks.net

This guide provides a brief overview of the VectorScript language.

Some Background On VectorScript

VectorScript originated in 1988 as MiniPascal in the MiniCAD+ 1.0 release. Later versions of MiniCAD expanded the API, adding support for new technologies as they were implemented. With the advent of Vectorworks in 1998, MiniPascal became VectorScript. At the same time, Vectorworks introduced plug-ins, allowing users to create tools, menu items, and objects using the VectorScript language.

What VectorScript Can Do

VectorScript is a relatively general purpose programming language, and as such, it provides the ability to perform most common programming tasks. Tasks such as computations, storing a value, and manipulating data are all supported by standard constructs and methods within the language. VectorScript also provides extended capabilities specific to the Vectorworks product, adding new features not found in more generalized languages.

Object Creation and Editing

VectorScript allows you to create and edit objects directly within a Vectorworks file. You can create primitive objects such as lines as well as more complex objects such as multiple 3D extrudes or complex 3D solids. VectorScript also provides the ability to edit both the geometry and graphic attributes of these objects through extensive APIs built into the language.

Document Control

VectorScript provides APIs for controlling the various settings of individual Vectorworks files. These interfaces allow you to retrieve and set geometric attributes of the file such as design layer scales or visibility, along with graphical attributes such as fill or pen color.

Extended Data

VectorScript allows you to manipulate the extended data contained within the file to suit your specific needs. VectorScript APIs provide access to and control over worksheets, data records, and textures which allow you to perform "deep editing" of your files.

What VectorScript Can't Do

VectorScript has an impressive range of capabilities; however, they are mostly confined to the scope of Vectorworks and Vectorworks files. Since VectorScript is intended to be used within this context, it does not have features that would be required for a standalone language:

- VectorScript does not have the ability to work across multiple files or outside of a Vectorworks file context.
- For reasons of simplicity and stability, VectorScript does not have the ability to manage or control memory allocation.
- VectorScript does not support system level calls for file-related or other tasks.
- VectorScript does not provide external database or other connectivity options.
- Finally, VectorScript does not provide multithreading capabilities.

An Example Script

Let's take a look at a simple example to become more familiar with some of the basics of a typical script. The listing below is an example of a small script which displays a message in the VectorScript message bar, then clears the message after five seconds:

```
PROCEDURE FirstExample;
CONST
   kGREETING = 'Hello ';
VAR
   MyMessage : STRING;

BEGIN
   myMessage:='VectorScript';

   Message(kGREETING,myMessage);
   Wait(5);
   SysBeep;
   ClrMessage;
END;
Run(FirstExample);
```

The program begins with a statement which names the procedure and identifies it to the VectorScript compiler:

```
PROCEDURE FirstExample;
CONST
   kGREETING = 'Hello ';
VAR
   myMessage : STRING;

BEGIN
   myMessage:='VectorScript';
```

Identifies the script to the VectorScript compiler

```
    Message(kGREETING,myMessage);
    Wait(5);
    SysBeep;
    ClrMessage;
END;
Run(FirstExample);
```

After this statement is what is known as the main program block. The main program block contains areas for declaring what data storage will be needed by the script when it is run along with an area for the source code of the script, which provides the instructions on what actions will be performed by the script:

```
PROCEDURE FirstExample;
CONST
    kGREETING = 'Hello ';
VAR                                  Declares data storage for the script
    myMessage : STRING;

BEGIN
    myMessage:='VectorScript';
    Message(kGREETING,myMessage);
    Wait(5);
    SysBeep;                         The source code of the script
    ClrMessage;
END;
Run(FirstExample);
```

The script ends with a special statement which tells the VectorScript compiler to execute the script code preceding it:

```
PROCEDURE FirstExample;
CONST
    kGREETING = 'Hello ';
VAR
    myMessage : STRING;

BEGIN
    myMessage:='VectorScript';

    Message(kGREETING,myMessage);
```

```
    Wait(5);
    SysBeep;
    ClrMessage;
END;
Run(FirstExample);
```

Tells the VectorScript compiler to run the script

Even though some of the concepts behind the parts of the script may not be clear to you at this point, studying the example should give you an idea of what a script looks like and how it works. Later sections will explain the various parts of a script and their underlying concepts in greater detail.

Exploring VectorScript

The best way to really learn any new programming language is to write programs with it. As you read through this guide and through the online function reference (http://developer.vectorworks.net), you are encouraged to try out features as you learn about them. There are several ways to do this, which make it easy to experiment with VectorScript and learn about the language.

The most basic way to explore VectorScript is to take a Vectorworks file and export it using the Export VectorScript command. Once you have exported the file, open it in a text editor. What you will see is a VectorScript representation of the complete Vectorworks file: objects, layers, classes, document settings, and so on. You can compare this script code to the source file to see how a particular setting is created using VectorScript, or you can modify part of the script code and import it into a blank file to see how your changes affect the file. You can also use parts of this script code in your own scripts, either as-is or as a basis for your own custom work.

Another useful technique for exploring VectorScript is to make use of the Custom Tool/Attribute and Custom Selection commands of Vectorworks. These tool items make use of VectorScript to perform actions in Vectorworks, and you can use them to explore how to use VectorScript. The Custom Tool/Attribute command lets you save graphical attribute and tool settings for later use, and Custom Selection lets you define search criteria to select subsets of objects in your file. Both these techniques can be very useful when writing your own scripts, and you can see how to use these techniques by opening up the scripts and examining the script code.

Possibly the best technique is to start writing your own scripts from scratch. You can use the Resource Browser in Vectorworks to create blank scripts and edit them through the VectorScript Editor. The VectorScript Editor provides several handy features which give you quick access to API information and other basics of the language.

While exploring VectorScript, you will probably write scripts which don't execute, or don't work as you expected. To correct problems which prevent your script from executing, you can check VectorScript's Error Output file, which will indicate the source of any fatal errors in your scripts. To correct problems which are preventing your script from working as desired, you can use the VectorScript debugger to trace through your code and locate the problem. You can also use the basic technique used by many other languages—insert statements which display the values of relevant variables in your script. VectorScript provides a convenient tool for this in the `Message()` statement.

Good luck with VectorScript, and have fun!

Lexical Structures of VectorScript 2

Every programming language has a set of rules which specify how to write programs using that language. These rules are known as the lexical structure of the language. This structure is the lowest level syntax of a language, specifying things like how variables are named, what separates one program statement from the next, and so on. This section explains the basic lexical structure of VectorScript.

Case Sensitivity
Symbols
Delimiters
Comments
Literals
Identifiers
Reserved Words
Special Symbols

Case Sensitivity

VectorScript is not case sensitive. This means that items such as language keywords, variables, function names, and any other identifiers can be specified using uppercase, lowercase, or a mixed case and still be compatible with other variations of the same item. This differs from languages such as JavaScript or C.

Symbols

In VectorScript, **symbols** are the atomic, or smallest meaningful, elements of the language. VectorScript source code is comprised of a succession of these symbols, which form the instructions in the script that tell Vectorworks what actions to perform. Another term for symbols is **tokens**. Several rules govern how symbols are defined:

Each symbol is written as a series of ASCII characters, and symbols must conform to the following rules:

- Each symbol must be unbroken; symbols cannot occur inside of other symbols.
- Symbols must be comprised of 8-bit ASCII characters (or, more technically, the ISO-8859-1 character set).
- Symbols can have a wide variety of meanings and uses in VectorScript. They can, among other uses, represent data storage locations, indicate mathematical operations to be performed, or control script execution.
- Symbols are separated by other characters known as **delimiters**. Delimiters separate symbols and identify them as discrete items; symbols and delimiters must alternate.

Delimiters

Delimiters allow the VectorScript compiler to distinguish variables, statements, and other language items as separate, meaningful objects within the script. The principal delimiters in VectorScript are spaces, tabs, and the newline character. VectorScript uses these characters to separate language objects, but otherwise ignores them. Delimiters cannot be inserted within a symbol; a delimiter placed within a symbol will break it into two separate items (and will generate a syntax error).

Certain lexical constructs in VectorScript can also function as delimiters while performing other functions within the script code. For example, the VectorScript compiler can process the mathematical expression

```
circumference:=2*3.14159*radius
```

because the * character and the term := both act as delimiters in addition to the other operations they perform. These terms, known as **special symbols**, are one type of lexical construct which perform this "double duty" in VectorScript. Others include **comments** and **compiler directives**; later sections will cover these items in greater detail.

Since spaces, tabs, and new lines do not have meaning to the VectorScript compiler, you are free to use them to indent and format your script code. This type of formatting makes your scripts easy to read and understand.

Comments

Comments in VectorScript are used to place descriptive text within script code. They are most often used to document script code for your reference and for others who may work on your scripts. The VectorScript compiler ignores comments.

The general syntax for VectorScript comments is:

```
{ <your comment text> }
```

The opening and closing braces indicate the limits of the comment text. VectorScript does not support C or C++ style comments.

It is highly recommended that you comment your code when writing your scripts. Script comments eliminate the frustration of trying to remember exactly how the code works when you (or others) need to revisit and modify a script at a later date.

The alternate syntax is parenthesis asterisk:

```
(* <your comment text> *)
```

This can be used to comment out a block of the script that may already contain comments.

For example:

```
(* block comment
{Some comment line.}
{Another comment.}
*)
```

Literals

Literals in VectorScript are data values that appear directly within the script code. Literals can be numbers, text strings, the Boolean values `TRUE` and `FALSE`, or the special value `NIL`. The following subsections describe each literal type.

Integer Literals

Integer values in VectorScript are represented as a sequence of digits with an optional minus sign prepending the sequence (for negative values).

3	-255	1000000

Floating-point Literals

Floating-point values may be represented using either the traditional decimal point notation or by using exponential (scientific) notation.

A floating-point value in decimal format is represented as:

- An optional plus or minus sign, followed by
- The integral part of the value, followed by
- A decimal point and the fractional part of the number.

A floating-point value in exponential notation is represented as:

- An optional plus or minus sign, followed by
- The integral part of the value, followed by
- A decimal point and the fractional part of the number, followed by
- The letter e or E, followed by
- An optional plus or minus sign, followed by
- A one, two, or three digit integral exponent value. The preceding integral and fractional parts of the value are multiplied by the exponent.

3.1415927	6.02e23	.333333333
-3.267E-04	-0.004568	1.1414e-15

VectorScript also allows you to use dimensional notation with numeric literals and values, and will recognize common dimensional symbols for units such as feet, inches, or meters. See “Units and Numeric Values in Scripts” on page 16 for details on how to use numeric literals with dimensional notation.

String Literals

Strings literals are any sequence of zero or more characters enclosed within single quotes. They are represented using the following rules:

- Each literal must be enclosed in single quotes.
- Constants may be written on multiple lines, but return characters will be converted to spaces.
- Blanks, tabs, and carriage returns count as valid characters within literals.
- The maximum length of a string literal is 255 characters.
- A string literal with nothing between the quotes is assumed to be the null string.
- To write a single quote within a string literal, use two consecutive single quotes in the literal statement.

```
'VectorScript'          'Nemetschek Vectorworks'
'Section A-A'           'Provide approx. 3'' clearance'
```

Boolean Literals

Boolean literals in VectorScript represent a “truth value” (whether something is true or false). Most comparison operations in VectorScript yield a Boolean value that indicates whether the operation succeeded or failed. Since there are two possible truth states, there are two Boolean literals in VectorScript: the keywords `TRUE` and `FALSE`.

The NIL Literal

The last literal type in VectorScript is a specialized literal, the NIL literal. Other literals in VectorScript represent a particular type of data. The NIL literal is different—it represents a lack of value. In a sense, NIL is like zero for data types other than numbers. NIL is usually associated with the HANDLE data type, where its use indicates that no handle exists.

Identifiers

Identifiers in VectorScript are symbols which are used to refer to something else: constants, variables, data types, procedure or function names, and other similar items.

The rules for writing VectorScript identifiers are similar to most programming languages:

- The first character must be a letter or an underscore.
- Subsequent characters may be a character, digit, or underscore.
- Identifiers may not contain spaces, tabs, or other characters.
- Identifiers may be any length, but the first 255 characters are significant (i.e., recognized by the VectorScript compiler).

Identifiers which do not follow the specified rules will prevent a script from compiling, and will generate a VectorScript compiler error.

Value Identifiers

- num
- color_32bit
- totalLumberUsed
- SUM
- _dummy
- A_very_fine_identifier

Invalid Identifiers

- 52pickup
- three+two
- SUB TOTAL

Reserved Words

Reserved words are a special class of symbol in VectorScript. Reserved words are specialized symbols which have significant meaning to the VectorScript compiler—they allow the compiler to determine important information about your script and how to use that information to compile and execute your script correctly. You should avoid using reserved words as identifiers in your scripts, as they will cause errors and/or unexpected behavior.

VectorScript Keywords

The following table lists the reserved words (also known as **keywords**) in VectorScript:

- ALLOCATE
- AND
- ARRAY
- BEGIN
- BOOLEAN
- CASE
- CHAR
- CONST
- DIV
- DO
- DOWNTO
- DYNARRAY
- ELSE
- END
- FALSE
- FOR
- FUNCTION
- GOTO
- HANDLE
- IF

- INTEGER
- LABEL
- LONGINT
- MOD
- NIL
- NOT
- OF
- OR
- OTHERWISE
- PI
- PROCEDURE
- REAL
- REPEAT
- STRING
- STRUCTURE
- THEN
- TO
- TRUE
- TYPE
- UNTIL
- USES
- VAR
- VECTOR
- WHILE

Other Keywords

The following table lists reserved words which have no current meaning to the VectorScript compiler, but have been reserved for possible use in the future. You should also avoid using them in your scripts, as they may cause problems with future versions of the language.

- FILE
- FORWARD
- IMPLEMENTATION
- INHERITED
- INTERFACE
- INTRINSIC
- OBJECT
- OVERRIDE
- PACKED
- PROGRAM
- SET
- UNIT
- USES
- WITH

Since VectorScript is not case sensitive, corresponding upper and lower case versions of terms (begin and BEGIN, for example) are equivalent and should be avoided.

Special Symbols

Special symbols are another specialized class of symbol in VectorScript. Special symbols, like reserved words, have significant meaning to the VectorScript compiler. They indicate actions the compiler should take and how to control and execute your script, as well as functioning as delimiters in other script statements.

+	-	*
/	^	=
(	)	[
]	{	}
.	,	$
<>	<=	>=
:=	..	**

The table lists characters and character pairs recognized as special symbols in the VectorScript language. The specific meanings and uses of the individual special symbols will be covered in detail in later sections.

Variables, Constants, Data Types, and Formats 3

The previous section introduced the concept of literals, data values embedded directly within your VectorScript code. Scripts that operate only on such static data are rather limited and inflexible; to move beyond this limitation, VectorScript uses **constants** and **variables**. Constants and variables are names (more technically, **identifiers**) that which have associated data values; we say that the variable or constant "stores" or "contains" the value.

Constants and variables provide a way to store and manipulate values by name. In the case of constants, the value cannot be changed during script execution; in the case of variables, however, the value associated with a name may be changed at any point by assigning a new value to the name (hence the term "variable").

Another important VectorScript concept is that of **data types**. As the name implies, data types are the kinds of data that can be manipulated by your scripts. Data types provide structure and meaning to the information being manipulated by a script, allowing VectorScript to process it efficiently and safely.

This section explains how to use variables and constants in your scripts, and provides detailed information on the various data types and numeric and data formats available in VectorScript.

Variables
Constants
VectorScript Data Types
Numeric and Data Formats

Variables

Variables are created through a **variable declaration**. The variable declaration associates the variable name identifier with a specific data type. This data type tells the VectorScript compiler how much memory storage will need to be allocated for the data that will be stored in that location.

The general syntax for a variable declaration is:

```
<identifier>(,<identifier>,...) : <data type>;
```

Multiple identifiers of a single data type can be specified by a comma delimited list.

VectorScript Type Declarations

jobName:STRING; i,j,k:INTEGER;

For simple data and array types, these declarations occur in one location in the script, known as the `VAR` block. This area of the script is located at the beginning of the main program block, prior to the main body of script code, and is indicated by the `VAR` keyword. The `VAR` block is the only location where variables can be declared; unlike languages such as Basic or JavaScript, variables cannot be declared in the source code of the script.

VectorScript uses the information provided by the `VAR` block to allocate memory needed for the script to execute properly. In the example below, two variables are declared to provide data storage for the script:

```
PROCEDURE Example_1;
VAR
    s:STRING;
    i:INTEGER;
```

```
BEGIN
    s:='VectorScript';
    i:=2;
    Message('Hello ',s);
    Wait(i);
    ClrMessage;
END;
Run(Example_1);
```

Note that values are not actually assigned to the variables declared in the VAR block. The actual assignment of values into the variable storage locations occurs in the body of the script. The purpose of the VAR block is to define storage requirements, not to define data.

Constants

Constants are created using a **constant definition**. Constant definitions also associate an identifier with a storage location in memory, but unlike variable declarations, a value is immediately assigned to the location. The value of the constant cannot be modified by a script after it is defined.

The general syntax for a constant definition is:

```
<identifier> = <value>;
```

Constants, unlike variables, do not require an explicit data type.

Constant definitions also occur at one location in the script, the CONST block. This area of the script is located at the beginning of the main program block, prior to both the main body of script code and the VAR block. The block is indicated by using the CONST keyword. Like the VAR block, the CONST block is the only location where this type of storage declaration (constant definitions) is allowed.

In the following example, constants are used to define values that could be used to customize the script for a specific target, such as a particular market:

```
PROCEDURE Example_1;
CONST
    {capitalized to distinguish them from variables}
    LOCAL_GREETING_ENGLISH = 'Hello ';
    LOCAL_GREETING_FRENCH = 'Bonjour ';
VAR
    s:STRING;
    i:INTEGER;

BEGIN
    s:='VectorScript';
```

```
    i:=2;
    Message(LOCAL_GREETING_ENGLISH,s);
    Wait(i);
    ClrMessage;
END;
Run(Example_1);
```

Once the value is defined, it can be used in the script as needed. Note again that no data type is required for constants; VectorScript will implicitly convert the value to the proper type if needed.

Constants can store any basic data type (INTEGER, LONGINT, REAL, STRING, CHAR, or BOOLEAN). VectorScript also supports the use of trigonometric, ordinal, and other mathematical functions in defining constants. The following table lists functions which are supported in the constant definition block and can be used to define constants in scripts.

- Abs( )
- Sqr( )
- Sqrt( )
- Ord( )
- Chr( )
- Trunc( )
- Round( )
- Sin( )
- Cos( )
- Tan( )
- ArcSin( )
- ArcCos( )
- ArcTan( )
- Ln( )
- Exp( )

VectorScript Data Types

Any data used in a script must have an associated data type. Data types allow the VectorScript compiler to determine how much memory to allocate for storage during script execution, as well as how to act on that data when performing calculations or other operations.

A data type must be specified whenever a variable is declared. Also, whenever a procedure or function is declared, a data type must be specified for each parameter as well as the return value in the case of a function (procedures and functions are covered in greater detail in "User-Defined Functions" on page 59).

There are two categories of data types within VectorScript: **fundamental types** and **user-defined types**. Fundamental types are predefined by the compiler, while user-defined types are defined within the script code itself.

Fundamental Data Types – Numeric
Fundamental Data Types – Text
Fundamental Data Types – Other

Fundamental Data Types – Numeric

VectorScript supports three numeric data types: INTEGER, LONGINT, and REAL.

INTEGER

Values of type INTEGER are a subset of the whole numbers. INTEGER values may be in a range of -32768 to 32767, and may not contain any fractional or decimal parts. Numbers which contain fractional or decimal parts will be truncated if assigned to a variable of type INTEGER.

In VectorScript, variables of type INTEGER will only accept INTEGER values or LONGINT values which fall within the valid INTEGER range.

LONGINT

Values of type LONGINT are also a subset of the whole numbers. LONGINT values can represent a larger range of values than the INTEGER type, with the range for LONGINT values spanning from -2,147,483,648 to 2,147,483,647.

LONGINT values, like INTEGER values, may not contain any fractional or decimal parts. Numbers which contain fractional or decimal parts will be truncated if assigned to a variable of type LONGINT. In VectorScript, variables of type LONGINT will accept either LONGINT or INTEGER values.

Arithmetic operations involving values of types INTEGER and LONGINT follow these rules:

- All integer constants in the valid value range of type INTEGER are considered to be of type INTEGER. All integer constants in the range of type LONGINT, but not in the range of type INTEGER, are considered to be of type LONGINT.
- When both operands of an operator (or the single operand of a unary operator) are of type INTEGER, the result is of type INTEGER (truncated if it falls outside the range of values which can be represented by that type). Similarly, if both operands are of type LONGINT, the result is of type LONGINT.
- When one operand is of type LONGINT and the other is of type INTEGER, the INTEGER operand is converted to LONGINT and the result is of type LONGINT. If this value is assigned to a variable of type INTEGER, it is truncated.

REAL

Values of type REAL (also known as floating-point values) are a subset of the real numbers, and can store fractional or decimal parts of a number. Valid REAL values fall within a range of 1.7 x 10e-307 to 1.7 x 10e307.

In VectorScript, variables of type REAL will accept REAL, LONGINT, or INTEGER values. LONGINT and INTEGER values will be converted to the REAL data type before being assigned to a variable.

Fundamental Data Types – Text

"Literals" on page 6 described how string literals may be included in a script by enclosing them in single quotes. VectorScript also allows string values to be stored as data during script execution, and supports three data types for representing this data within scripts: STRING, CHAR, and CHAR arrays. This section will discuss the first two types; CHAR arrays will be discussed in detail in "Extended String Support with CHAR Arrays" on page 25.

STRING

STRING values are used to store and manipulate textual data within scripts. A variable of type STRING will store up to 255 characters of textual data, and STRING data values will support any valid ASCII character. Data values of type STRING are also compatible with string and character literals.

CHAR

CHAR data values store a single ASCII character, and they are a distinct type from the STRING data type. CHAR values can be used to obtain and convert single characters from STRING values, and they are often used to define special characters for use in a script.

STRING and CHAR values are compatible types, and values of these types may be assigned and compared directly.

Fundamental Data Types – Other

VectorScript also supports the following data types.

BOOLEAN

BOOLEAN data values may hold one of two values, the truth values (and reserved words) TRUE or FALSE. Values of the BOOLEAN type are more closely similar to Java or JavaScript boolean values in that they are a distinct type; unlike C or C++, they do not use numeric values to simulate TRUE or FALSE.

Boolean values are generally the result of comparison operations that occur within a script, and they are most often used for decision making during script execution.

HANDLE

HANDLE values in VectorScript are used to store a reference to other Vectorworks data in memory. Values of type HANDLE are most often used to reference data related to objects, layers, classes, or other Vectorworks internal structures. VectorScript makes extensive use of HANDLE values throughout the VectorScript API as an easy means of retrieving or setting this data directly from a script.

Aside from a reference to data located in memory, HANDLE values can also be set to the value NIL. As explained in "The NIL Literal" on page 8, the value NIL indicates no reference exists or was found.

Since HANDLE values are references to dynamic memory locations, they should not be stored or otherwise treated as if they were permanent reference to a given item within a document. Storing and reusing HANDLE values can cause errors or other unpredictable behavior within your scripts.

VECTOR

VectorScript provides the specialized VECTOR data type to support vector operations within VectorScript. Vectors are used to represent quantities which have an associated displacement, characterized by a direction and a distance (or magnitude). A VectorScript VECTOR consists of three component REAL values which can also be treated as a single unit value.

When used in conjunction with the vector API of the VectorScript language, VECTOR values can be highly useful in performing complex geometric computations in scripts. Details on this API may be found in the VectorScript Function Reference.

POINT

The POINT data type is used to store the coordinates of a 2D point. It is a compound data type consisting of two component REAL values: x and y. The value is assumed to be in the units of the current document, and relative to the document origin.

POINT3D

The POINT3D data type is used to store the coordinates of a point in 3D space. It is a compound data type consisting of three component REAL values: x, y, and z. The value is assumed to be in the units of the current document, and relative to document origin.

RGBCOLOR

The RGBCOLOR data type can store a color as three components: red, green, and blue. Each component is a LONGINT value.

Numeric and Data Formats

Units and Numeric Values in Scripts

Numeric values which are associated with unit markings follow these rules:

- VectorScript will scan for all legal predefined unit marks when parsing numeric values. If illegal characters are found after numeric values, a VectorScript warning will be generated.
- VectorScript will not scan for user-defined unit marks.
- Numeric values in VectorScript which are bound to a unit marking will be sized to be accurate to their unit mark within the current units setting of the active document. For example:

```
Rect(a,a,a + 1'2",a + 1'2");
```

  will always draw a rectangle that is 14" on a side independent of the units setting, and

```
Rect(a,a,a + 14cm,a + 14cm);
```

  will always draw a rectangle which is 14 centimeters on a side, independent of the document unit setting.

- Numeric values which are not bound to a unit marking will be sized to the current units setting of the document. For example:

```
Rect(a,a,a + 14,a + 14);
```

  will draw a rectangle 14 document units on a side. If the current units setting is Feet, the rectangle will be 14 feet on a side; if the units setting is millimeters, the rectangle drawn will be 14 mm on a side.

- Numeric constants are bound to any specified unit mark. For example:

```
CONST
    kX = 5.5cm;
```

  will be bound to and retain its centimeter unit marking.

Absolute and Relative Modes
Distance-angle Mode
Data Formatting with Write and WriteLn

Absolute and Relative Modes

The default drawing mode of VectorScript is **absolute mode**. In absolute mode, values passed as parameters for drawing or positioning objects are assumed to be actual coordinate values relating to the Vectorworks coordinate system. For example:

```
Rect(2',0',0',2');
```

will draw a rectangle with the top left corner at `(2',0')` and the bottom right corner at `(0',2')`.

In **relative mode**, values are assumed to be relative offsets from the current drawing pen position in the active document. Using the example above:

```
Rect(2',0',0',2');
```

If the pen position prior to the call was `(4',2')`, the call would draw a rectangle with its top left corner located at `(4',4')` and its bottom right corner located at `(6',2')`. Additional drawing calls while in this mode would be relative to the last function call which positioned the drawing pen.

VectorScript uses two calls, `Absolute()` and `Relative()`, to explicitly set the drawing mode of the document. These calls can be used to set the document draw mode and draw objects using offset rather than absolute values. For example:

```
Relative;
MoveTo(2",2");
Poly(1",0",0",1",-1",0",0",-1");
```

will draw a square polygon 1" on a side with the lower left corner located at `(2",2")`. The same calls made without a call to `Relative()` will draw a different polygon using absolute coordinate locations.

Once the relative mode is set, it will remain active until a call to `Absolute()` or when the script finishes execution. Be sure to reset the drawing mode to the desired state in order to ensure correct results from your script.

Distance-angle Mode

VectorScript also supports an additional numeric mode for drawing objects, distance-angle mode. With distance-angle mode, coordinate locations are defined using a distance and a direction angle, similar to polar coordinates. When specifying a distance-angle pair, the distance is specified in place of the x-coordinate, and the angle is specified in place of the y-coordinate. For example:

```
Relative;
MoveTo(2",2");
Poly(1",0",0",1",-1",0",0",-1");
```

could be specified as

```
Relative;
MoveTo(2",2");
Poly(1",#0d,1",#90d,1",#180d,1",#270d);
```

In distance-angle mode, the pound (#) sign is used to denote that an angle value follows.

VectorScript supports a wide array of formats for specifying the angle component of a distance-angle pair. The table below lists the supported angle formats.

Angle Format	Example
Integer value	`Rect(2,#90,2,#0);`
Decimal value	`Rect(2,#89.5,2,#359.5);`
Degrees	`Rect(2,#90d,2,#0d);`
Degrees-minutes-seconds	`Rect(2,#90d15'12",2,#25d30'45");`
Surveyors' Units	`Rect(20',#N45d30'00"E,15',#S45d15'2"W);`
Radians	`Rect(2,#1.57r,2,#0r);`
Gradians	`Rect(2,#100g,2,#45g);`

When using surveyors' units, be sure to use `AngleVar()` and `NoAngleVar()` to ensure that the bearing values are interpreted correctly.

Data Formatting with Write and WriteLn

Each parameter in a `Write` or `WriteLn` parameter list may be formatted for output as follows:

```
Parameter : [MinWidth] : [DecPlaces]
```

where the fields `MinWidth` and `DecPlaces` are optional.

`MinWidth` specifies the minimum overall field width, or number of characters, in the data value. Its value must be greater than or equal to zero.

Numeric Values and Formatting
String Values and Formatting
Examples of Numeric Values and Write-WriteLn
Examples of String Values and Write-WriteLn
Units and Numeric Values in Scripts

Numeric Values and Formatting

If `MinWidth` is less than the overall width of the value, Vectorworks overrides the `MinWidth` so that the entire value is displayed (see also `DecPlaces` below). If `MinWidth` is greater than the overall length of the value, blank spaces will be appended to the beginning of the value.

For `REAL` data, `DecPlaces` allows control over the display of the number of decimal places in the value. `DecPlaces` works independently of the `MinWidth` format specifier.

If `DecPlaces` for a value is set to 2, two decimal places of accuracy will always be shown, overriding the `MinWidth` specifier if necessary. If the number of decimal places in the value exceeds the number of decimal places specified, the value will be rounded. For values other than `REAL`, `DecPlaces` will generate an error.

Examples of Numeric Values and Write-WriteLn

String Values and Formatting

The `MinWidth` value acts as the length display specifier for the string, and will truncate the string if `MinWidth` is less than the length of the string value. If `MinWidth` is larger than the string length, spaces will be prepended to the value.

Examples of String Values and Write-WriteLn

Examples of Numeric Values and Write-WriteLn

INTEGER Values

In the following example, the value being formatted overrides the specified value for `MinWidth`:

```
theInt:=23456;
Write(theInt:3);
```

will write `'23456'` to the file.

When `MinWidth` exceeds the width of the formatted value, spaces are prepended the value:

```
theInt:=23456;
Write(theInt:7);
```

will write `'  23456'` to the file.

REAL Values

In the following example, a combination of `MinWidth` and `DecPlaces` values are used to format the value string. The value displays a total character length (including the decimal point) of six characters, and displays two-place decimal precision. The value is rounded to meet the specified display settings:

```
theReal:=789.128;
Write(theReal:6:2);
```

will write `'789.13'` to the file.

If the `DecPlaces` setting exceeds the precision of the value to be displayed, zeroes will be appended to bring the value up to the `DecPlaces` setting. `MinWidth` is overridden by both the value and the `DecPlaces` setting:

```
theReal:=789.128;
Write(theReal:2:6);
```

will write `'789.128000'` to the file.

Examples of String Values and Write-WriteLn

In the example, the `MinWidth` specifier is varied to display parts of the overall string value:

```
theString:='This is a sample string';
Write(theString:7);
```

will write `'This is'` to the file.

```
theString:='This is a sample string';
Write(theString:25);
```

will write `' This is a sample string'` to the file.

```
Write('VectorScript':6);
```

will write `'Vector'` to the file.

```
Write('VectorScript':16);
```

will write `'      VectorScript'` to the file.

Arrays in VectorScript 4

An **array** in VectorScript is a collection of data values referenced by a single identifier. Arrays allow large amounts of data to be stored and manipulated during script execution.

The data values contained within an array are stored in a contiguous set of memory locations, and can be accessed either randomly or in sequential order. In VectorScript, you can access this data by means of an **array index**. An array index is an `INTEGER` value corresponding to a specific storage location within the array. VectorScript arrays are indexed (that is, an individual data value is retrieved from the array) by enclosing the index value in square brackets after the array name. For example, if `my_data` is an array, and `i` is an `INTEGER` variable, then

```
my_data[i]
```

is an element of the array.

VectorScript provides support for two types of arrays: static arrays (`ARRAY`), and dynamic arrays (`DYNARRAY`). This section explains the syntax and conventions for using arrays in your scripts.

Static Arrays
Dynamic Arrays

Static Arrays

Static arrays (`ARRAY`) are declared using the same method as used for variables, except that a series of storage locations is allocated for the array values, rather than a single location typical of a variable. Static array declarations occur in the `VAR` block along with other variables.

Static arrays come in one- and two-dimensional varieties. The general syntax for one-dimensional static arrays is:

```
<identifier> : ARRAY [ m..n ] OF <data type>;
```

In the array declaration, the term `[m..n]` indicates the dimension, or size, of the array. An array declared with a dimension of `[1..10]` will allocate ten contiguous storage locations in memory. Static arrays support any valid fundamental data type, as well as the user-defined `STRUCTURE` type (see “Creating Structures” on page 31 for details).

To retrieve a value from an element of a one-dimensional array, the same bracket notation described earlier is used. The array name should appear to the left of the brackets, and a non-negative `INTEGER` value representing the array index should appear within the brackets:

```
j := values[3];
values[23] := 15.5;
total := price[i] + tax;
```

An array index may be any constant non-negative `INTEGER` value or expression which resolves to such a value.

The following example illustrates the practical use of a one-dimensional array:

```
PROCEDURE Example_41;
VAR
       s:STRING;
       i:INTEGER;
```

```
words:ARRAY[1..10] OF STRING;
BEGIN
    words[1]:='VectorScript ';
    words[2]:='is ';
    words[3]:='a ';
    words[4]:='fine ';
    words[5]:='language.';
    FOR i:=1 TO 5 DO s:=Concat(s,words[i]);
    Message(s);
END;
Run(Example_41);
```

In the example, a ten element array of `STRING` is declared, and the script code begins with assignment of values to the elements of the array. In the assignments, constants are used to represent the array indices, but a variable or other identifier which evaluates to an `INTEGER` value could have been used in their place. Such an identifier is used later in the `Concat()` function call to reference array elements.

Two-dimensional static arrays extend the syntax of a one-dimensional array by adding an additional array index to the declaration:

```
<identifier> : ARRAY [ m..n,r..s ] OF <data type>;
```

In the declaration for the two-dimensional array, the first index value defines the number of "rows" in the array, while the second index defines the number of "columns." In such a two-dimensional array, `n x s` contiguous storage locations will be allocated to hold data values (when `m` and `r` are 1).

Accessing an element in a two-dimensional array is not very different from a one-dimensional array:

```
j := values[3,5];
values[23,1] := 15.5;
total := price[i,j] + tax;
```

If we think of the two-dimensional array in terms of rows and columns, we would use two index values to indicate the row and column position of the array element to be indexed.

Dynamic Arrays

Dynamic arrays (`DYNARRAY`) in VectorScript are similar to static arrays, with the notable exception of how they are dimensioned, or sized. While static arrays are explicitly sized when they are declared in the `VAR` block of your script, the size of a dynamic array is declared during the actual execution of a script. Dynamic arrays can also be resized at any point during script execution to suit your data storage requirements. As with static arrays, dynamic arrays support any valid fundamental data type, as well as the user-defined `STRUCTURE` type (see "Creating Structures" on page 31 for details).

Dynamic arrays can also be specified as one- or two-dimensional. The general syntax for a one-dimensional dynamic array is:

```
<identifier> : DYNARRAY [] OF <data type>;
```

Note that, unlike static arrays, dynamic arrays do not include the size (dimension) of the array in the brackets. This size will be defined when your script is executed. The syntax for a two-dimensional dynamic array is very similar:

```
<identifier> : DYNARRAY [,] OF <data type>;
```

As with the one-dimensional dynamic array, note that the index dimensions are not specified in the declaration. The comma, however, is needed to indicate that the array will have two dimensions.

To dimension a dynamic array, VectorScript uses the `ALLOCATE` keyword (along with a reference to the array) to reserve sufficient space in memory for all the data values that will be stored in the array. `ALLOCATE` can be used to initially dimension the array prior to first use, or it can be used to re-dimension the array should more (or less) storage space be required. For instance, to allocate five storage locations to an array `int_values` storing `INTEGER` values, you could use the following call:

```
ALLOCATE int_values[1..5];
```

The range specified inside the brackets indicates the number of elements to be created and reserved for storage.

The following example illustrates practical use of a dynamic array within a script:

```
PROCEDURE Example_42;
VAR
i,j,numtxt : INTEGER;

h : HANDLE;
textStore: DYNARRAY[] OF STRING;

BEGIN
numtxt:=Count(((T=Text) & (SEL=TRUE)));
j:=1;

ALLOCATE textStore[1..numtxt];

h:=FSActLayer;

WHILE (h <> NIL) DO BEGIN
IF (GetType(h) = 10) THEN BEGIN
textStore[j]:=GetText(h);
j:=j+1;
END;

h:=NextSObj(h);
END;
```

```
ALLOCATE textStore[1..numtxt+2];

TextOrigin(2,2);
CreateText('New text 1');
numtxt:=numtxt+1;
textStore[numtxt]:=GetText(LNewObj);

TextOrigin(2,4);
CreateText('New text 2');
numtxt:=numtxt+1;
textStore[numtxt]:=GetText(LNewObj);
FOR i:=1 TO numtxt DO BEGIN
Message('Array element ',i,' contains ', textStore[i]);
Wait(1);
END;

END;
Run(Example_42);
```

In the example, a dynamic array is used to store the text of any selected strings that may be found in the selection set. The script begins by declaring the dynamic array, `textStore`, along with several other variables. In the `VAR` block declaration, the dynamic array is specified, but no space is allocated at this point for storage.

The body of the script begins with storing the number of selected text objects found within the selection set in the variable `numtxt`. This value is then used with the `ALLOCATE` keyword:

```
ALLOCATE textStore[1..numtxt];
```

to initialize the amount of storage space in the dynamic array.

Next, the script processes the selected items, and when it encounters a text object, stores the text in an element of the array. Since the text objects within the selection set were counted, `textStore` is sized to provide sufficient storage within the array for the exact number of text strings that were found.

Once all the objects have been processed, the array can be redimensioned to allocate more or less space as needed. In the example, additional storage space is reserved with another call to `ALLOCATE`,

```
ALLOCATE textStore[1..numtxt+2];
```

and use the newly added storage locations to store the text created by the script. The script concludes by displaying the values currently stored within the `textStore` array.

Note that the existing data values stored in the array are preserved when the array is re-dimensioned. If an array is redimensioned to a larger size during execution of the script, VectorScript will preserve all the values currently in the array. VectorScript will also attempt to preserve as many data values as possible if an array is redimensioned to a smaller size. In the case of dimensioning to a smaller size, any values contained in locations beyond the newly defined boundaries of the array will be lost.

Performance Considerations with Dynamic Arrays
Vectors and Array Notation
Extended String Support with CHAR Arrays
Performing Standard STRING-Related Operations

Performance Considerations with Dynamic Arrays

Dynamic arrays require more "overhead" than comparable static arrays in order to allocate memory during script execution and to maintain array values. As a result, scripts using dynamic arrays may execute more slowly than scripts using static arrays.

It is highly recommended that you use static arrays wherever possible for the best possible script performance. If dynamic arrays are required in your scripts, avoid making frequent calls to `ALLOCATE` to reserve storage. Use `ALLOCATE` only when absolutely necessary to change reserved storage during script execution, and avoid any use of `ALLOCATE` inside of a loop or repetition statement (see "Repetition Statements" on page 47 for details on these statement types).

Vectors and Array Notation

As mentioned earlier, you can create arrays of any fundamental data type, which includes the `VECTOR` type. Vectors support two methods of accessing the fields of the vector: array-style brackets and dot notation.

To access a vector field using array-style notation, you can append an additional set of brackets to the array reference, and specify the vectors' field index within the second set of brackets. For example,

```
vec_field[5][2];
```

will access the second field (the `y`-component) of the vector in element 5 of the one-dimensional array `vec_field`. Two-dimensional arrays can also use this notation; if `vec_field2` is a two-dimensional array, then

```
vec_field2[4,5][2];
```

will access the second field of the vector located in the fourth row and fifth column of the array.

To access a vector field using dot notation, simply append the dot (field access) operator and field identifier to the array reference you want to index. Using the previous example,

```
vec_field[5].y;
```

will perform the same operation, accessing the second field (the `y`-component) of the vector in element 5 of `vec_field`. Two dimensional arrays work in a similar fashion; the reference

```
vec_field2[4,5].y;
```

will access the y-component of the vector located in the fourth row and fifth column of `vec_field2`.

Extended String Support with CHAR Arrays

VectorScript also supports a specialized set of functionality when using arrays of the `CHAR` data type. This functionality with static or dynamic arrays of the `CHAR` type provides you with a means of handling extended strings up to 32,767 characters long within your scripts.

Arrays of type `CHAR` can be used in place of the `STRING` data type in certain operations within VectorScript. The following sections provide details on operations supporting `CHAR` arrays and `STRING`s in VectorScript.

Assignments Between STRING Values and CHAR Arrays

Retrieving or Assigning Strings to Text Objects
Retrieving or Assigning Strings to Record Fields

Assignments Between STRING Values and CHAR Arrays

Both static CHAR arrays (ARRAY OF CHAR) and dynamic CHAR arrays (DYNARRAY OF CHAR) can be used in place of STRING values when assigning to or retrieving from a STRING variable.

When using either static or dynamic CHAR arrays to assign a value to a STRING variable, if the array length exceeds 255 characters, the first 255 characters will be copied into the string variable, and the remaining characters in the array will be dropped. Values of less than 255 characters will be completely copied into the STRING variable.

Assigning values from a STRING variable or constant to a static CHAR array works in a similar fashion. If the CHAR array has a length less than the length of the STRING value to be assigned, the value will be truncated to fit the array. For instance:

```
PROCEDURE Example_43;
VAR
   Part_name: STRING;
   NameArray: ARRAY[1..16] OF CHAR;

BEGIN
part_name:= 'Acme Left-handed Smoke Shifter';

NameArray:=part_name;
END;
Run(Example_43);
```

In the example, the STRING value assigned to the variable part_name would be truncated to

```
Acme Left-handed
```

when assigned to the array. When using static CHAR arrays to handle STRING values, be sure to declare the size of the array to accommodate the longest STRING value expected to be stored within the array.

In contrast to static CHAR arrays, dynamic CHAR arrays will automatically size to the length of the STRING value being assigned to the array. For example:

```
PROCEDURE Example_44;
VAR
   sampleString: STRING;
   mytext: DYNARRAY[] OF CHAR;
BEGIN
sampleString:= 'VectorScript now handles lots of text';
mytext:= sampleString;
```

```
END;
Run(Example_44);
```

If the array `mytext` was declared but not previously used, the assignment would size the array length to 36, and the array would contain the string

```
VectorScript now handles lots of text
```

If `mytext` had been previously assigned a value, the assignment would resize the array to a length of 36 and assign the `STRING` value to the array. The values previously held in the array would be lost.

Retrieving or Assigning Strings to Text Objects

VectorScript allows `STRING` values greater than 255 characters long in text objects to be set or retrieved via `CHAR` arrays. The VectorScript API functions `GetText()` and `SetText()` support the use of a `CHAR` array in place of a `STRING` value.

To set or retrieve the text string, use the name of the `CHAR` array (without brackets) in place of the `STRING` parameter or variable. For example:

```
PROCEDURE Example_45;
VAR
    h : HANDLE;
    theText : STRING;
textArray : DYNARRAY[] OF CHAR;
BEGIN
    h:=FSActLayer;
    theText:= GetText(h);
CreateText(theText);
END;
Run(Example_45);
```

In the example, you would be limited to returning the first 255 characters of the text string. By using a dynamic array:

```
PROCEDURE Example_45;
VAR
    h : HANDLE;
    theText : STRING;
    textArray : DYNARRAY[] OF CHAR;
BEGIN
    h:=FSActLayer;
    textArray:= GetText(h);
    CreateText(textArray);
```

```
END;
Run(Example_45);
```

You can retrieve the entire text string and store it in the dynamic array. The entire text string can then be used in other operations.

Retrieving or Assigning Strings to Record Fields

VectorScript also allows you to set and retrieve `STRING` values greater than 255 characters long contained in record fields via the use of `CHAR` arrays. The VectorScript API functions `GetRField()` and `SetRField()` support the use of a `CHAR` array in place of a `STRING` value.

To set or retrieve the record field string, use the name of the `CHAR` array (without brackets) in place of the `STRING` parameter or variable. For example:

```
PROCEDURE Example_46;
VAR
   theText : STRING;
   longtext : ARRAY[1..512] OF CHAR;
BEGIN
   theText:= GetRField(FSActLayer,'Boring Info','Boring Notes');
   CreateText(theText);
END;
Run(Example_46);
```

In the example, using a `STRING` variable would be limited to retrieving only the first 255 characters of the text string stored within the field. By using a `CHAR` array:

```
PROCEDURE Example_46;
VAR
   theText : STRING;
   longtext : ARRAY[1..512] OF CHAR;
BEGIN
longtext:= GetRField(FSActLayer,'Boring Info','Boring Notes');
   CreateText(textArray);
END;
Run(Example_46);
```

In the example, up to 512 characters of text from the field can be retrieved and stored in the array. Alternately, the dynamic array could be sized to support whatever amount of text might be found in the record field (up to 32K of text).

Performing Standard STRING-Related Operations

VectorScript also provides support in its string API for handling the extended strings in CHAR arrays. Operations such as obtaining string length, substring position, and string concatenation can be performed on CHAR arrays just as they can on STRING values.

To use CHAR arrays with string API functions, just use the CHAR array in place of a STRING variable for a given function parameter or return value. For example:

```
PROCEDURE Example_47;
VAR
   s : STRING;
   textArray :ARRAY [1..32] OF CHAR;
BEGIN
   textArray:= 'A VectorScript text string';
s:= Copy(textArray,3,12);
END;
Run(Example_47);
```

In the example, a CHAR array is used in place of a STRING as the source value for the Copy() function. The result of the Copy() operation is then assigned to a STRING variable. String API function calls support both static and dynamic CHAR arrays.

The following VectorScript API functions have CHAR array support.

- Len()
- Pos()
- Concat()
- Copy()
- Delete()
- Insert()
- UprString()
- GetText()
- SetText()
- GetRField()
- SetRField()
- CreateText()

Structures 5

A **structure** in VectorScript is a collection of one or more variables which are grouped together under a single identifier for convenient handling. Structures help to organize complex data into groupings that may be treated as a single "unit" instead of separate entities.

> The standard Pascal term for this type of construct is record. To avoid possible conflicts and confusion with other Vectorworks or VectorScript features, VectorScript refers to this construct as a structure.

The variables contained within a structure are known as the members of the structure. These variables may be of any fundamental type found in VectorScript. Static and CHAR arrays are also supported as structure members, as are other structures (which are known as nested structures). Dynamic arrays are not supported in structures.

Creating Structures
Accessing Values in a Structure

Creating Structures

Structures are declared in a special section of your scripts, the TYPE block. This optional section, which is located between the CONST and VAR sections of the main program block, is the only location where structures may be declared. There is no limit to the number of structures that may be declared in a TYPE block.

The general syntax for a structure declaration is:

```
<structure name> = STRUCTURE
    <identifier>[,<identifier>,…] : <data type>;
    <identifier>[,<identifier>,…] : <data type>;
    …
    …
    <identifier>[,<identifier>,…] : <data type>;
END;
```

The declaration begins with the identifier used to refer to the structure. Following this identifier is the special symbol = and the keyword STRUCTURE, which indicates that the member declarations which follow should be grouped under the specified identifier name. The members of a structure are declared just as you would declare any other variable, with all the same rules for declaring variables applying to the member declarations. The structure declaration is terminated by using the END keyword.

Structure declarations, unlike variables or constants, do not reserve storage space for data. Instead, they define a new data type which can be used in your scripts as you would any of the fundamental data types. Such a user-defined type can be used to declare variables or arrays in the same manner as using INTEGER, STRING, or other fundamental types.

For example, suppose you wish to define a structure which represents a 2D point. The structure which represents the point can be defined as shown below:

```
Point = STRUCTURE
    x,y : REAL;
END;
```

The structure POINT contains two members of type REAL, but no space is allocated until variables or arrays are declared using the structure as a user-defined type:

```
PROCEDURE StructExample1;
TYPE
   POINT = STRUCTURE
x,y : REAL;
END;
VAR
   centerPt, target : POINT;
   vertex_list : ARRAY[1..20] OF POINT;
BEGIN
END;
Run(StructExample1);
```

The centerPt and target variables each contain storage for the two REAL values contained within the structure, and the vertex_list array reserves sufficient memory to store twenty POINT items, or forty REAL values. The POINT structure acts as a "template" to use when defining data value storage for your script.

Accessing Values in a Structure

Members within a structure may be referred to directly using the . (structure member) operator. This operator is used in conjunction with the structure name and the member name you intend to reference in the form:

```
<structure name>.<member name>
```

This format, also known as "dot notation," gives you direct access to the value within the specified member. This type of structure member reference can be used in place of any simple variable to retrieve or assign values:

```
centerPt.x:= 0;
total:= windowData.cost + tax;
```

This notation can also be used when comparing values or when passing values to VectorScript or user-defined functions:

```
partData.location:= GetLName(ActLayer);
GetObject(partData.name);
```

Arrays of structures also support the use of dot notation to reference individual structure members:

```
vertices[5].x:= 2.67;
vertices[6].y:= vertices[5].y + 2.6;
```

The reference to a member of a structure in an array element is created by appending the member operator and the member name to an array element reference.

As mentioned before, structures support the use of static arrays as data members. Arrays within structures present a bit more of a syntactical challenge when referencing a member value, but otherwise they are not difficult to use. To reference a value in an array element within a structure, append the member operator and a member array element reference to the structure instance identifier:

```
p.name[5]:= 'Marvin';
total:= total + winAssembly1.cost[k];
```

As with non-member arrays, any expression or constant which resolves to an `INTEGER` value can be used when indexing the member array element.

It is also possible to have arrays of structures which have arrays as members. Once again, a combination of the member operator with a reference to the desired array element is used to obtain the data value. In this case, array element references will appear on *both* sides of the member operator. This can lead to some rather interesting looking syntax within a script:

```
doorAssembly[3].cost[4]:= 24.55;
subtotal:=subtotal+doorAssembly[i].cost[j]+doorAssembly[i].cost[j+1];
```

These expressions are perfectly valid; however, they do require extra attention to ensure the correct syntax is specified.

Structures containing other structures as members also present an additional layer of complexity when referencing members of the nested structure. The key in this situation is to use member chaining to descend through the data hierarchy to the desired value. For example:

```
PROCEDURE Example_51;
TYPE
   POINT = STRUCTURE
      x,y : REAL;
END;
CIRCLE = STRUCTURE
   ctr : POINT;
   radius : REAL;
END;
VAR
   c1,c2 : CIRCLE;
BEGIN
c1.ctr.x:= 4.5;
c2.ctr.y:= c1.ctr.y;
END;
Run(Example_51);
```

The CIRCLE structure declaration makes use of an instance of the POINT structure to more logically organize data. To reference either the x- or y-component of the POINT instance, chain the members of the nested structures:

```
c1.ctr.x:= 4.5;
c2.ctr.y:= c1.ctr.y;
```

References to the member ctr and its members x and y are chained together using the member operator to reference and access the values in the nested structure. Chaining of members in nested structures can be used repeatedly in scripts to access structure members which may be nested several levels deep.

Expressions 6

Every value in VectorScript is designated by way of an **expression**. An expression is a "phrase" in VectorScript that can be evaluated to produce a value. Expressions can be simple, consisting of a single component expressing the value, or complex, expressing the value through a combination of other expressions and operations on them.

Simple Expressions
Complex Expressions
Operator Precedence
Operator Associativity
Arithmetic Operators
Comparison Operators
Logical Operators
Other Operators

Simple Expressions

Simple expressions use a single component, or **operand**, to express a value. Simple expressions in VectorScript are most often constants (such as string or numeric literals), variable names, or function names.

The value of a simple constant expression is essentially the constant itself. The value of a simple variable expression is the value that is associated with the variable identifier. The value of a function expression is the value returned when the function has completed execution.

Expression	Description
`1.7`	Numeric literal
`'This is VectorScript'`	String literal
`TRUE`	Boolean literal
`NIL`	The value NIL
`i`	The variable "i"
`sum`	The variable "sum"

Complex Expressions

Complex expressions, also known as compound expressions, derive their values from combining or transforming the values of other expressions. For example, the value of expression

```
i + 1.7;
```

is derived from the combining values of `1.7` and `i`. Since we know that both `1.7` and `i` are also simple expressions which each have their own value, they can be combined to obtain a value.

In the expression above, the resulting value is determined by adding the values of the two simpler expressions. The expression uses an operator, in this case the plus sign, to perform an operation (addition) on the simpler expressions and to combine them into a more complex expression.

The expressions combined by the plus sign in the example above can also be referred to as operands. Operators are usually grouped by the number of operands that they require in order to perform their intended operations.

VectorScript supports two types of operators, unary operators, which require a single operand, and binary operators, which require two operands.

Each operator produces a resulting value whose data type is determined both by the operator and the operands from which the value was derived. Operators may have restrictions on the types of operands with which they are compatible, and all these factors impact the data type of the resulting value.

Operator Precedence

Just as it does in mathematics, **operator precedence** in VectorScript controls the order in which operations are performed. Operators having a higher precedence have their operations performed before those having a lower precedence. In the expression

```
p = q + r * s;
```

the multiplication operator (*) has higher precedence than the addition operator, so the multiplication operation is performed before the addition. The assignment operator (=) has the lowest precedence of all the operators, so the association, or assignment, of the value to the variable p occurs only after the other operations are completed.

Operator precedence can be overridden by the explicit use of parentheses. To force the addition operation to be performed first in the prior example, parentheses would be used to modify the expression to be:

```
p = (q + r) * s;
```

In everyday use, it is good practice to use parentheses if you are unsure about precedence in order to make the evaluation order explicit.

Operator Associativity

Operator associativity specifies the order in which operations of the same precedence are performed. Left-to-right associativity means that operations are performed left to right when operators are of equal precedence. For example, the expression

```
p = q + r + s;
```

is equivalent to the expression

```
p = ((q + r) + s);
```

because the addition operator has left-to-right associativity. Conversely, the expression

```
w = x = y = z;
```

is equivalent to the expression

```
w = ( x = (y = z));
```

because the assignment operator has right-to-left associativity.

Arithmetic Operators

Arithmetic operators perform such familiar mathematical operations as addition or multiplication on the specified operands. Arithmetic operators are restricted to working on numeric VectorScript data types. The table below summarizes the arithmetic operators available in VectorScript.

Operator	Operand	Precedence	Associativity	Operation
-	any number	1	R-L	Unary minus (negation)
^	any number	2	L-R	Exponentiation
*	any number	2	L-R	Multiplication
/	any number	2	L-R	Division
DIV	INTEGER,LONGINT	2	L-R	Integer Division
MOD	INTEGER,LONGINT	2	L-R	Modulo (remainder division)
+	any number	3	L-R	Addition
-	any number	3	L-R	Subtraction

Unary negation (-)

When - is used as a unary operator preceding a single operand, it performs a negation operation on the operand. That is, it converts a positive value to an equivalently negative value, or it converts a negative value to its equivalently positive value.

Addition (+)

The + operator adds two numeric operands. This operator is limited to addition only; unlike in many other languages, this operator may NOT be used to concatenate strings.

Subtraction (-)

The - operator subtracts the second operand from the first. Both operands must be numeric.

Multiplication (*)

The * operator multiplies its two numeric operands.

Division (/)

The / operator divides the first operand by its second operand. The operator performs floating-point division, always returning a value of type `REAL` even when both operands are of `INTEGER` or `LONGINT` type.

Integer Division (DIV)

The `DIV` operator divides the first operand by its second operand, always returning a result of type `INTEGER` or `LONGINT`. The value of `i DIV j` is the mathematical quotient of `i/j`, rounded down to the nearest `INTEGER` or `LONGINT` value. For example, the operation

```
j:= 36 DIV 5;
```

will return a result of 7, which is assigned to the variable `j`.

Remainder Division (MOD)

The MOD operator divides the first operand by the second and returns the remainder of the operation as a result of type INTEGER. For example, the operation

```
k:= 36 MOD 5;
```

will return a value of 1, which is assigned to k.

Exponentiation (^)

The ^ operator raises the first operand to the power indicated by the second operand; that is, x^y is equivalent to x to the y^{th} power.

Comparison Operators

Comparison operators in VectorScript are used to compare values of various types and return a Boolean value (true or false) result. The results of expressions using comparison operators are most often used to control the flow of script execution.

The table below summarizes the comparison operators available in VectorScript.

Operator	Operand	Precedence	Associativity	Operation
<	Number, STRING, CHAR	4	L-R	Less than
<=	Number, STRING, CHAR	4	L-R	Less than or equal to
>	Number, STRING, CHAR	4	L-R	Greater than
>=	Number, STRING, CHAR	4	L-R	Greater than or equal to
=	Any type	5	L-R	Equal to
<>	Any type	5	L-R	Not equal to

Less Than (<)

The < operator evaluates as TRUE if the first operand is less than the second operand; otherwise it will evaluate as FALSE. Operands may be numbers, strings, or characters; strings are evaluated alphabetically, by character encoding.

Less Than or Equal To (<=)

The <= operator evaluates as TRUE if the first operand is less than or equal to the second operand; otherwise it will evaluate as FALSE. Operands may be numbers, strings, or characters; strings are evaluated alphabetically, by character encoding.

Greater Than (>)

The > operator evaluates as TRUE if the first operand is greater than the second operand; otherwise it will evaluate as FALSE. Operands may be numbers, strings, or characters; strings are evaluated alphabetically, by character encoding.

Greater Than or Equal To (>=)

The >= operator evaluates as TRUE if the first operand is greater than or equal to the second operand; otherwise it will evaluate as FALSE. Operands may be numbers, strings, or characters; strings are evaluated alphabetically, by character encoding.

Equality (=)

The = operator returns TRUE if its two operands are exactly equal, and returns FALSE if they are not equal. The operands may be of any type. For operands of type STRING, the values are compared on a character-by-character basis, and must contain exactly the same characters.

Inequality (<>)

The <> operator tests for the exact opposite of the = operator. If two equal values are compared using the inequality operator, the resulting value will be FALSE. Comparison of two values which are not equal will yield a TRUE result.

Logical Operators

Logical operators perform the rough equivalent of a comparison operation on Boolean values. Logical operators use Boolean algebra to evaluate their operands and return the result of the operation. In programming, they are most often used to express complex comparisons which involve multiple operands by linking smaller expressions together.

The following table summarizes the comparison operators available in VectorScript.

Operator	Operand	Precedence	Associativity	Operation
NOT	BOOLEAN	1	R-L	Logical NOT
AND	BOOLEAN	7	L-R	Logical AND
&	BOOLEAN	7	L-R	Logical AND (short-circuit)
OR	BOOLEAN	8	L-R	Logical OR
|	BOOLEAN	8	L-R	Logical OR (short-circuit)

Logical NOT (NOT)

The unary NOT operator is used preceding a single operand of BOOLEAN type to inverts the value of the operand. For example, if a variable z of BOOLEAN type contains the value TRUE, then the expression NOT z will return a value of FALSE. This operation also holds for the results of more complex expressions; for example, if the result of the expression p>=q evaluates to FALSE, the expression NOT(p>=q) will evaluate to TRUE.

Logical AND (AND)

The AND operator evaluates to TRUE if and only if the first operand and the second operand both are TRUE. If either operand evaluates to FALSE, the result returned will be FALSE. Expressions using the AND operator will always evaluate both operands before returning the result of the expression, regardless of the value of the first operand.

Logical short-circuit AND (&)

The & operator evaluates to TRUE if and only if the first operand and the second operand are both TRUE. If either operand evaluates to FALSE, the result returned will be FALSE. Expressions using the & operator will not evaluate the second operand if the first operand returns a value of FALSE. If the second operand should have any side effects (such as those produced by a function call returning value) they may not occur. In general, it best to avoid expressions such as the following which combine side effects with the & operator:

```
(a = b) & SetVectorFill(h,'Stone'){ function call may not occur }
```

Logical OR (OR)

The OR operator evaluates to TRUE if the first operand or the second operand are TRUE. Both operands must evaluate to FALSE for the result returned to be FALSE. Expressions using the OR operator will always evaluate both operands before returning the result of the expression, regardless of the value of the first operand.

Logical Short-circuit OR (|)

The OR operator evaluates to TRUE if the first operand or the second operand are TRUE. Both operands must evaluate to FALSE for the result returned to be FALSE. Expressions using the | operator will not evaluate the second operand if the first operand returns a value of TRUE. If the second operand should have any side effects (such as those produced by a function call returning value) they may not occur. In general, it is best to avoid expressions such as the following which combine side effects with the | operator:

```
(a = b) | SetVectorFill(h,'Stone') { function call may not occur }
```

Other Operators

Assignment Operator

As described in “Variables” on page 11, variables are associated with (assigned) a value. This value can also be modified at any point during execution of your scripts. Both these operations are performed using the assignment operator.

The := operator expects the first (left-hand) operand to be a variable, array, element, or vector field/structure member. The second (right-hand) operand can be an arbitrary value of any type, though the value must be compatible with the data type of the first operand. The value of the expression is the value of the right-hand operand.

The assignment operator has right-to-left associativity, which means that the second operand is evaluated first in the expression (and is how VectorScript determines if the value and the variable are of compatible types).

Array Access Operator

As mentioned in “Arrays in VectorScript” on page 21, array elements are accessed using square brackets [ ], along with the positional index of the value to be retrieved. This bracket pair is treated as an operator in VectorScript.

The [ ] operator uses as the name of an array as its first operand (to the left of the brackets). The second operand, which goes between the brackets, can be any expression which evaluates to an INTEGER value.

If the array specified as the first operand is two-dimensional, the array access operator requires a third operand, which also goes between the brackets. In this case, both the second and third operands (which are separated by a comma) may be any expression evaluating to an INTEGER value.

For example, the expression

```
price[3]
```

will evaluate to the value in the third element of the `price` array. For a two dimensional array `plant_data`, the expression

```
plant_data[2,i+4]
```

will evaluate to the value contained in the element specified by `[2,i+4]`. The expression `i+4` must evaluate to an `INTEGER` value in order to be used as an operand in the expression.

Vector / Structure Member Access Operator

The `.` operator in VectorScript is a specialized operator that allows you to directly access values contained within certain data types, notably vectors and structures.

The . operator requires a vector or structure as its first (left) operand. The second operand, unlike most operators, must be either a vector field or structure member name; no expressions are allowed. Vector field identifiers must be one of the three valid vector field names— x, y, or z. Structure member names should correspond to a valid member in the structure type declaration.

For example, the expression:

```
distance_vector1.x
```

will evaluate to the value in the `x` field of the vector `distance_vector1`. When dealing with a structure, the expression

```
window_data.cost
```

will evaluate to the value within the member `cost` of the structure instance

`window_data`.

Statements 7

Statements in VectorScript are the actions of the language. Whereas expressions in VectorScript can be thought of as "phrases" that can be evaluated to a value, expressions don't "do" anything. To make something happen, you need to use a VectorScript statement, which is akin to a complete sentence or a command. Statements in VectorScript perform the execution tasks of your script, managing your script data and controlling the flow of script execution.

Statements in VectorScript are always found in "blocks," and a script is simply a large block containing a collection of statements. Each statement in VectorScript is terminated with a semi-colon, which indicates to the VectorScript compiler where each statement ends.

This section describes the various statement types found in VectorScript and explains their syntax in detail.

Assignment Statements
Compound Statements
Procedure Statements
GOTO Statements
Repetition Statements
Conditional Statements

Assignment Statements

Assignment statements set the value of a variable or like identifier in a script. Assignment statements use the assignment operator (:=) to set the value of the identifier on the left-hand side of the symbol to the value of the constant or identifier on the right-hand side of the symbol. This may also be thought of as assigning the value of the identifier on the right-hand side to the identifier on the left.

The generalized syntax for assignment statements is:

```
<identifier> := <identifier or constant value>;
```

The identifier on the left-hand side may be any VectorScript data type; it may also be an array element, a full array reference, or a structure field.

For example:

```
PROCEDURE Example_71;
CONST
    kInitialValue = 0;
TYPE
    POINT = STRUCTURE
        x,y:REAL;
    END;
VAR
    s : STRING;
    i : INTEGER;
    h : HANDLE;
    textdata : ARRAY[1..100] OF STRING;
```

```
   p1,p2 : POINT;
BEGIN
{ assignment of constant value to a variable }
   i:= kInitialValue;
{ assignment of return value to variable }
   h:= FSObject(ActLayer);
{ assignment of return value to variable }
   s:= GetText(h);
{ assignment of variable value to array element }
   textdata[1]:= s;
{ assignment of values to structure members }
   p1.x:= 0; p1.y:= 2;
{ assignment of member value to another member }
   p2.x:= p1.y;
{ assignment of member value to another member }
   p1.y:= p2.x;
END;
Run(Example_71);
```

From the example, it is evident that the assignment statement is very flexible. The example makes use of constants, variables, structure fields, and function return values when assigning values to an identifier. Note also that more than one statement can reside on a single line, as long as they are separated by a semi-colon indicating the end of each statement.

While assignment statements are very flexible in how they get or assign values, they do observe some rules regarding compatibility of data types. When writing assignment statements, the following rules should be observed:

- A variable of `REAL` type may be set to a `REAL`, `INTEGER`, or `LONGINT` value, as well as any expression yielding those results.
- A `LONGINT` variable may be set to a `LONGINT` or `INTEGER` value or any expression yielding such a value. It may also be set to a `REAL` value, but the value will be truncated and rounded to the nearest whole value.
- An `INTEGER` variable may be set to an `INTEGER` value, or any expression yielding such a value. It may also be set to a `REAL` value, but the value will be truncated and rounded to the nearest whole value.
- A `BOOLEAN` variable may be set a `BOOLEAN` value or an expression yielding such a value.
- A `STRING` variable may be set to a `STRING` or `CHAR` value or any expression yielding those values. It may also be set to an `ARRAY` or `DYNARRAY OF CHAR` value; however, the value in the array will be truncated to 255 characters.
- A `CHAR` variable may be set to a `CHAR` value or any expression yielding a `CHAR` value. It may also be set to a `STRING` value, but will be truncated if the `STRING` is greater than 1 character in length.
- A `HANDLE` variable may be set to a `HANDLE` value or any expression yielding a `HANDLE` value.

Assignment statements also support block copying of values in arrays when they are used without an array element index in a script. This method facilitates transferring large amounts of data without the need for copying on an element-by-element basis. For example:

```
PROCEDURE Example_72;
VAR
   values1,values2:ARRAY[1..5] OF INTEGER;
BEGIN
   values1[1]:= 2;
   values1[2]:= 4;
   values1[3]:= 8;
   values1[4]:= 16;
   values1[5]:= 32;
END;
Run(Example_72);
```

In order to transfer the values in `values1` to `values2`, it would appear that multiple assignment statements are needed, one for each array element. For large arrays, this would be a time-consuming task. Fortunately, VectorScript overloads (extends the functionality of) the assignment operator so that operation to copy the values becomes a single statement:

```
PROCEDURE Example_72;
VAR
   values1,values2:ARRAY[1..5] OF INTEGER;
BEGIN
   values1[1]:= 2;
   values1[2]:= 4;
   values1[3]:= 8;
   values1[4]:= 16;
   values1[5]:= 32;
values2:= values1;
END;
Run(Example_72);
```

The assignment statement copies the data from the `values1` array directly into the corresponding elements of the `values2` array. This sort of assignment operation can be also be performed with dynamic arrays; in both cases, however, the dimensions of the arrays on both sides of the assignment operator must be exactly the same in order to complete the operation.

Vectors and structures may also be copied in this manner; the member values of the item on the right side of the assignment operator will be copied into the corresponding member fields of the item on the left side of the operator. For example, the values in a vector `direction_vector1` could be copied into another vector:

```
new_vector:= direction_vector1;
```

The values in the fields of `direction_vector1` would be copied into the fields of `new_vector` without the need for assignment statements for each field.

Compound Statements

VectorScript provides compound statements as a way to execute several statements as if they were a single statement. This capability is quite useful when it is necessary to combine statements and execute them together—for instance, when being executed as a branch of a control statement or in a loop.

To create a compound statement from a sequence of statements, preface the first statement in the sequence with the `BEGIN` keyword. The sequence is terminated with the `END` keyword, and each statement in the sequence is separated by a semi-colon. For example:

```
BEGIN
    i:=1;
    j:= (3*2)+5;
    Message(i+j);
END;
```

The three statements contained within the `BEGIN` and `END` keywords will be executed together when the compound statement is called.

The generalized syntax for compound statements is:

```
BEGIN
<statement>; [<statement>; <statement>;...]
END;
```

Compound statements may also be nested; the VectorScript compiler will associate the last `BEGIN` keyword with the next `END` keyword in the script, the second-last `BEGIN` with the following `END`, and so on. Mismatched `BEGIN-END` pairs will cause a VectorScript error to occur.

If you noticed that the body of a script looks suspiciously similar to a compound statement, you would be correct; the script body of any VectorScript script, user-defined procedure, or user-defined function is in fact a single compound statement.

Procedure Statements

Procedure statements in VectorScript call predefined VectorScript API function calls as well as user-defined procedures and functions to perform actions within a script. With VectorScript API function calls, the actions are performed directly by Vectorworks; user-defined function calls encapsulate other VectorScript source code; which is executed when the procedure statement is called in a script.

The general syntax for procedure statements is:

```
<procedure identifier>[(<parameter list>)][:<return value>];
```

Function calls such as:

```
Message('Hello VectorScript');
```

or

```
SetSelect(h);
```

are examples of procedure statements in VectorScript. For more details on user-defined procedures and functions, see "User-Defined Procedures" on page 57 and "User-Defined Functions" on page 59.

GOTO Statements

GOTO statements transfer execution of the script to the beginning of the statement following the label associated with the GOTO. For example:

```
PROCEDURE Example_73;
LABEL 100;
VAR
   i,j : INTEGER;
BEGIN
   i:= 10;
   j:= 2;
   IF (j MOD 2 = 0) THEN GOTO 100;
   i:= i * 5;
   100: i:= i + 1;
   Message(i);
END;
Run(Example_73);
```

If the condition `(j MOD 2) = 0` evaluates to TRUE, execution in the script is transferred immediately to the beginning of the statement `i:= i + 1`, and the expression `i:= i * 5` is never executed.

The general syntax for a GOTO statement is:

```
GOTO <label>;
```

GOTO statements have several cautions which must be observed whenever using them:

- GOTO statements can only transfer execution within the same procedure, function, or main body of a script. They cannot be used to jump between procedures or between scripts.
- The destination of a GOTO statement must always be the beginning of a statement.
- Jumping to statements that are contained within the structure of other statements can have undefined effects; the VectorScript compiler will not recognize this action as an error.

Repetition Statements

VectorScript supports three methods of executing a section of a script repeatedly—the process referred to as looping. The repetition statements supported by VectorScript are the FOR statement, the WHILE statement, and the REPEAT statement.

The FOR Statement
The WHILE Statement
The REPEAT Statement

The FOR Statement

The FOR statement in VectorScript executes the same script section a specified number of times. This value is held within a control variable which is evaluated by the FOR statement to determine whether execution of the script section should continue.

The general syntax for FOR statements is:

```
FOR <control variable> := <initial value> [TO | DOWNTO] <limit value>
DO <statement>;
```

The initial and final values, or **limit values**, of the control variable are set in the FOR statement. These values may be INTEGER, LONGINT, or CHAR values, and can be either constants or values derived from an expression. The value of the control variable is modified and evaluated by the FOR statement prior to each pass through the script section controlled by the statement.

FOR statements come in two varieties: the FOR-TO statement, and the FOR-DOWNTO statement. In the FOR-TO statement, the value of the control variable is incremented (increased) by one on each pass through the section controlled by the statement. For example:

```
FOR i:=1 TO 10 DO Message('Pass ',i,' through FOR loop.');
```

In the FOR-TO statement, the control variable i will be incremented by one and evaluated on each pass before the Message() function call is executed.

In a FOR-DOWNTO statement, the value of the control variable is decremented (decreased) by a value of one on each pass until the limit value is reached. For example:

```
j:= 9;
FOR i:=10 DOWNTO 1 DO BEGIN
   Message('Pass ',i-j,'(',i,') through FOR loop.');
   j:= j - 2;
END;
```

In the FOR statement, the value of i is decremented on each pass until it reaches the limit value of one. Also note that a compound statement can be used to execute any number of other statements within the FOR statement structure.

The following cautions should be observed when working with FOR statements:

- Do not try to change the limit value of the control variable from within the FOR statement; doing so can lead to unpredictable results.
- Do not include the control variable in either of the limit expressions of the FOR statement.
- If the limit values are equal, the FOR statement will execute its controlled statement exactly once.
- If the limit values are reversed, the FOR statement will be skipped.

The WHILE Statement

The WHILE statement in VectorScript will execute the same script section as long as the control expression, which returns a BOOLEAN value, evaluates to TRUE. The general syntax for the WHILE statement is:

```
WHILE <control expression> DO <statement>;
```

The control expression is evaluated prior to executing the controlled statement, and as such it can bypass the controlled statement altogether. For example:

```
PROCEDURE Example_74;
VAR
h:HANDLE;
BEGIN
h:= FActLayer;
WHILE (h <> NIL) DO BEGIN
      SetSelect(h);
   h:=NextObj(h);
END;
END;
Run(Example_74);
```

In the example, a handle to the first object on the active layer is returned by the FActLayer() function call. If there are no objects on the active layer, the calls to select the object and obtain the next object on the layer are bypassed.

If there were objects on the layer, the example would automatically exit the loop when it ran out of objects to process. This is because the NextObj() call returns NIL when it cannot return a handle, and since the WHILE statement will evaluate the expression before executing its controlled statement, the example would bypass the controlled statement once the expression evaluated to FALSE (h = NIL). Unlike a FOR statement, the WHILE statement allows execution to be controlled from within the controlled statement.

The REPEAT Statement

The REPEAT statement, like the WHILE statement, executes the same script section repeatedly until its control expression evaluates to FALSE. Unlike the WHILE statement, however, the REPEAT statement evaluates the control expression after executing its controlled statement. This means that the controlled statement will always execute at least once.

The general syntax for the REPEAT statement is:

```
REPEAT <statement> UNTIL <control expression>;
```

The example from the WHILE statement section could easily be rewritten using a REPEAT statement:

```
PROCEDURE Example_75;
VAR
h:HANDLE;
BEGIN
```

```
h:= FActLayer;
REPEAT
      SetSelect(h);
   h:=NextObj(h);
UNTIL (h = NIL);
END;
Run(Example_75);
```

In this format, the statements within the REPEAT-UNTIL structure would be executed at least once, whether or not h was initially NIL, which could cause detrimental effects or errors. Generally speaking, REPEAT statements should be used in conditions where executing the controlled statement will not have a negative impact. WHILE statements are most useful when the condition controlling their execution may have already been satisfied; REPEAT statements, on the other hand, are most useful when the condition can be satisfied only by executing the statement.

Also note that REPEAT statements do not require the use of BEGIN or END, as the REPEAT and UNTIL keywords create their own compound statement out of the statements between them.

Conditional Statements

VectorScript supports two methods of making decisions within a script which affect the flow of execution—a process referred to as branching. The conditional statements supported by VectorScript are the IF statement and the CASE statement.

The IF Statement
The CASE Statement

The IF Statement

The VectorScript IF statement evaluates a BOOLEAN control expression and executes a controlled statement only if the expression evaluates to TRUE. IF statements can also be optionally written to execute a second statement if the control expression evaluates to FALSE.

The general syntax for IF statements is:

```
IF <control expression> THEN <statement> [ ELSE <statement>];
```

When an IF statement executes, the control expression is evaluated to obtain a BOOLEAN result. If the result is TRUE, the statement after the THEN keyword is executed and the IF statement is exited. If the expression evaluates to FALSE, the statement is skipped unless the ELSE keyword and a statement are encountered. In this case, the statement after the ELSE keyword is executed. For example:

```
IF (i mod 2) THEN Message('Even value') ELSE Message('Odd value');
```

If the value in i is even, then the expression i MOD 2 will evaluate to TRUE and the statement Message('Even value') will be executed. If the value of i is odd, then Message('Odd value') will be executed.

Note that the statement contained between the THEN and ELSE keywords does not require a semi-colon after it; in this case the ELSE keyword indicates the end of the statement. If the ELSE keyword were omitted, a semicolon would be required.

Like other statements, the IF statement supports the use of a compound statement as the controlled statement. IF statements can also be nested; that is, the statement following the THEN keyword may also be an IF statement. Nesting

allows you to construct statements which can take actions based on the results of several mutually exclusive conditions.

Nested `IF` statements can rapidly become confusing:

```
PROCEDURE Example_76;
VAR
i:INTEGER;
BEGIN
i:= Ord('c');
IF (i > 48) THEN IF (i > 57) THEN IF (i > 65) THEN IF (i > 90) THEN
IF (i > 97) THEN IF(i < 123) THEN Message('Lower case alpha')
ELSE Message('Out of range') ELSE Message('Some punctuation')
ELSE Message('Upper case alpha') ELSE Message('Some punctuation')
ELSE Message('Number') ELSE Message('Out of range');
END;
Run(Example_76);
```

If the matching of `IF` and `THEN` becomes confusing, you can clarify the source code by using compound statements or by applying indentation and comments:

```
PROCEDURE Example_76;
VAR
i:INTEGER;
BEGIN
i:= Ord('c');
{out of range}
IF (i > 48) THEN
    {number}
    IF (i > 57) THEN
        {punctuation}
        IF (i > 65) THEN
            {upper alpha}
            IF (i > 90) THEN
                {punctuation}
                IF (i > 97) THEN
                    {lower alpha}
                    IF(i < 123) THEN
                        Message('Lower case alpha')
```

```
                ELSE
                    Message('Out of range')
            ELSE
                Message('Some punctuation')
        ELSE
            Message('Upper case alpha')
    ELSE
        Message('Some punctuation')
ELSE
    Message('Number')
ELSE
    Message('Out of range');
END;
Run(Example_76);
```

The CASE Statement

The VectorScript `CASE` statement lets you specify a list of alternative statements to be executed, associating a constant with each statement to identify it. When the `CASE` statement is executed it evaluates the controlling expression, and if the result matches one of the constants, it then executes the associated statement. An optional `OTHERWISE` clause allows a different statement to be executed if no other option was selected from the list of constants.

The general syntax for a `CASE` statement is:

```
CASE <control expression> OF
    <constant>:<statement>;
    <constant>:<statement>;
    ...
    ...
    [OTHERWISE <statement>;]
END;
```

The control expression may evaluate to an `INTEGER`, `CHAR`, or `BOOLEAN` value. For example:

```
PROCEDURE Example_77;
VAR
j:INTEGER;
BEGIN
j:= Ord('C');
CASE j OF
```

```
    49: Message('Number');
    77: Message('Upper case alpha');
    110: Message('Lower case alpha');
    OTHERWISE Message('Out of range');
END;
END;
Run(Example_77);
```

The variable `j` evaluates to an `INTEGER` value, and this value is compared to the list of constants in the `CASE` statement. In the example, the value of `j` falls outside of the listed constants, so the `OTHERWISE` clause is executed.

`CASE` statements provide some flexibility when specifying constants. For instance, there may be applications of the `CASE` statement where several cases will need to execute the same code. Rather than use redundant options, the `CASE` statement lets you specify a comma delimited list of constants for a single `CASE` option:

```
PROCEDURE Example_78;
VAR
j:INTEGER;
BEGIN
j:= Ord('C');
CASE j OF
    49: Message('Number');
    58,59,60,61,62,63,64: Message('Non alpha printable character');
    110: Message('Lower case alpha');
    OTHERWISE Message('Out of range');
END;
END;
Run(Example_78);
```

Should the control expression evaluate to any of the values in the list, the associated statement will be executed.

For longer contiguous lists of constant values, the CASE statement also supports the use of ranges within the CASE statement constant specification. These ranges specify a contiguous list of constant values to be associated with a statement to be executed:

```
PROCEDURE Example_78;
VAR
j:INTEGER;
BEGIN
j:= Ord('C');
CASE j OF
```

```
    48..57: Message('Number');
    58,59,60,61,62,63,64: Message('Non alpha printable character');
    65..90: Message('Upper case alpha');
    97..122: Message('Lower case alpha');
    OTHERWISE Message('Out of range');
END;
END;
Run(Example_78);
```

Ranges and comma delimited lists may be mixed for further flexibility in associating constants with an executable statement:

```
PROCEDURE Example_78;
VAR
j:INTEGER;
BEGIN
j:= Ord('C');
CASE j OF
    48..57: Message('Number');
    33..47,58..64,91..96:Message('Non alpha printable character');
    65..90: Message('Upper case alpha');
    97..122: Message('Lower case alpha');
    128,133,134,168..170: Message('Special characters');
    OTHERWISE Message('Out of range');
END;
END;
Run(Example_78);
```

In the example, it can be seen that the available methods of specifying `CASE` statement constants provide the ability to specify complex options for branching in a very concise format. Ranges and lists also work with the other supported constant types:

```
PROCEDURE Example_78;
VAR
j:CHAR;
BEGIN
j:= 'C';
CASE j OF
    '0'..'9': Message('Number');
```

```
    'A'..'Z': Message('Upper case alpha');
    'a'..'z': Message('Lower case alpha');
    OTHERWISE Message('Out of range');
END;
END;
Run(Example_78);
```

Like other statements, CASE statements can also support the use of compound statements as the controlled statement to be executed. Extending this concept, it is also possible to create nested CASE statements to handle even more complex branching in scripts:

```
PROCEDURE Example_78;
VAR
j:CHAR;
BEGIN
j:= 'C';
CASE j OF
    '0'..'9': Message('Number');
    'A'..'Z': Message('Upper case alpha');
    'a'..'z': Message('Lower case alpha');
    OTHERWISE BEGIN
        CASE Ord(j) OF
        33..47,58..64,91..96:Message('Non alpha printables');
        128..159:Message('Accented characters');
        168..170: Message('Special characters');
        OTHERWISE Message('Out of range');
        END;
    END;
END;
END;
Run(Example_78);
```

Some cautions to be observed when using CASE statements:

- Constant values in the CASE statement must have the same type as the value of the controlling expression.
- Constant types may not be mixed in a single CASE statement.

User Defined Functions 8

In addition to the over 1,900 function calls built into the API, VectorScript also lets you create your own **user-defined functions**. By creating these custom functions, you can break large script tasks into smaller ones, and build on the work that you have done previously instead of starting over from scratch. Another term for user-defined functions is **subroutines** which, as the name implies, are pieces of script code which perform tasks within the main script.

User-defined functions come in two varieties: **procedures**, which perform actions but are not associated with a value, and **functions**, which perform actions and also have an associated value that can be used in situations requiring a constant or expression-derived value.

This section describes in detail how to create and use your own procedures and functions, and addresses some of the issues involved in using them within scripts.

User-Defined Procedures
User-Defined Functions
Parameters
Program Blocks and Block Scope

User-Defined Procedures

User-defined procedure subroutines are the most common type of subroutine. They allow commonly used code to be "encapsulated" under a single identifier which can easily be called from within a script.

User-defined procedures are declared after the definition (`CONST`, `TYPE`, and `VAR`) blocks of a script, but before the script body. To create a user-defined procedure to use within a script, you will need to create a procedure declaration statement which associates an identifier with the subroutine and defines how the subroutine is to be used. The general syntax for user-defined procedures is:

```
PROCEDURE <procedure identifier>[(<parameter list>)]
```

The procedure declaration begins with the `PROCEDURE` keyword, and is followed by the identifier to be associated with the subroutine block. After this identifier comes the parameter list for the procedure. The parameter list provides a means for moving data in and out of the subroutine, and the identifiers in the list may be used just like variables within the subroutine block. Parameters and parameter lists will be covered in more detail later in this section.

After the procedure declaration statement has been created, the actual working code of the subroutine is defined. Just like a script, subroutines may have any of the standard VectorScript definition blocks (`LABEL`, `CONST`, `TYPE`, or `VAR`) as well as a script body containing the script code to be executed when the subroutine is called from elsewhere in your script. For example, suppose you wish to take the following script:

```
PROCEDURE SubrExample2;
VAR
   n,sum:INTEGER;
BEGIN
   n:=IntDialog('Enter the limit value','0');
   {sum of squares for the first n integers}
   sum:= n*(n+1)*(2*n+1)/6;
   Message('The sum of squares is: ',sum);
```

```
END;
Run(SubrExample2);
```

and modify it so that the sum of squares code can be easily reused whenever it is needed. To do this, a subroutine is needed to contain the code which performs the operation. Creating the subroutine begins by writing a procedure declaration statement and the skeleton of the subroutine:

```
PROCEDURE SubrExample2;
VAR
   n,sum:INTEGER;
PROCEDURE SumOfSquares(limit:INTEGER;VAR result:INTEGER);
BEGIN

END;
BEGIN
   n:=IntDialog('Enter the limit value','0');
   {sum of squares for the first n integers}
   sum:= n*(n+1)*(2*n+1)/6;
   Message('The sum of squares is: ',sum);
END;
Run(SubrExample2);
```

The declaration statement associates the identifier `SumOfSquares` with the new subroutine. Following the subroutine identifier is the parameter list for the subroutine. This optional list defines a method of moving data in and out of the subroutine. While it is possible to refer to values in the enclosing program blocks directly, doing so would eliminate the ability to easily use the subroutine in other code, which is one of the major advantages of using subroutines.

The parameter list declares a set of identifiers (and their associated data types) that will be used to pass data to and from the subroutine; the `VAR` keyword indicates an identifier that will be used to pass data out of the subroutine to the calling code. Identifiers in the parameter list can be treated as variables and used within the subroutine script code.

When the subroutine is called in the script, the parameter list as shown in the declaration is replaced with a list of variable identifiers that provide and/or receive the data being passed through the parameters. The order and types of the variable identifiers must exactly match those in the declaration.

Now that the skeleton of the subroutine is in place, the summation script code can be moved into the subroutine and modified to work with the subroutine:

```
PROCEDURE SubrExample2;
VAR
   n,sum:INTEGER;
PROCEDURE SumOfSquares(limit:INTEGER;VAR result:INTEGER);
BEGIN
   result:= limit*(limit+1)*(2*limit+1)/6;
```

```
END;
BEGIN
   n:=IntDialog('Enter the limit value','0');
   {sum of squares for the first n integers}
   sum:= n*(n+1)*(2*n+1)/6;
   Message('The sum of squares is: ',sum);
END;
Run(SubrExample2);
```

The final change needed to the script is to modify the main body of the script to use the subroutine:

```
PROCEDURE SubrExample2;
VAR
   n,sum:INTEGER;
PROCEDURE SumOfSquares(limit:INTEGER;VAR result:INTEGER);
BEGIN
   result:= limit*(limit+1)*(2*limit+1)/6;
END;
BEGIN
   n:=IntDialog('Enter the limit value','0');
   {sum of squares for the first n integers}
   SumOfSquares(n,sum);
   Message('The sum of squares is: ',sum);
END;
Run(SubrExample2);
```

Note again that when the script is called in the main program block, the `SumOfSquares` parameter list is replaced by the variables `n` and `sum`. The value contained in `n` is passed into the subroutine, where it is referred to through the identifier `limit`. The resulting value is stored in the local identifier `result`, and is passed back to the main program block and stored in the variable `sum` when the subroutine completes its execution.

By using a subroutine, the script can be broken up into manageable chunks which are easy to understand and to debug. The `SumOfSquares` subroutine can also be reused as many times as needed in the current script, and the subroutine can be copied and used in other scripts.

User-Defined Functions

User-defined functions incorporate all the features of user-defined procedures, but they have one additional feature which makes them extremely useful when writing scripts: an associated value. User-defined functions, unlike procedures, can pass data out of the subroutine through a return value, which associates the value with the subroutine identifier. This means that, like a variable, a function can be used wherever a value is required—in an expression, an assignment statement, or other operation in a script.

User-defined functions, like procedures, are declared between the definition blocks and the body of the script. To create a user-defined function, a function declaration statement will be used to associate an identifier with the subroutine and define how it will be used. The general syntax for user-defined functions is:

```
FUNCTION <procedure identifier>[(<parameter list>)]:<return value type>
```

Just like procedures, the declaration begins with a keyword, in this case the `FUNCTION` keyword, and is followed by the identifier to be associated with the subroutine block. Next comes the parameter list for the function. Parameter lists for user-defined functions work exactly like they do for user-defined procedures, so everything learned in the previous section applies here as well.

User-defined function declarations have one additional requirement: a return value type after the parameter list. This data type indicates what type of data will be passed through the return value mechanism and will be associated with the identifier.

After the function declaration has been created, define the actual working code of the subroutine in the same way you would for a user-defined procedure.

To illustrate the differences between procedure and function subroutines, look at the sum of squares example from the previous section:

```
PROCEDURE SubrExample2;
VAR
   n,sum:INTEGER;
PROCEDURE SumOfSquares(limit:INTEGER;VAR result:INTEGER);
BEGIN
result:= limit*(limit+1)*(2*limit+1)/6;
END;
BEGIN
   n:=IntDialog('Enter the limit value','0');
   {sum of squares for the first n integers}
   SumOfSquares(n,sum);
   Message('The sum of squares is: ',sum);
END;
Run(SubrExample2);
```

The `SumOfSquares` subroutine provides a handy reusable piece of code which is very useful, but the result is returned to the main script in such a way that it is difficult for anyone reading the script code to determine how the value is obtained. In this instance, the return value mechanism of a function subroutine can be used to provide a much more user-friendly method. To create the function subroutine, the first step is to make some changes to the declaration statement:

```
PROCEDURE SubrExample2;
VAR
   n,sum:INTEGER;
FUNCTION SumOfSquares(limit:INTEGER):INTEGER;
```

```
BEGIN
   result:= limit*(limit+1)*(2*limit+1)/6;
END;
BEGIN
   n:=IntDialog('Enter the limit value','0');
   {sum of squares for the first n integers}
   SumOfSquares(n,sum);
   Message('The sum of squares is: ',sum);
END;
Run(SubrExample2);
```

The first change to the declaration is to convert the keyword from `PROCEDURE` to `FUNCTION` to indicate the correct type of subroutine. The output parameter result is then eliminated, since a return value will be used for the subroutine's output. Next, a return value data type is added to the declaration.

Once the declaration statement has been modified, one additional change to the subroutine is needed to associate the result value with the subroutine identifier. VectorScript performs this association by using an assignment statement, except that the identifier used on the left side of the statement is the subroutine identifier:

```
PROCEDURE SubrExample2;
VAR
   n,sum:INTEGER;
FUNCTION SumOfSquares(limit:INTEGER):INTEGER;
BEGIN
   SumOfSquares:= limit*(limit+1)*(2*limit+1)/6;
END;
BEGIN
   n:=IntDialog('Enter the limit value','0');
   {sum of squares for the first n integers}
   SumOfSquares(n,sum);
   Message('The sum of squares is: ',sum);
END;
```

All that is left to do now is to modify the main script to match the new syntax of the function:

```
PROCEDURE SubrExample2;
VAR
   n,sum:INTEGER;
FUNCTION SumOfSquares(limit:INTEGER):INTEGER;
BEGIN
```

```
    SumOfSquares:= limit*(limit+1)*(2*limit+1)/6;
END;
BEGIN
    n:=IntDialog('Enter the limit value','0');
    {sum of squares for the first n integers}
    sum:= SumOfSquares(n);
    Message('The sum of squares is: ',sum);
END;
Run(SubrExample2);
```

As you can see, using a function subroutine in this instance makes for much more readable code, and simplifies the interface of the subroutine as well. In general, functions are best suited to subroutines which return a value that is the result of a calculation or other similar operation. Procedures should be used when creating a subroutine that performs an operation which does not return a value.

Parameters

User-defined subroutines, like the built-in functions of the VectorScript API, make use of parameters and parameter lists to move data values in and out of subroutines.

Formal and Actual Parameters

Formal parameters in VectorScript refer to the parameters which are defined in the parameter lists of built-in or user-defined functions. Formal parameters provide the data interface "template" for the function, indicating the order and typing of the values that will be passed in and out of the function call. Actual parameters refer to the expressions or values that are passed by a function in the body of the script. For example, in the declaration statement of the subroutine `SumOfSquares`:

```
PROCEDURE SumOfSquares(limit:INTEGER;VAR result:INTEGER);
```

The identifier's limit and result are both formal parameters of the subroutine procedure. When `SumOfSquares` is used in the script:

```
SumOfSquares(n,sum);
```

the subroutine procedure has two actual parameters, `n` and `sum`. These actual parameters contain the data used and returned by the function call. Checking the `VAR` block of the script, notice that the data types of the two identifiers match the types found in the formal parameter list.

Value and Variable Parameters

Value parameters in VectorScript are parameters which are used to pass data values into a subroutine. Within the subroutine, they act just like local variables except that they obtain their initial value from a corresponding actual parameter in the parameter list. In the `SumOfSquares` example:

```
PROCEDURE SumOfSquares(limit:INTEGER;VAR result:INTEGER);
```

the identifier `limit` is a value parameter, or more fully, a formal value parameter. In the function call of the main script:

```
SumOfSquares(n,sum);
```

the value contained in the variable `n` would be assigned to the value parameter `limit` for use within the subroutine.

Variable parameters in VectorScript are the opposite of value parameters—they are used to pass data values out of a subroutine. They are denoted by the `VAR` keyword which precedes them in the parameter list, and like value parameters, act as local variables within the subroutine. In the `SumOfSquares` example:

```
PROCEDURE SumOfSquares(limit:INTEGER;VAR result:INTEGER);
```

the identifier `result` is a formal variable parameter, which can be used within the subroutine script code to pass values back to the calling code. In the function call of the main script:

```
SumOfSquares(n,sum);
```

the value contained in the variable parameter `result` would be assigned to the variable `sum` when the subroutine finished execution.

Program Blocks and Block Scope

As mentioned in "An Example Script" on page 2, a script can be referred to as a program block, which is the basic unit of VectorScript source code. Program blocks consist of a block declaration statement, sections such as the `CONST`, `TYPE` or `VAR` blocks for declaring or defining data within the block, and the body of the block, which contains the VectorScript source code to be executed. User-defined functions extend this concept, and are in fact smaller program blocks nested within the main program block that is your script.

Each subroutine that will be used in a script is a self-contained program block, with its own data declarations and body. Subroutine blocks can also have nested subroutine blocks of their own, with other data declarations and script code. Such nesting of subroutine blocks brings up an important concept, block scope, that should be considered whenever writing subroutines for scripts.

Block scope describes the area of a script where a given identifier is considered valid and has a defined value associated with it. Whenever a variable, constant, or structure is declared in a program block, the item is said to be **local** to that program block. This means that the item will only be valid and have a defined value in the block where it was declared, as well as in any areas which are enclosed by that block. For example:

```
PROCEDURE Main;
    Subroutine "A"()
        Subroutine "B" ()
        BEGIN
        END;
    BEGIN
    END;
    Subroutine "C"()
    BEGIN
    END;
BEGIN
END;
```

In the example, the scope of an identifier is determined by its location:

Identifier Declaration Location	Identifier Scope
Main	Main,A,B,C
Subroutine "A"	A,B
Subroutine "B"	B
Subroutine "C"	C

An identifier is considered undefined outside the program block where it was declared and may not be accessed or referred to in script code outside of the block. If the block in which the identifier is declared is a subroutine, this means that the identifier will be undefined in any block enclosing the subroutine. Any attempt to refer to or evaluate the item from source code in the blocks enclosing the subroutine will cause an error and will cause the script to fail.

The following example also illustrates the concept of block scope:

```
PROCEDURE WoodPrice;
CONST
   kTax:=0.05;
VAR
   boardFeet,price,totalCost:REAL;
PROCEDURE CalcCost(feet,ppf:REAL; VAR cost:REAL);
VAR
   baseCost:REAL;
FUNCTION AddTax(rawcost:REAL):REAL;
BEGIN
   AddTax:= rawcost+(rawcost*kTax);
END;
{ begin CalcCost code }
BEGIN
   baseCost:= feet*ppf;
   cost:= AddTax(baseCost);
END;
{ end CalcCost code }
{ begin main script }
BEGIN
   boardFeet:= RealDialog('Enter no. of feet','0');
   price:= RealDialog('Enter price per foot','0');
   CalcCost(boardFeet,price,totalCost);
   Message('Total cost is $',totalCost:6:2);
END;
```

```
{ end main script }
Run(WoodPrice);
```

In the example there are three program blocks, or areas of scope. The largest block is the main script, `WoodPrice`; contained within it is the subroutine block `CalcCost`, and within `CalcCost` is the subroutine function and program block `AddTax`.

Any variable or constant identifiers defined in the `WoodPrice` block can be referred to in the `WoodPrice` script code, and can also be referenced from within any of the subroutines declared within the block. These items are said to have global scope because they are defined at the top level of the script, and can be accessed from any subroutine within the script.

Identifiers defined in the `CalcCost` subroutine (including those in the declaration statement) can be referred to in the `CalcCost` subroutine, or within the `AddTax` function. They are undefined, however, in the `WoodPrice` block, which lies outside the `CalcCost` scope. This means that items such as `baseCost` or the subroutine `AddTax` cannot be referenced directly from the main body of the `WoodPrice` script.

The identifiers defined in the `AddTax` subroutine have the smallest scope of any of the blocks in the script; they are available only to code contained within that subroutine. They are undefined for and cannot be referenced from the `CalcCost` and `WoodPrice` program blocks. In the example, the `kTax` constant can be referenced directly in the `AddTax` function because `kTax` is defined in the main script and has global scope. The result of `AddTax`, however, cannot be accessed directly from the main script, since it is declared within the `CalcCost` subroutine and is only valid within that subroutine.

Compiler Directives 9

VectorScript supports compiler directives for controlling how scripts are compiled and executed.

{$INCLUDE}
{$DEBUG}
{$NAMES}
{$STRICT}
{$VER}
Conditional Compile Directives

{$INCLUDE}

The include directive instructs the compiler to insert source code from an external file at the position of the include directive statement. The syntax for an include directive is:

```
{$INCLUDE <file path>}
```

The path to the file containing VectorScript source code may be either a fully specified or partial file path. Macintosh style path delimiters (:), or Windows style path delimiters (\ ) are supported. Windows style delimiters are recommended for scripts which may be used in a cross-platform environment to ensure compatibility on all platforms.

Example: Macintosh-style include directive

```
{$INCLUDE MyHD:Vectorworks:Projects:VS:mycode:math.vss}
```

Example: Windows-style include directive

```
{$INCLUDE MyHD\Vectorworks\Projects\VS\mycode\math.vss}
```

Include files specified without any path information are assumed to reside in a predefined default path relative to the script. For document scripts and scripts run from text files, the default path is the location of the Vectorworks application. For plug-ins, the default path is assumed to be the folder where the associated plug-in is located.

Include statements may also be chained by specifying include directives in other include files. Chaining include directives should be used with care, as it can cause file dependencies which may cause scripts to fail under certain circumstances.

Caution should also be exercised when positioning include directives in your scripts to avoid calling functions before they are defined within the script.

{$DEBUG}

The debug directive instructs the compiler to launch the VectorScript debugger when compiling and executing the script. The debugger may then be used to observe and control script execution during script development. The syntax for the debug directive is:

```
{$DEBUG}
```

The directive may be positioned anywhere within the main block of the script to invoke the debugger.

Details on using the debugger can be found online at http://developer.vectorworks.net.

{$NAMES}

The names directive instructs the compiler to recognize only the identifiers which are valid for the Vectorworks version specified in the compiler directive. Identifiers screened by this directive include procedure, function, and constant identifiers. The syntax for the names directive is:

```
{$NAMES <version number>}
```

Identifiers which are not defined for the specified version of the product will generate a VectorScript error. The names directive is intended for use in testing compatibility of scripts with different versions of Vectorworks.

Example: Names directive

```
{$NAMES 8}
```

In the example, the VectorScript compiler will recognize only those identifiers valid for Vectorworks 8. Any identifier names not supported by the compiler (such as new functions in subsequent versions) will return an error, and should not be used in scripts that must be compatible with the version specified in the directive.

{$STRICT}

The strict directive instructs the compiler to recognize observe syntax and semantic rules which are valid for the Vectorworks version specified in the compiler directive. The syntax for the strict directive is:

```
{$STRICT <version number>}
```

Syntax which is not valid for the specified version will generate a VectorScript error. The strict directive is intended for use in testing compatibility of scripts with different versions of Vectorworks.

Example: Strict directive

```
{$STRICT 7}
```

In the example, the VectorScript compiler will recognize only syntax conventions valid for MiniCAD 7. Any new syntax conventions not valid in this version (such as dynamic arrays or structures) will return an error, and should not be used in scripts that must be compatible with the version specified in the directive.

{$VER}

The `{$VER}` directive suppresses warnings from functions that were deprecated before the specified version number. The syntax for the directive is:

```
{$VER <number>}
```

The `<number>` refers to the Vectorworks software version. For example, Vectorworks version 2015 is number 20, Vectorworks 2016 is number 21, and so on.

Conditional Compile Directives

VectorScript includes four comment directives that allow scripters to control which portions of a VectorScript will be compiled and which will be skipped. This allows for better structuring of complex projects and makes testing of functionality fixes easier.

The directives determine if a block of code should be compiled; the directives are processed before the compiler begins to process the script. Place the directives at any point in a script where a comment is allowed.

{$IF}

This directive defines the beginning of a block of code. The syntax for the directive is:

```
{$IF <condition>}
```

It contains a logical statement that will be evaluated. If the evaluation results are false, the block of code until the {$ENDIF} directive (or until the end of the file) is skipped by the compiler; if the evaluation results are true, the code is compiled.

A built-in ver variable exists to be used within the {$IF} condition. The value of ver corresponds to the version of Vectorworks software. A value of 17 indicates Vectorworks version 2012, a value of 18 indicates Vectorworks version 2013, and so on.

{$ENDIF}

This directive defines the end of a conditional block of code. The corresponding end directive of an {$IF} directive, both directives should be inside the same include file.

The syntax for the directive is:

```
{$ENDIF}
```

In this example, the {$IF} and {$ENDIF} directives control the code based on the specified version number. The example shows a different dialog box depending on the Vectorworks software version.

```
PROCEDURE Test;
BEGIN
{$IF ver = 17}
   AlrtDialog('Code for Vectorworks 2012');
{$ENDIF}

{$IF ver > 17}
   AlrtDialog('Code for versions later than Vectorworks 2012');
{$ENDIF}

END;
RUN(Test);
```

{$DEFINE}

This directive allows defined named values to be used within {$IF} directive conditions; this named variable is valid from the moment it is defined until the end of the script, or until it is undefined.

The syntax for the directive is:

```
{$DEFINE <name> = <value>}
```

This example uses the {$DEFINE} directive to control the code:

```
PROCEDURE Test;
   {$DEFINE library = 1}

   {$IF library = 1}
      {$INCLUDE Code/Library.px}
   {$ENDIF}

BEGIN
   {$IF library = 1 & ver > 17}
      LibraryFunction( 34 );
   {$ENDIF}
END;
RUN(Test);
```

{$UNDEF}

This directive removes a named variable that was defined previously. The syntax for the directive is:

```
{$UNDEF <name>}
```

Index

Symbols

{$DEBUG} 67
{$DEFINE} 69
{$ENDIF} 69
{$IF} 69
{$INCLUDE} 67
{$NAMES} 68
{$STRICT} 68
{$UNDEF} 70

A

Actual parameters 62
Arrays (VectorScript)
  dynamic 22
  index of 21
  static 21

B

Block scope 63
Branching in VectorScript 50

C

CASE 52
Comments 6
Compound expressions 35
Conditional compile directives 68
CONST block 12
Constant definition 12
Constants 12

D

Data types
  BOOLEAN 15
  CHAR 14
  HANDLE 15
  INTEGER 13
  LONGINT 14
  REAL 14
  STRING 14
  VECTOR 15
Delimiters 5
Dynamic arrays 22
  ALLOCATE 23
  dimensioning 23
  extended string support with CHAR arrays 25
  one-dimensional dynamic array 22
  performance considerations 25
  two-dimensional dynamic array 23

E

Expressions
  arithmetic operators 36
  associativity 36
  comparison operators 38
  complex expressions 35
  logical operators 39
  operator precedence 36
  simple expressions 35

F

Floating-point values 14
FOR..DOWNTO 48
FOR..TO 48
Formal parameters 62
Fundamental types 13

G

Global scope 65

I

Identifiers 8
IF..THEN 50

K

Keywords 8

L

Literals 6
  BOOLEAN literals 7
  floating-point literals 7
  integer literals 6
  NIL 8
  string literals 7
Looping in VectorScript 47

O

Operand 35
Operators 35
  arithmetic 36
  array access 40
  assignment 40
  associativity 36
  binary 36
  comparison 38
  logical 39
  member access 41
  precedence 36
  unary 36

P

Parameter list 57
Program block 63

R

REPEAT..UNTIL 49
Reserved 8
Reserved words 8
Return value 59

S

Special symbols 6, 9
Statements
  FOR..DOWNTO 48
Statements in VectorScript
  assignments 43
  compound 46
  constant ranges with CASE 53
  control expressions in CASE statements 52
  FOR..TO 48
  GOTO 47
  IF..THEN 50
  procedures 46
  REPEAT..UNTIL 49
  WHILE..DO 49
Static arrays 21
  accessing an array element 22
  one-dimensional static array 21
  two-dimensional static array 22
Structures
  member access 32
  members 31
Subroutines in VectorScript 57
Symbols 5

T

Tokens 5
TYPE block 31

U

User-defined
  function 59
  procedures 57
  types 13

V

Value parameters 62
VAR block 11
Variable declaration 11
Variable parameters 63
Variables 11
Vectors and array notation 25

W

WHILE..DO 49