VectorScript Language Guide

VS

This Language Guide is intended to explain the intricacies of the VectorScript Language. It provides basic concepts needed to understand the language as well as explanations of the individual language components. VectorScript is based on the Pascal programming language in that it has similar constructs; however it should not be confused with Pascal which is a full featured high-level programming language. VectorScript is a scripting language which requires no compiling.

The process of creating and editing scripts using the VectorScript Editor is contained in your User's Manual, Chapter 19. Ready-to-use Functions and Procedures are accessed from the VectorScript Editor. These functions and procedures are defined in your on-line help for easy reference while creating scripts. The information contained within this manual can be used when creating your own scripts which entail a detailed understanding of VectorScript's language constructs.

BASIC CONCEPTS

VectorScript provides a comprehensive set of predefined functions which provide access to the objects in a VectorWorks document. These can be supplemented with any user defined functions contained in a script. The VectorScript language contains identifiers, statements, expressions and operators.

The most basic script of most programming languages is the script that produces the Hello World string of text. The following example presents the HelloWorld script in the VectorScript language.

```
PROCEDURE HelloWorld;
BEGIN
   Message('Hello, World!');
END;
Run(HelloWorld);
```

Notice that in this example there is a user-defined procedure (HelloWorld), a pre-defined procedure call (Message). This script shows a similarity between VectorScript and the Pascal language. One difference between VectorScript and Pascal is the need for the Run() call at the end of the script to indicate where execution should begin.

The following sections describe the basic concepts behind the VectorScript language and the handling of data types, numbers, and strings.

Identifiers

An Identifier:

- variable, constant, function, and procedure names are identifiers
- stored to 20 characters
- case insensitive
- cannot be a reserved word
- can't redefine a predefined or standard identifier (VectorScript differs from Pascal here)

You can use standard identifiers which make up the standard pascal library calls and are part of the VectroScript language. These include: write, writeln, sin, cos, chr, and ord among others. Predefined identifiers include the 650 plus VectorScript functions and procedures which manipulate VectorWorks and its objects. Standard and predefined identifier types behave the same. User defined identifiers include any function, procedure and variable created by a user. The names must follow identifier naming conventions and can not conflict with existing identifiers.

Spaces

- spaces can not be inserted into the middle of identifier names, reserved words or multi character operators (>=)
- spaces must be inserted to clarify adjacent identifiers and reserved words

```
for index := first to last do
```

- Inside single quotes, a space character is literal
- Otherwise, spaces are ignored
- Spaces can and should be used to make the script more readable such as indenting compound statements.

Statements

Statements are the individual elements of a script which perform an action.These consist of:

- uses of built-in VectorScript statements such as IF, CASE, WHILE, FOR, REPEAT, and GOTO.
- calls to predefined or user defined procedures
- assignments (using := operator)

Simple statements can be combined into a compound statement by including them within a Begin...End block. Compound statements can be used anywhere a statement can be used.

Expressions

An expression describes an algebraic or logical computation which results in a value. Expressions combine constants literals, variables, function results, and the temporary results of sub-expressions according to the rules of algebra and logic. Expressions can be used anywhere a value is accepted.

Operators

Operators provide the built-in algebraic and logic capabilities used in VectorScript in evaluating expressions. Expressions are evaluated observing the precedence of the operators.

Data Types

In VectorScript, as with other programming languages, the information which you use and process, also known as the data, can be categorized according to types. These types are important for a number of reasons. Different types of data require different amounts of storage space in memory; VectorScript uses these types to make sure there is enough memory reserved for your script to run properly. Data types are also used to check for logic errors in your scripts; for instance, multiplying a word by a number would result in a syntax error.

In VectorScript the following types of data are allowed:

Data Type	Explanation
BOOLEAN	Boolean data values may have one of two values: TRUE or FALSE. Boolean data values are used to make decisions within VectorScript scripts
CHAR	A char data value holds a single character, such as a letter, punctuation mark, etc.
HANDLE	A handle data value is a specialized object identifier within VectorScript, sort of a “serial number” for the object. Each object within a VectorWorks document can be accessed and its object data (attributes, measurements) can then be retrieved or modified. Handles to objects may change as the drawing is manipulated. They cannot be saved between separate executions of a script.

Data Type	Explanation
STRING	String data values are sequences of characters, such as words or sentences. String data values may take from 0 to 255 characters per STRING value.
Structure Types	
ARRAY	An array data value stores a group of other values of the same type, which can then be accessed through the name of the array and a location within the array.
VECTOR	Vector data values store x, y, and z data for performing mathematical vector operations within VectorScript.
Primitive Types	
INTEGER	Integer data values are the positive and negative counting numbers(ex., -3,0,4,23). Integer values may range from -32,768 to 32,767 (-2E15 to 2E15-1). No fractions or decimals are allowed in integer data.
LONGINT	Long integer data values are for storing larger whole numbers whose range will exceed that of integer values. LONGINT values may range from -2,147,483,648 to 2,147,483,647 (-2E31 to 2E31-1). No fractions or decimals are allowed in LONGINT data.
REAL	The REAL type is a 64 bit (IEEE double precision floating point number capable of representing approximately 1.79E 308 with about 15 digits of precision.

Numbers and Strings

The two types of data you will be most commonly handling in VectorScript will be numeric data (or numbers), and character data (strings). Whether they are handled literally (using the actual value) or through variables, knowing the format of this data is essential to programming in VectorScript.

Numeric data in VectorScript may be formatted using either decimal or fractional conventions. Exponential notation is also supported, using the ‘E’ method of specifying the exponent value.

Example - Supported numeric formatting

```
    0.256724 1/165E7 52.74E-3-256
```

Entering Coordinate data

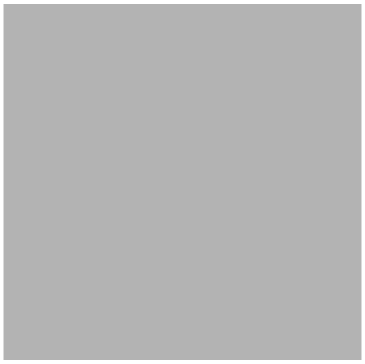
VectorScript supports several methods for entering 2D coordinate data. These methods allow the user to choose the method of inputting data that best fits their needs.

Absolute/Relative point methods:

The default mode of VectorScript is absolute mode. In this mode, values entered as parameters for procedures are assumed to be actual coordinate values relating to VectorWorks' coordinate system. For example, if the user specifies:

Example - Absolute Method

```
      Absolute;
      Rect(0,1/2,1/2,1);
      Rect(1/2,1/2,1,0);
```



the values are assumed to be absolute X-Y coordinate pairs.

In relative point mode, values are treated as X and Y offsets from the current position of the graphics pen. When any object is drawn in VectorWorks, a virtual “pen” is used to draw from point to point. When the operation completes, the pen will remain at that position until another draw routine is called.

The relative method can be used effectively to draw objects whose location would make entry of coordinate data cumbersome, or if the user knows the dimensions of an object and wishes to be able to easily draw it at any location.

Example - Relative Method

```
      MoveTo(1/16, 5/32);
           Relative;
           { NOTE:when in relative mode,}
           {    the poly’s first point is implicit}
           Poly(1,0, 1,1, -2,0, 0,-1);
```

The example above shows how the relative method can be used to draw an object. By using the relative method, the object was drawn by specifying the offset locations of all the vertex points, essentially telling the graphics pen the path to follow to draw the polygon. It would have been cumbersome to specify all of the coordinates in Absolute mode.

At the beginning of execution, a script is always in absolute coordinate mode. Procedures Absolute and Relative are used to switch VectorScript between these modes. Users should be careful to set the mode appropriately to ensure proper execution.

Distance-Angle Method:

VectorScript supports one additional coordinate entry format, the distance-angle format. Distance angle format specifies coordinate locations in terms of a distance and direction angle, similar to polar coordinates found in mathematics. When specifying distance and angle, the general format is to specify the distance in place of the X coordinate, and the angle in place of the Y coordinate. The pound symbols are used to denote that an angle value follows, not a coordinate value. When using Distance-angle coordinates, the point is always specified as a relative offset from the current pen location.

Example - Distance-Angle Method

```
MoveTo(-1, 0);
     Poly(0.25,#0, 0.5,#80,
        1,#-80, 1,#80, 1,#-80, 1,#80, 1,#-80, 1,#80, 1,#-80,
        0.5,#80, 0.25,#0);
```

VectorScript also supports various methods for entering angular information, described in the following table.

	Example
Integer degrees	RECT(2,#90,2,#0)
Decimal degrees	RECT (2,#89.5,2,#359.5)
Degree symbol	RECT (2,#90°,2,#0°)
Degrees, Minutes, Seconds	RECT(2,#90d30'0",2,#0d30'0")
Grads	RECT (2,#100g, 2,#0g)
Radians	RECT (2,#1.57r,2,#0r)
	RECT (2,#1/2 _,2,#0_)
Surveyor's units	RECT (2,#N 45d30'0" E,4,#S45d30'0" E)

When using surveyor's units, users should consult the section on Procedures AngleVar and NoAngleVar in order to ensure that input data is interpreted correctly.

String Data

String data is usually found, when not in a variable, as a quoted string constant. This term describes a character string enclosed in single quotes that is between 0 to 255 characters in length, and is constructed from the ASCII character set. The following example shows a quoted string constant.

Example - Quoted string constant

```
'I am a quoted string constant'
```

When specifying string constants in VectorScript, you should remember three important points:

- Each string must be enclosed in single quotes
- Spaces count as characters.
- The maximum number of characters in one string is 255 characters.

Another example of a string constant in use is the value 'Hello, World' in the HelloWorld script.

While it is important to know how to handle numeric and character data directly in VectorScript, most of the time you will be handling your data through the use of variables. Variables provide the flexibility that allows your scripts to adapt to your needs. To use them effectively, you will need to learn more about how variables are used and specified.

VARIABLES

In VectorScript, as with most programming languages, the primary means of storing and transferring data is through variables. Variables are identifiers associated with a reusable storage location for a data value. This value can then be retrieved from the location for output or use in other ways.

VectorScript attempts to ensure that scripts assign appropriate data to variables. All VectorScript variables must be declared with a type before they are used.

If necessary, VectorScript will convert (coerce) data from one type to another when it can do it accurately and unambiguously. Otherwise, an attempt to store an incompatible type in a VectorScript variable will result in an error.

The variable's data type is defined at the beginning of the procedures, in the special areas known as the variable declaration (VAR) section. All data types in VectorScript are supported as variables.

In VectorScript, variables for all supported data types are defined in the variable declaration section of the VectorScript script. This section begins the use of the VAR reserved word, after which variables are declared. It is terminated by the BEGIN reserved word, which defines the executable section of the script.

Syntax

```
VAR
   <variable ident 1>,<variable ident 2>,…,<variable ident n > :<data type>;
   <variable ident 1>,<variable ident 2>,…,<variable ident n > :<data type>;
```

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

Declaration of variables is formatted as follows:

```
<the variable name> : <data type> ;
```

For example, to declare the variable myMessage, of type STRING, the following declaration statement would be used:

```
myMessage : STRING;
```

Multiple variables of a specific type can be declared in a comma delimited list, as shown:

```
i, j, k, l, m : INTEGER;
```

In addition, multiple declarations of the same data type may be included. This is often useful in large scripts for logically grouping related variables, as shown:

```
height,width,length:REAL;
objName:STRING;
price,unitprice : REAL;
```

Declaration of variables reserves sufficient memory for storage of all variable data during script execution. Variables declared at the beginning of the main script routine (or program block) will persist throughout execution of the script. In addition, variables can be declared local to script subroutines; these variables will only persist as long as the subroutine is within scope, and will be destroyed when the subroutine is exited.

Variables may be named according to the preferences of the programmer, though they are subject to the following rules:

- Variable names may be of any length, but it is the first 20 characters that are used by VectorScript to identify the variable.
- Variable names are not case sensitive; upper and lowercase letters are equivalent.
- Variable names must be comprised of letters, digits, and underscores; in particular, spaces are not allowed.
- Variable names must begin with a letter.

The following are examples of valid variable names:

```
fubar1      line_wt        TheBestLayerEver  A_BigNumber
```

The following examples are NOT valid:

```
      2546a    32array      -ABC-        BEGIN
```

BEGIN is a reserved word, which cannot be used as a variable, since it would create a conflict, and an error would be generated.

Variable values are always undefined at the start of the program block in which they are declared. It is up to you to properly initialize variables prior to use.

CONSTANTS

Constants are values in a VectorScript script routine which can not change throughout the entire duration of execution. These values differ from variables in that they cannot be modified by the script. Constants are defined in a special section of the script (known as the constant definition section), preceded by the reserved word constant declaration format CONST. Each constant is represented by an identifier, and each constant definition is separated by a semicolon.

Syntax

```
CONST
   <constant identifier 1>=<value>;
   <constant identifier 2>=<value>;
   . . .
   <constant identifier n>=<value>;
```

The general format is as follows :

```
Procedure Foo;
CONST
   constant identifier = constant value;
   constant identifier = constant value;
   ..
   etc.
VAR
   etc.
```

Constants can be used whenever a value is needed throughout a script (they follow variable scope rules). For example, if you are performing calculations which require a value to be used throughout your code as a multiplier (the value PI, for example), this multiplier could be defined as a constant.

There is also a predefined constant for use in VectorScript: NIL. This value is returned when a VectorScript procedure is unable to return a value, such as a handle to an object. NIL can also be used to initialize variables prior to use.

This constant can be used as a comparison value when evaluating the result of a relational expression. For example, when processing through objects by handle, if no objects are left in the traversal list, NIL will be returned.

RESERVED WORDS AND SPECIAL SYMBOLS

As with any language, programming or otherwise, there is a basic vocabulary from which you can construct something meaningful. In VectorScript, reserved words and special symbols make up this vocabulary, allowing the programmer to construct a script which is meaningful to the VectorScript interpreter.

Reserved words constitute important markers to VectorScript. Without them, there would be no way for VectorScript to derive a meaningful set of instructions from your code. These words provide a framework which allows the VectorScript interpreter to read and understand your code.

The meanings of reserved words and special symbols cannot be changed.

The following is a list of reserved words in VectorScript:

AND	FOR	REPEAT
ARRAY	FUNCTION	THEN
BEGIN	GOTO	TO
CASE	IF	UNTIL
CONST	LABEL	VAR
DIV	MOD	WHILE
DO	NOT	
DOWNTO	OF	
ELSE	OTHERWISE	
END	PROCEDURE	

Special symbols provide a similar function, telling the VectorScript interpreter how to act on numeric data and variables found in your code. The following is a list of special symbols in VectorScript.

+	-	*	/	=	<	>
.	‘	,	(	)	:	;
{	}	|	&	#	^	$
•	@	[	]			

The following pairs of characters are also considered special symbols.

<>	<=	>=	:=	..	**

The uses of specific reserved words and special symbols will be explained in greater depth later in this manual.

DELIMITERS

Delimiters act as separators between the discrete objects (variables, statements, operators) in your source code so that the VectorScript interpreter can distinguish them as individual items. Spaces, tabs, and carriage returns are the principal delimiters found in VectorScript.

In addition, special symbols act as delimiters along with their other functions. Thus, the VectorScript interpreter can understand the expression

```
number_of_planets:=7+two;
```

because the := and + act as delimiters between the other components of the expression.

COMMENTS

Comments are essentially informational remarks which can be included in the VectorScript source code to describe and clarify the way the VectorScript script works. They are important as a way of providing a quick means of reorienting yourself with the code should you need to work with it in the future, and are of great importance in allowing others to understand your code. It is highly recommended that you always place comments in your code.

Comments are placed in the VectorScript code between opening and closing braces, as shown in the example.

Example - Comment

```
{I have no idea what this code does- I didn’t comment it}
```

Comments are ignored by the interpreter when executing a VectorScript script.

LABELS

Labels are a specialized identifier that is used with GOTO statements. GOTOs allow the programmer to jump to any point in the program, and the label is used to mark the target point of the GOTO statement. Labels are defined in their own special section in the code, in a manner similar to that of constants.

Syntax

```
LABEL
   <label identifier 1>,
   <label identifier 2>,
   . . .
   <label identifer n>;
```

The label declaration section is formatted as follows:

```
Procedure Foo;
LABEL
   label identifier 1,label identifier 2,...,label identifier n;
VAR
   etc.
```

where the label identifier is a number between 1 and 9999.

When using labels in VectorScript scripts, the label is always located preceding a VectorScript statement, with the label identifier being followed by a colon. The label is then referenced by the GOTO statement, and during execution, the program flow of the script will jump to the label location when the GOTO is encountered. For more information on GOTO statements, see “Control Statements” on page 1-39.

Example - Label

```
Procedure JmpTolabel;
LABEL
   1,2;
VAR
   IntegerVal : INTEGER;
BEGIN
   IntegerVal:=5;
   IF IntegerVal = 5 THEN
     GOTO 1;
   Message('Integer was not equal to 5');
   GOTO 2;
1: Message('Integer equals 5');
2: END;
Run(JmpTolabel);
```

ASSIGNMENT STATEMENTS

Assignment statements perform a basic operation of VectorScript; they assign a value to a variable. In the examples earlier in the manual, you have already seen numerous instances of the assignment statement being used. Assignment statements are the primary method of assigning and moving values in VectorScript.

Syntax

```
<variable identifier>:=<the value to be assigned>;
```

The value being assigned can be represented by an explicit value (52, ‘VectorWorks’, TRUE), another variable, the result or return value of a function or an arithmetic expression, or the returned value from a procedure statement. The special symbol “:=”, also known as the assignment operator, is used to indicate to the interpreter that the value to be assigned is to be stored in the specified variable.

Assignment statements follow a set of rules governing which data types can be assigned to which type of variable.

- Coercion between numeric types is automatic REAL, INTEGER, and LONGINT can be assigned to each other without restriction. Overflows produce undefined results.

- Coercion between string types is automatic. CHAR and STRING variables can be assigned to each other.

The next examples show how assignment statements can be used in a VectorScript script.

Example - The assignment statement

```
Procedure AssignVals;
VAR
   RealValue1,RealValue2,Result:REAL;
BEGIN
   RealValue1:=6;
   RealValue2:=2;
   Result:=0;
   Result:=RealValue1*RealValue2;
   Message(Result);
END;
Run(AssignVals);
```

This example shows how values can be assigned explicitly and by other means. In the example, *RealValue1* and *RealValue2* are assigned explicit number values. *Result* is also assigned a value, illustrating a bit of good programming practice, initializing your variables. Also shown is an assignment statement where a value is assigned that is the result of an arithmetic expression. The various methods of assignment allow you a wide range of ways to move and manipulate your data.

STRING and BOOLEAN values can also be used with assignment statements. The next example shows how a STRING value is manipulated with the assignment statement.

Example - The assignment statement

```
Procedure AssignVals2;
VAR
   StringValue1,StringValue2:STRING;
   CharVal1:CHAR;
BEGIN
   StringValue1:='This is ';
   CharVal:=’a’;
   StringValue2:=’ nice string’;
   StringValue1:=Concat(StringValue1,CharVal,StringValue2);
   Message(StringValue1);
END;
```

```
      Run(AssignVals2);
```

Using assignment statements with STRING values, is very similar to using them with numeric values. In the first three lines of the actual procedure, each variable is being assigned an explicit value, in this case two literal strings and a character value. The next line in the procedure illustrates how the assignment statement can be used to assign a value, returned from a VectorScript standard function, to a variable. In this case, the result from Concat, which takes different strings and combines them into one, is assigned to *StringValue1*.

You might have also noticed that *StringValue1* is used by Concat as a parameter. This shows not only how variables can be reused, but also a key concept in how VectorScript executes. In VectorScript, the assignment of the value to the variable will always occur AFTER any calculations, comparisons, or calls to standard procedures. This method of working with variables can be used to your advantage in your VectorScript scripts.

COMPOUND STATEMENTS

In VectorScript, you will often need to treat several other statements as a group (for instance, when executing several statements as part of a loop). To do this, you will need to use a compound statement.

The general format of a compound statement is:

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

If you noticed that the main part of a VectorScript script is a compound statement, you are correct. The body of every VectorScript script and subroutine consists of a single compound statement.

Compound statements can be nested as many times as you like; however, every BEGIN must have a matching END keyword. A mismatch of the BEGIN-END pair will cause the VectorScript interpreter to generate an error, and your script will not run.

PREDEFINED PROCEDURE STATEMENTS

Procedure statements are probably the most common statement type used in VectorScript. Procedure statements call a predefined VectorScript routine to perform an action in the document. You have already seen several instances of procedure statements in previous examples. Provided procedure and functions are described in the on-line HTML reference available through the help system.

EXPRESSIONS

An expression describes an algebraic or logical computation which results in a value. Expressions combine constants, literals, variables, function results, and the temporary results of sub-expressions according to the rules of the algebra and logic.

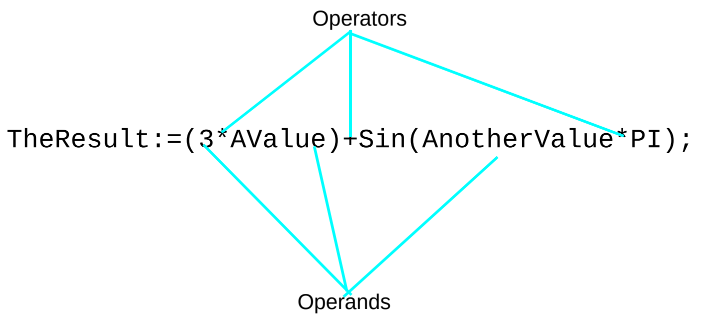
Operators and Operands

All expressions consist of two parts: operators and operands.

Operators are special symbols or reserved words which tell the interpreter what actions to perform. They are usually categorized as either arithmetic operators (which indicate a mathematical operation to be performed), relational operators (which compare 2 operands), or logical operators (which evaluate the “state” existing between two operands).

Operands are the data values which are used as input for the expression. This data can be in the form of a literal value (as shown above), but it can also be a variable, a constant, a function which returns a value, or another expression.

Example - Operators and Operands



The example shows all four described ways of providing operand data. The VectorScript interpreter processes this expression by first performing the operation within the first parentheses, which is itself an arithmetic expression. Next, the sine operation is performed by calling the Sin procedure statement, using the product of a constant PI and the value contained within the variable as the operand. The results of these operations are then used as the operands for the addition operation. The result of the expression is then assigned to the variable TheResult.

Arithmetic Expressions

Arithmetic expressions are exactly what the name implies— expressions which execute a mathematical operation in VectorScript. These expressions usually consist of numeric values as operands, either literally or in variables, combined with operators which indicate the mathematical operation to be performed. These operations can be performed singly, or can be combined or chained to form larger expressions.

Arithmetic expressions in VectorScript support all the four standard mathematical operations, along with exponentiation and modulo (remainder) division. VectorScript also contains a library of predefined mathematical procedures to provide support for trigonometric, logarithmic, and other operations.

Example - Arithmetic Expressions

```
myValue := 3 * 2;cost := (basePrice * 1.5) + tax;
slope := rise / run;cotng := 1 / Tan(angleValue);
```

When working with arithmetic expressions, it is important to consider what numeric types you are using. The difference in numeric data types found in VectorScript can have serious implications in your calculations, and you will have to consider these differences when writing your scripts. The example below provides some insight as to what problems may arise in arithmetic using the different numeric types.

Example

```
IntVal:=6+4;{the result is 10.}
IntVal:=6-4;{the result is 2.}
IntVal:=6*4;{the result is 24.}
IntVal:=6/4;{the result is 2.}
```

In the example, you may have noticed that the last result seems incorrect. This is not a typo; were you to divide these numbers and return the result to a variable of type INTEGER or LONGINT, you would get this value.

This result indicates the difference between calculations involving REAL values and calculations with INTEGER or LONGINT data. While addition, subtraction, and multiplication of INTEGER or LONGINT data will always return a whole number, division will have cases where a fractional result will be returned. Since INTEGER and LONGINT data definitions do not accommodate fractions, the result will be data loss. To prevent this, VectorScript provides a workaround, always converting the result of INTEGER or LONGINT division to a REAL value. If you were to substitute a variable of type REAL, the entire result would be returned.

Example

```
RealVal:=6/4;{the result is 1.5.}
```

VectorScript assumes that you are aware of the possibility of losing data when performing INTEGER or LONGINT division. It therefore does not generate an error if the result from your calculations is assigned to a data type where rounding of the value occurs.

Suppose, however, that you want to divide two integers and retrieve both the whole and fractional parts of the result, and also preserve the data type of the results. VectorScript provides two alternate operators for division operations with INTEGER or LONGINT values: DIV and MOD.

DIV provides the same basic functionality as the “/” operator, returning the quotient (whole number) of the division. In this case, though, the quotient result is the same data type as the operands.

MOD provides the opposite result of DIV, returning the remainder, or modulus, as the result. The MOD result is also returned as a value of the same data type as the operands.

Example

```
9 DIV 4{The result is 2.}
9 MOD 4{The result is 1.}
```

The ** operator allows a real to be raised to real power

```
      2.5**2.0 {The result is 6.25}
```

Precedence of operations on the same level is left to right. Multiple instances of exponentiation, however, are performed from right to left. Parentheses can be used to force a higher precedence of operations.

Relational Expressions

Expressions can also be used to compare values as quantities. These types of expressions, called relational expressions, are used extensively in VectorScript in making decisions on how to control the execution of the script.

Relational expressions are comprised of either numeric or string operands, combined with special relational operators which define how the operands are to be compared. The result of a relational expression is always a logical (TRUE or FALSE) value.

Example - Relational Expressions

```
      j < 2 this >= that count = 5
      total < (amount+tax) objectHandle <> NIL
```

Relational Operators are:

>	Greater than
>=	Greater than or equal to
=	Equal to
<>	Not equal to
<=	Less than or equal to
<	Less than

Relational expressions can be combined with an assignment operator to store the result in a variable, but they are most often used with other statements to form a decision making structure in your script.

Example - Relational Expression

```
      PROCEDURE Test;

      VAR
         userVal:REAL;
```

```
FUNCTION GetValueInRange(userPrompt:STRING; defaultVal, minVal,
  maxVal:REAL):REAL;
{
    GetValueInRange demonstrates using relational expressions
    to guarantee a user-entered value falls within a desired range.
}
VAR
    theVal:REAL;
    done:BOOLEAN;
    prompt:STRING;
BEGIN
    done := FALSE;

    prompt := Concat(userPrompt, ' [between ',
          Num2Str(3, minVal), ' and ',
          Num2Str(3, maxVal), '] :');
    REPEAT
      theVal := RealDialog(prompt, Num2Str(3, defaultVal));
      IF theVal < minVal THEN
        Message('You entered a value less than the minimum.')
      ELSE IF theVal > maxVal THEN
        Message('You entered a value greater than the maximum.')
      ELSE
        done := TRUE;
    UNTIL done;

    GetValueInRange := theVal;
END;



BEGIN
    userVal := GetValueInRange('Enter first number', 5, 0, 10);
    Message(userVal);
    userVal := GetValueInRange('Enter second number', 180, 0, 360);
    Message(userVal);
END;
```

```
        Run(Test);
```

When comparing relational expressions and STRING data using relational operators, the ordering of the ASCII character set becomes important in determining how relational expressions will evaluate. The following rules apply when evaluating STRING data:

- String values are compared a character at a time, from left to right.
- If the ASCII value of one character is greater than the other, the corresponding string is greater than the other.
- If corresponding characters are equal, the comparison point advances to the next character.
- If the end of one string is reached, it's value is less than the other string.
- If the end of both strings is reached, the strings are equal.

Relational expressions and their results can also be combined into larger expressions, which allow you to make complex decisions and evaluate objects and documents based on multiple factors. These larger expressions, called logical expressions, are the last expression type found in VectorScript.

Logical Expressions

Logical expressions may be thought of as an extension of relational expressions. Logical expressions use relationals as their operands, and when combined with special logical operators, return a boolean valued result.

Logical expressions are used to construct more complex decision and control statements than could be accomplished by using simple relational expressions. As you become more proficient in using the VectorScript language and write more complex scripts, you will often need to control execution based on the results of several inputs.

Example - Logical Expressions

```
        (i <=5) AND (j<>2)
        (unitcost < 123.50) OR (quantity > 25)
        (value > 12) AND NOT (value = 22)
```

There are five specialized logical operators, which handle the three logical operations available in VectorScript: AND, OR, and NOT.

AND, &	And operator
OR, |	Or operator
NOT	Not operator

AND Operator

An expression using the AND operator will evaluate to TRUE if and only if both operands in the expression are TRUE. All other conditions will evaluate to FALSE.

Operand 1	**Operator**	**Operand 2**	**Result**
TRUE	**AND**	TRUE	TRUE
TRUE	**AND**	FALSE	FALSE
FALSE	**AND**	TRUE	FALSE
FALSE	**AND**	FALSE	FALSE

Example - AND Operator

```
BoolVal1:=TRUE
BoolVal2:=FALSE
BoolResult:=BoolVal1 AND BoolVal2;
```

evaluates to FALSE.

OR Operator

An expression using the OR operator will evaluate to TRUE if either operand in the expression is TRUE. Both operands must be FALSE for the expression to evaluate to FALSE.

Operand 1	**Operator**	**Operand 2**	**Result**
TRUE	**OR**	TRUE	TRUE
TRUE	**OR**	FALSE	TRUE
FALSE	**OR**	TRUE	TRUE
FALSE	**OR**	FALSE	FALSE

Example

```
BoolVal1:=TRUE
BoolVal2:=FALSE
BoolResult:=BoolVal1 OR BoolVal2;
```

evaluates to TRUE.

NOT Operator

The result of the NOT will be the opposite of it's single operand.

	Operand	**Result**
NOT	TRUE	FALSE
NOT	FALSE	TRUE

Example - NOT Operator

```
BoolVal1:=TRUE;
BoolResult:=NOT BoolVal1
```

evaluates to FALSE.

Short Circuit Operators

The two alternate operators for AND and OR operations, & and |, are also known as "short circuit" operators.

In logical expressions, since both the operands are also expressions, they must first be evaluated so that the return result may be used as input for the larger expression. When using AND and OR, both operands are always evaluated and the result of the logical expression is found.

With short circuit operators however, the first operand is evaluated and checked by the interpreter. If a definite result of the entire expression is determined, the second operand is not evaluated; instead, the expression immediately returns a result.

The time saved by using this "short-circuit" operation may seem trivial, but in complex scripts or loops which perform numerous calls to the logical expression, using these operators can result in scripts which run much faster.

Example - Short Circuit Operator

```
foo:= 3;
bar:=5;
WHILE ((foo > 4) & (bar < 6)) DO BEGIN
   SysBeep;
END;
```

In the example, the logical expression will return false and prevent the loop from executing. The first operand, foo < 4, returns FALSE(since foo is equal to 3). Once this operand has been evaluated as FALSE, the entire expression returns FALSE, since both expressions must return TRUE for an AND to return TRUE. In this instance, the expression bar < 6 never gets evaluated, saving some time in execution.

Short circuit operators also allow notational efficiency when calling fucntions which require their parameters to be validated. If you need to call a function which can not accept a NIL handler do the following:

```
IF(handleValue <>NIL &
   ProcessHandle(HANDLEValue))THEN
```

Operator Precedence

In VectorScript, operators and operands can be chained together to form complex expressions. The order in which these operations are performed can have a significant impact on the result. This is where operator precedencematters. All operators in VectorScript are assigned a level of importance, or precedence. When a complex expression is encountered, this precedence is evaluated to determine which operation should be performed first, second, and so on. In this way, uniform results are derived according to the established rules of precedence. The table below lists operator precedence in VectorScript.

	Precedence
(), **, ^, NOT	highest
*, /, DIV, MOD, AND, &	second
+, -, |, OR	third
=, <>, <, >, <=, >=	lowest

REPETITION STATEMENTS

In VectorScript, your script statements are executed in a linear fashion, one after another until the entire script has been processed. This type of execution does not lend itself to repeated execution of script statements to process multiple objects or to perform a progressive operation on the document. To accomplish this type of repeated execution, you need loop statements.

Program loops provide a means of executing a section of your script repeatedly, but still under your control. Entry into the program loop is controlled by the entry condition, which determines whether the necessary prerequisites exist to execute the loop. Once past the entry condition, the statements which you have defined as part of the loop will begin to execute. This process will repeat itself indefinitely. To stop this execution, you must check for a specific condition to indicate that the loop should terminate. This condition, known as the exit condition, can be determined by whether there are any objects left to process, whether a specific document state exists, or even by a simple numeric count.

As an example, suppose you wanted to create a script which changes the pen foreground color of all the selected objects in your document. Your script would need to contain a program loop which would process each object individually, setting it’s pen foreground color.

In your script’s program loop, the entry condition of your loop would determine if any objects are selected.

The exit condition of your loop would determine if there were any objects left to process; once all the selected objects are processed, then the loop is exited, and your script can finish.

Note: To stop VectorScript during execution, depress and hold, Command-period (Macintosh) or the Esc (Windows). In most cases, this will terminate the VectorScript script.

The following example illustrates a typical loop.

Example - Program Loops

```
PROCEDURE SetSelObjectColor;
   CONST
     kNewColor = 45;
   VAR h:HANDLE;
BEGIN
   h := FSActLayer;
   WHILE h <> NIL DO BEGIN
     SetPenFore(h, kNewColor);
     h := NextSObj(h);
```

```
      END;
   END;
   Run(SetSelObjectColor);
```

In the example, the expression (h <> NIL) acts as a gatekeeper for the larger WHILE-DO repetition statement. The result of the relational expression directly controls the continuing execution of the loop.

In VectorScript, when you step through a list of objects, NIL is returned when the end of the list is reached. Until the end is reached, there is a value other than NIL in objectHandle, and TRUE is returned by the expression to the repetition statement. When NIL is returned (the end of the list is reached), FALSE is returned, and the loop terminates.

There are three types of repetition statements in VectorScript: FOR-TO/FOR-DOWNTO statements, REPEAT-UNTIL statements, and WHILE-DO statements

FOR-TO /FOR-DOWNTO Statements

Both types of FOR structures adhere to several rules which govern the layout and function of the structure:

- The counter variable must be a variable of type INTEGER and must be declared within the procedure where it is used.
- Do not try to change the value of the counter variable from within the FOR statement; this can produce unpredictable results.
- If limit expressions are used, the counter variable should not be included in the expressions.
- The TO and DOWNTO conditions are inclusive; that is, equal values will evaluate to a TRUE condition.

For-To

The FOR structure uses a “counter” or “limit” variable to determine the number of times the specified statement or statements will be executed.

Syntax

FOR-TO structure

```
FOR  <variable identifier> = initial value TO upper limit value DO
   statement;
```

```
FOR  <variable identifier> = initial value TO upper limit value DO
BEGIN
   statement;
   statement;
   . . .
   . . .
   statement;
END;
```

Example - FOR Loop

```
Procedure Loop01;
VAR
   Count: INTEGER;
BEGIN
   FOR Count:=1 TO 10 DO
     Message(Count);
END;
Run(Loop01);
```

With a FOR loop structure, the counter variable is initialized when the loop is entered. The statement or statements in the loop structure are then executed, and execution returns to the beginning of the loop. In the example, the message bar is displayed with the value of the counter variable. The counter variable is then incremented, the limit expression is evaluated (is *Count* less than or equal to 10) and the whole process begins again. When the limit of the loop (10 in the example) is exceeded, the loop is exited and execution continues with the next statement following the repetition statement.

FOR statemens and compound statements in a FOR structure must have a BEGIN and END statement to define their bounds. More complex expressions may also be used to define the limits of execution for the FOR loop. These limit expressions may be of any type previously outlined.

FOR-DOWNTO

There may be times when you may want the count to decrement, or count down, rather than count up. The FOR structure accommodates this possibility by allowing you to replace the TO reserved word with DOWNTO.

Syntax

FOR-DOWNTO structure:

```
FOR  <variable identifier> = initial value DOWNTO lower limit value DO
   statement;
FOR  <variable identifier> = initial value DOWNTO lower limit value DO
BEGIN
   statement;
   statement;
   . . .
   . . .
   statement;
END;
```

Example - FOR-DOWNTO loop

```
Procedure Loop02;
VAR
   Count: INTEGER;
BEGIN
   FOR Count:=10 DOWNTO 1 DO
     Message(Count);
END;
Run(Loop02);
```

Other than reversing the direction of the count, the FOR-DOWNTO structure functions in the same way as a FOR structure. The following is a more complex example of a For Loop:

Example - Complex For loop

```
PROCEDURE RRExample;
   {
   RRExample is an example of using a FOR loop
   to construct a segment of railroad track with
   a very repetitive geometry.
   }
   CONST
     kTieInterval = 0.3;
   VAR i:INTEGER;
      totalTies: INTEGER;
   PROCEDURE DrawRRTie(which: INTEGER);
    CONST
      kHalfWidth = 0.1 / 2;
      kHalfLength = 0.8 / 2;
    VAR tie:REAL;
   BEGIN
    tie := which * 0.3;
    Rect(tie-kHalfWidth, -kHalfLength, tie+kHalfWidth, kHalfLength);
   END;

   PROCEDURE DrawRails(howManyTies: INTEGER);
    VAR halfRailLength:REAL;
   BEGIN
    halfRailLength := (howManyTies * kTieInterval) / 2.0;
    Rect(-halfRailLength, 0.3, halfRailLength, 0.25);
    Rect(-halfRailLength, -0.3, halfRailLength, -0.25);
   END;
BEGIN
   totalTies := 7;
   FOR i := -totalTies DIV 2 TO totalTies DIV 2 DO
    DrawRRTie(i);

   DrawRails(totalTies);
END;
```

REPEAT-UNTIL Statement

The REPEAT structure will execute the included statements until the limit expression (exit condition) returns a value of TRUE.

REPEAT and REPEAT-UNTIL statements do not require BEGIN or END keywords even when executing multiple statements. This is because the REPEAT and UNTIL keywords indicate the statement boundaries to the VectorScript interpreter.

Syntax

```
REPEAT
   statement;
   statement;
   . . .
   . . .
   statement;
UNTIL boolean  expression;
```

Example - REPEAT-UNTIL

```
Procedure RU01;
VAR
   Count,NewVal:INTEGER;
BEGIN
   Count:=0;
   REPEAT
     NewVal:=Count*2;
     Message(NewVal);
     Count:=Count+1;
   UNTIL(Count>10);
END;
Run(RU01);
```

In the example, the statements between REPEAT and UNTIL will be executed until the exit condition is encountered. The limit expression is not evaluated until after all the statements contained within are executed, so that even if the limit condition is initially FALSE, all the statements contained with in the REPEAT-UNTIL structure will execute at least once. Also, unlike the FOR-TO statement, you are responsible for both initializing and incrementing your limit (counter) variables. More complex limit expressions may also be used with the REPEAT statement for greater control over it’s execution.

The following is a more complex loop:;

```
PROCEDURE Accelerate;
  {
  Accelerate is an example of using a REPEAT loop
  to simulate the effect of gravity on a falling body.
  }
  CONST
    g = 386.088;{in/sec^2}
    kTimeInterval = 0.015;{Seconds}
  VAR yPos, t:REAL;

  PROCEDURE DrawBall(p, t:REAL);
    CONST kRadius = 0.125;
    VAR ballLabel:STRING;
  BEGIN
    Oval(-kRadius, p-kRadius, kRadius, p+kRadius);

    TextJust(1); {left justify}
    TextSize(6);
    TextOrigin(2*kRadius, p);
    ballLabel := Concat('d: ', Num2Str(3, p),
      ' in. at ', Num2Str(3, t), ' sec.');
    BeginText;ballLabel
    EndText;
  END;
  FUNCTION GetBallPosition(t:REAL):REAL;
  BEGIN
    GetBallPosition := -t*t*g/2.0;
  END;
BEGIN
  t := 0.0;
  yPos := 0.0;
  REPEAT
    DrawBall(yPos, t);
    t := t + kTimeInterval;
    yPos := GetBallPosition(t);
  UNTIL yPos < -4.5;

END;
```

WHILE-DO Statement

WHILE-DO statements contain elements of functionality from both the FOR-TO and REPEAT-UNTIL statements. In the WHILE-DO statement, like the FOR-TO, the limit expression is evaluated prior to entering the loop part of the structure. If the WHILE limit does not evaluate to TRUE initially, no statements will execute; the limit expression functions as both the entry and exit conditions for the statement. However, unlike FOR-TO, and similar to the WHILE and REPEAT statement, you are responsible for initializing and incrementing your limit variables.

Syntax

```
WHILE  boolean expression DO
   statement;
WHILE  boolean expression DO
BEGIN
   statement;
   statement;
   . . .
   . . .
   statement;
END;
```

Example - WHILE-DO Structure

```
Procedure T1;
VAR
   Count:INTEGER;
   BEGIN
     Count:=0;
     WHILE Count < 10 DO BEGIN
       Message(Count);
       Count:=Count+1;
       Wait(1);
     END;
     Message(‘All done.’);
END;
Run(T1);
```

In the example the value of Count is initialized, so that when the conditional for the WHILE-DO loop is encountered, it is TRUE. The loop is entered, and processes until Count >= 10. The loop then exits and execution continues.

Infinite Loops

An important item to consider when using REPEAT or WHILE statements is the careful choice of limit expressions. In the example above, if the greater than (>) sign were accidentally entered as a less than sign (<), the loop would not execute at all. In other cases, a subtle mistake such as this can cause an infinite loop, which never reaches an exit condition. This will often result in an application crash. Even if you do not experience problems this severe, incorrectly choosing your limit expressions can result in your loop failing to complete it's task properly. If your scripts are returning bad results, or worse, causing application crashes, you should always double check the limit expressions in your repetition statements.

Note: To stop VectorScript during execution, depress and hold, Command-period (Macintosh) or the Esc (Windows). In most cases, this will terminate the VectorScript script.

CONDITIONAL STATEMENTS

Conditional statements are used to control the "program flow" of the VectorScript script. They are used to make decisions as to what other statements need to be acted upon, and do so by evaluating a particular condition. Relational or logical expression statements provide the actual decision making ability for the conditional statement, which then controls the path of further execution. In VectorScript there is one type of conditional statement, the IF-THEN-ELSE statement.

IF-THEN-ELSE Statement

The IF-THEN-ELSE statement uses relational and/or logical expressions to determine the TRUE-FALSE validity of a condition related to the document, an object, or the script. Based on the result, the execution of the script can take one of two paths. If the expression or value is evaluated to TRUE, the statement or statements which follow the THEN branch are executed. If FALSE is returned as a result, the ELSE branch is taken.

Syntax

```
IF (NOT) logical expression THEN
   statement
```

```
      ELSE
         statement;
```

Example - IF-THEN-ELSE structure

```
      Procedure ASample;
      VAR
         IntegerVal:INTEGER;
      BEGIN
         IntegerVal:=8;
         IF(IntegerVal > 7) THEN
           Message(‘Value was greater than 7’)
         ELSE
           Message(Value was less than or equal to 7');
      END;
      Run(ASample);
```

In the example, a relational expression is used to return a TRUE or FALSE result, which is then used to determine what path execution should take. Since *IntegerVal* is equal to 8, the THEN path is taken, and the appropriate message is displayed. If you changed the value to 6, the execution would follow a different path, executing the statements found after the ELSE reserved word.

The expressions which determine the branching of execution can be as simple or complex as your needs warrant. Several expressions can be used to test various conditions, and thereby define a very explicit circumstance under which a series of statements will execute. The next example illustrates the use of such complex expressions.

IF-THEN-ELSE statements can also become more complex in terms of the statements executed in the two branches of the structure. Multiple (compound statements can be executed in each branch, and additional nested conditional or repetition can also be included in these branches.

Syntax

```
      IF (NOT) logical expression THEN
      BEGIN
         statement;
         statement;
         . . .
         . . .
         statement;
      END
      ELSE BEGIN
         statement;
         statement;
         . . .
         . . .
         statement;
      END;
```

Example - IF-THEN-ELSE with complex expression

```
      Procedure ASample;
      VAR
         IntegerVal:INTEGER;
      BEGIN
         IntegerVal:=90;
      IF((IntegerVal > 0) AND (IntegerVal < 90)) OR
      ((IntegerVal > 180) AND (IntegerVal < 270)) THEN
           Message('Tangent is positive.')
         ELSE
           Message('Tangent is negative or value out of range');
      END;
      Run(ASample);
```

You can specify a very complex set of conditions which must be satisfied for a statement to be executed. In the example, only angle values in a certain range will be considered as generating a positive tangent value.

Note the use of the parentheses to control the precedence of evaluating the different expressions. When using multiple expressions to evaluate a condition, parentheses should be used to define the boundaries of the expressions, and indicate to the VectorScript interpreter that they should be evaluated first.

In the example, two levels of parentheses were used, one to obtain operands for the AND operation, and the second to return the results of the AND operations as operands for the OR operation. The result of the OR operation determines the path to be taken in the structure.

Syntax

```
      IF (NOT) boolean expression THEN
      BEGIN
      statement;
      statement;
      . . .
      . . .
      statement;
   END
   ELSE  IF boolean expression  THEN
   BEGIN
      statement;
      statement;
      . . .
      . . .
      statement;
   END
   ELSE BEGIN
      statement;
      statement;
      . . .
      . . .
      statement;
   END;
```

Example - IF-THEN-ELSE with compound statements

```
   Procedure ASample;
   VAR
      RealVal:INTEGER;
   BEGIN
      RealVal:=0;
      RealVal:=RealDialog(‘Enter a width’,’6');
      IF REALVal< 6.0 THEN BEGIN
      RealVal:=RealVal*1.25;
       Message(RealVal);
      END
      ELSE BEGIN
       RealVal:=RealVal*1.1;
       Message(RealVal);
      END;
   END;
```

```
      Run(ASample);
```

In the example, multiple statements are specified for each branch of the IF-THEN-ELSE structure. If the value entered by the user is less than 6, the value will be multiplied by 1.25 and displayed; otherwise, the value is multiplied by 1.1 and displayed.

CASE STATEMENT

A CASE statement allows the result of a single expression to be compared to a number of constant values. VectorScript can execute a CASE statement more efficiently then a series of nested If-THEN statements, although the CASE statement is not quite as flexible.

Example - Case statement

```
      CASE integerValue OF
           1: Message('One');
           2: Message('Two');
           3: Message('Three');
           OTHERWISE Message('Unknown Number');
         END;

         {
         The case labels must be computable at script compilation time. They
         cannot depend on variables or function results. When the CASE
         statement executes,if an exact match is found between integerValue
         and one of the case labels, the statement following that label is
         executed and then execution proceeds at the statement following
         END. If an OTHERWISE clause exists and no match was found, the
         statement following OTHERWISE is executed. If there is no OTHERWISE
         and no match was found, then execution continues at the statement
         following the END.
         }

         {============================================================}

      END;
```

CONTROL STATEMENTS

Control statements are a special type of statement which allows the programmer to have direct control over the program flow. The single control statement available in VectorScript is the GOTO statement, which interrupts execution and allows you to jump to a predefined location in the script.

Syntax

```
GOTO <label identifier>
```

Where the label is a corresponding numeric identifier located in the code which indicates the destination of the jump.

Example - GOTO statement

```
PROCEDURE GOTOExample;
   LABEL
    911;
BEGIN
   IF YNDialog('Are you done?') = TRUE THEN
    GOTO 911;

   IF YNDialog('Are you sure you are not done?') = TRUE THEN
    GOTO 911;

   IF YNDialog('Are you positive?') = TRUE THEN
    GOTO 911;

   Message('You cannot be convinced.');

   911: { bail out - GOTOs are most useful in processing }
      { errors or cancellations during a long sequence }
      { of operations }
END;
Run(GOTOExample);
```

GOTO statements can be used to exercise a great deal of control over script execution. In practice however, use of GOTO statements should be carefully considered, as improper use of GOTO can result in unpredictable behavior from VectorScript scripts, and can make problems very difficult to trace and diagnose. Good program design usually eliminates the need for using GOTO statements at all in VectorScript scripts.

PROCEDURES AND FUNCTIONS

In VectorScript, procedures and functions are the basic " action" unit for performing tasks. The scripts which you create function as a unit, performing a particular task when called. On the smallest scale, the predefined VectorScript procedures and functions which you use in your code perform tasks for the VectorScript script you have defined. It is no accident that your script begins with the Procedure statement. In many ways your VectorScript script is no different than VectorScript's predefined procedures and functions. To the VectorScript interpreter, they are almost identical.

Procedures

In VectorScript, a procedure is a series of statements grouped together and called as a unit to perform a specific task. Procedures can receive input in the form of parameters, and can return data back through them as well using variable parameters.

Procedure statements have two components: the procedure name, or identifier, and it's parameters, which supply needed data to the statement.

Syntax

Procedure statement:

```
<procedure name>(param1,param2,…,paramn(VAR param1,param2,…));
```

Procedure declaration:

```
<Procedure name>;
   <definitions section>
   <declarations section>
(<subroutine declarations>)
BEGIN
   <statement section>
END;
Run(<Procedure name>);
```

The general format is :

```
Procedure name(parameter1,parameter2,…);
```

Example - Procedure statement

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

In the example, the procedure statement, when called, will draw a rectangle. The rectangle will be defined by the parameters (0,0) and (2,2), which represent two corners of the rectangle. You can also define your own procedures for use in your scripts.

Functions

A function is subroutine which is designed to specifically compute and return a value. Instead of returning a result through a variable parameter, the result is returned directly. As such, functions can be used as operands in expressions, because the function call itself represents a value. In all other ways, functions are very similar to procedures; they also must be declared, and they can accept input via a formal parameter list. Let’s take a look at how functions can be used to our advantage in VectorScript.

Example - Functions

```
Procedure CalcYthPower;
VAR
   Base,Exponent,Result:REAL;
{—subroutine to calculate x raised to y—}
Function Raise2Power(theBase,theExp:REAL) : REAL;
BEGIN
   Raise2Power:=theBase**theExp;
END;
BEGIN
   Base:=RealDialog(‘Enter base value’,’1');
   Exponent:=RealDialog(‘Enter exponent value’,’0');
   Result:=Raise2Power(Base,Exponent);
   Message(Base,’ to the ‘, Exponent,’th = ‘,Result);
END;
Run(CalcYthPower);
```

Function returns value to variable result

Functions are indeed similar to procedures. You must declare them so that the VectorScript interpreter knows what input the function will accept, and what it will return.

With the function in the example, the returned value is passed back to the main program directly. In order to do this, you must declare the type of data that will be returned, and direct the result out of the subroutine appropriately. To direct the data, instead of passing the final result to a variable, you use the name of the function. This specialized syntax is understood by the interpreter, and the value is handled accordingly. This direct return is what allows a function to be used as an operand in an expression.

Function calls can be used with expressions to condense and streamline code, resulting in faster scripts. The next example shows how to use a function in a conditional expression.

Example - Functions and conditional expressions

Function is used as operand of relational expressiion

```
Procedure CalcYthPower;
VAR
   Base,Exponent,Result:REAL;
{—subroutine to calculate x raised to y—}
Function Raise2Power(theBase,theExp:REAL) : REAL;
BEGIN
   Raise2Power:=theBase**theExp;
END;
BEGIN
   Base:=RealDialog(‘Enter base value’,’1');
   Exponent:=RealDialog('Enter exponent value','0');
   IF (Raise2Power(Base,Exponent) < 0 ) THEN
     Message('Odd exponent value')
   ELSE
     Message('Even exponent or positive base value');
END;
Run(CalcYthPower);
```

Since the function returns a value, it can be used anywhere a value would ordinarily be used. This flexibility makes it possible to create complex expressions which can make specific decisions based on the results returned by your subroutines.

Defining Subroutine Procedures and Functions

The procedures and functions you define work almost identically to the predefined ones that are part of the VectorScript language. They can perform calculations, process data, or any of the other tasks that can be performed by the packaged versions. Subroutines cannot be run on their own, but will perform part of the overall work of the main script.

Let’s look at a simple example of a subroutine procedure to learn more about how they are defined and used.

Example - Subroutine

```
Procedure SwapTwoNums;
VAR
   IntValue1,IntValue2:INTEGER;
{–the subroutine–}
Procedure PerformSwap;
VAR
   Temp:INTEGER;
BEGIN
   Temp:=IntValue1;
   IntValue1:=IntValue2;
   IntValue2:=Temp;
END;
{————————}
BEGIN
   IntValue1:=IntDialog("Enter first value','0');
   IntValue2:=IntDialog('Enter second value',’0');
   PerformSwap;
   Message('First value is now :',IntValue1);
   Wait(1);
   Message('Second value is now :',IntValue2);
   Wait(1);
   ClrMessage;
END;
Run(SwapTwoNums);
```

Subroutine call

In the example, we take input from the user in the form of two integer values. The subroutine we have defined is used to swap the numbers, which are then displayed.

Defining your subroutine is almost the same as defining a script. When defining the subroutine, you use the same basic structure defining the statement and (optionally) declaration parts, and you begin the definition with the Procedure statement. The one major difference is the absence of the Run statement; this statement is only used at the end of a full fledged VectorScript script, to tell the VectorScript interpreter to execute the script in VectorWorks.

Note the position of the subroutine in the script structure; subroutines must always be defined before the beginning of the main part of the script. If they are not defined in this way, the subroutines will not be available to your script.

The subroutine in the example used the integer variables directly from the main part of the script by accessing global variables. You can also pass them as parameters to the subroutine. This method has advantages over direct reference; we will explore these a little bit later. For now, let’s see how input parameters are defined for subroutines.

Example - Subroutines with Input Parameters

```
Procedure SwapTwoNums;
VAR
   IntValue1,IntValue2:INTEGER;
{–the subroutine–}
Procedure PerformSwap(theFirst,theSecond:INTEGER);
VAR
   Temp:INTEGER;
BEGIN
Temp:=theFirst;
   theFirst:=theSecond;
   theSecond:=Temp;
END;
BEGIN
   IntValue1:=IntDialog('Enter first value','0');
   IntValue2:=IntDialog('Enter second value','0');
   PerformSwap(IntValue1,IntValue2);
   Message('First value is now :',IntValue1);
   Wait(1);
   Message('Second value is now :',IntValue2);
   Wait(1);
   ClrMessage;
END;
Run(SwapTwoNums);
```

Subroutine call

The VectorScript interpreter treats your subroutine procedures and functions just like the predefined ones; consequently, you have to tell the interpreter what to expect as input when using your procedures to perform tasks. In the example, we passed the two integer values as parameters to our subroutine. The interpreter knows from the declaration to expect that two integer values will be passed into the positions we defined, which are also known as the formal parameters of the subroutine. The variables were then used in the subroutine and the swap was performed. The variable used to temporarily store one of the integer values is defined as part of the subroutine, and is known as an actual parameter.

The advantage in using parameters is that we could easily pass two other variables to the subroutine to be swapped; we could then use the subroutine as many times as we wished in our main program. This modular approach can result in programs which are easier to read and understand, and which are smaller and which run faster.

Parameter passing has another distinct advantage; it also allows for passing data back from the subroutine to your main script. This two-way data exchange feature is the key to the modularity of subroutines.

To pass data back from your subroutine, you use what are known as variable parameters, which are defined in the subroutine's formal parameter list with the reserved word VAR. You can then use the variable parameter in your subroutine calculations, and when the subroutine completes, the value is passed into whatever variable you specify in the call in your main script.

Example - Passing data using variable parameters

```
Procedure CalcYthPower;
VAR
   Base,Exponent,Result:REAL;
{—subroutine to calculate x raised to y—}
Procedure Raise2Power(theBase,theExp:REAL;VAR Value:REAL);
BEGIN
   Value:=theBase**theExp;
END;
BEGIN
   Base:=RealDialog('Enter base value','1');
   Exponent:=RealDialog('Enter exponent value','0');
   Raise2Power(Base,Exponent,Result);
   Message(Base,' to the ', Exponent,'th = ',Result);
END;
Run(CalcYthPower);
```

Function accepts values as input

In the example, the input values are passed as the first two values in the parameter list of the subroutine, and the result is returned into the variable occupying the third position in the list.

The use of variable parameters makes it very simple to exchange data between your main program and your subroutines, doing so in a way that makes it very easy to understand and which makes your subroutines easy to reuse without modification.

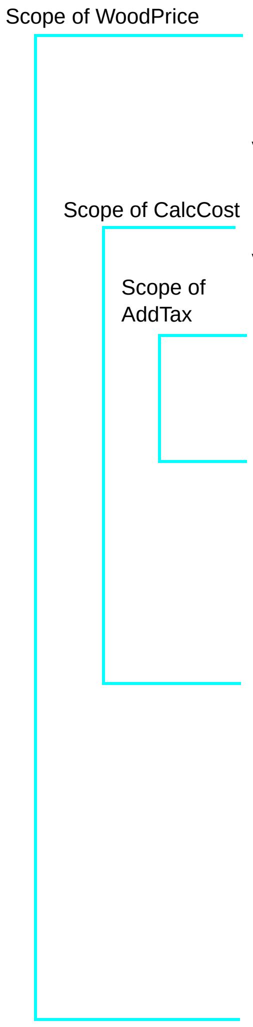
PROGRAM SCOPE

As you begin to use subroutines in your scripts, it becomes important that you understand the concept of program scope.

Program scope describes the “realm”, or area, where the variable is considered defined and valid, and may be used to represent a value or action. The scope of a variable can be defined as a “program block” of the program, or an area comprising one definition, one declaration, and one statement part. A subroutine, whether it is a procedure or a function, is one block; the main script itself is also a block. If a variable is defined within a block (declared at the beginning of the block), it’s scope is said to be that block, plus any blocks that may be nested inside. Variables declared as part of the main script are said to be “global” in scope; that is, they are valid everywhere throughout the program.

This concept has important implications for accessing data. If a variable is declared as an actual parameter of one of your subroutines, you cannot access it from your main program. It may, however, be used by your subroutine, and any subroutines nested within. The next example illustrates this concept.

Example - Program Scope



```
Procedure WoodPrice;
CONST
   Tax=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*Tax);
END;
{-begin CalcCost -}
BEGIN
   baseCost:=Feet*PPF;
   Cost:=AddTax(baseCost);
END;
{–end CalcCost–}
{– begin main–}
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–}
Run(WoodPrice);
```

The example has three blocks, or areas of scope: the main procedure(WoodPrice), the subroutine(CalcCost), and the function(AddTax). In the example, the total cost of the wood is returned via variable parameter to the main program for output. If we tried to reference the value directly, by replacing TotalCost in the message procedure with Cost, we would get an error. This is because in the main program, Cost is undefined; it's scope does not extend to the main program.

On the other hand, if we wanted to access the board feet directly, we could, by replacing Feet with BoardFeet. Because BoardFeet is defined in the main program, it can be referenced in any subroutine; in other words, it's scope is global.

Another example of this is the use of the constant Tax. Because it is declared in the main program, we can access it directly in the function AddTax. This makes adjusting the tax rate very simple, as we only need change one value and our entire script will still work properly.

Program scope applies to the all identifiers (constants, variables, and subroutines) within VectorScript. The figure below shows another way of illustrating the concept of program scope. Scope may be alternately thought of as a boundary around a particular program block; the identifier is valid anywhere within the boundary, and invalid outside of it.

Figure - Program Scope

Identifiers (internal subroutines, variables, and constants) defined in each of the blocks shown in the figure would then have scope as follows:

	Block scope
main routine 'A'	A,B,C,D
subroutine 'B'	B,C,D
subroutine 'C'	C,D
subroutine 'D'	D

RECURSION

Past versions of MiniPascal have not supported recursively called functions. VectorScript fully supports recursion. Be aware that while VectorScript attempts to gracefully handle infinitely recursive code, it can cause VectorWorks to crash.

Example - Recursion

```
FUNCTION CountObjects:LONGINT;
   {
   CountObjects is an example of using recursion
   to traverse a VectorWorks drawing which contains hierarchical
   groups.
   }
```

```
VAR count:LONGINT;
 root:HANDLE;
 levels:LONGINT;
PROCEDURE WriteNodeInfo(level:LONGINT; h:HANDLE);
 VAR i:LONGINT;
  objType:INTEGER;
BEGIN
 FOR i := 1 TO level DO
  Write(' ');
 objType := GetType(h);
 CASE objType OF
  2:   Write('Line');
  3:   Write('Rect');
  4:   Write('Oval');
  5:   Write('Polygon');
  6:   Write('Arc');
  10:  Write('Text');
  11:  Write('Group');
  OTHERWISE Write('Unknown Type');
 END;
 WriteLn(' (', objType, ')');
END;

{
The CountList subroutine is called for the root layer. It
loops through each object in the layer and processes it. If
The object is a group, CountList recursively calls itself
to process all objects in the group. Because VectorWorks
lists are guaranteed to be finite, this will never cause
infinite recursion.
}

PROCEDURE CountList(node:HANDLE);
 VAR child:HANDLE;
BEGIN
 WHILE node <> NIL DO BEGIN
  WriteNodeInfo(levels, node);
```

```
            child := FInGroup(node);
            IF (child <> nil) THEN BEGIN
              {node has children, so we recursively count them}
              levels := levels + 1;
              CountList(child);
              levels := levels - 1;
            END;
            count := count + 1;
            node := NextObj(node);
          END;
        END;



      BEGIN
         levels := 0;
         count := 0;
         root := FObject;
         CountList(root);
         CountObjects := count;
      END;
```

This is the recursion output:

```
      Group  (11)
       Group  (11)
        Rect  (3)
        Line  (2)
        Line  (2)
        Text  (10)
        Text  (10)
       Rect  (3)
       Rect  (3)
      Group  (11)
       Group  (11)
        Rect  (3)
        Line  (2)
        Line  (2)
        Text  (10)
```

```
         Text  (10)
        Polygon  (5)
```

ARRAYS

Array variables are a method of storing related data under a single variable reference, allowing a large amount of information to be referenced from a single item. Arrays store their information in locations that are contiguous, that is, one right after the other, making it possible to sort or methodically process large amounts of data.

Arrays are declared in VectorScript as follows:

```
<array name> : ARRAY[1..n] OF <data type>;
```

where n is between 2 and 32767. The values within the brackets represent the size, or bounds, of the array, which is the number of places that may hold data.

To retrieve a value from an array requires two components: the name of the array and the location, or array index, of the data which we want to retrieve. Array definition and use is shown in the example below.

Example - Arrays

```
Procedure ArrayExample;
VAR
   i :INTEGER;
   Words : ARRAY[1..7] OF STRING;
   Result:STRING;
BEGIN
   Words[1]:='This ';
   Words[2]:='is ';
   Words[3]:='an ';
   Words[4]:='example ';
   Words[5]:='of ';
   Words[6]:='array ';
   Words[7]:='usage.';
   i:=1;
   WHILE i < 8 DO BEGIN
     Result:=Concat(Result,Words[i]);
```

```
         i:=i+1;
      END;
      Message(Result);
   END;
   Run(ArrayExample);
```

The example shows how easily data can be accessed by moving along, or traversing, the array. While the example shows how the array is traversed for output, the same type of traversal can be adapted for input as well.

Arrays can also be two dimensional, with the data values forming a grid pattern rather than a linear order. The format for declaring two dimensional arrays is

```
   <array name> : ARRAY[1..m,1..n] OF <data type>;
```

The corresponding call to retrieve a data value would then require two index values to identify the data's position within the array.

Arrays of three or more dimensions are not allowed in VectorScript. Arrays are powerful tools for implementing search and sort routines in VectorScript. These types of tools extend the data processing capabilities of VectorWorks far beyond that of most other CAD packages.

VECTORS

Vectors in VectorScript are specialized data types used to provide vector calculation capabilities within the language. Vector quantities are an important tool in the physical sciences, as well as in mechanical and other types of design. While VectorScript does not provide sophisticated tools for analyzing vectors, all basic vector operations are supported in the language. In VectorScript, a vector variable is declared as follows:

```
   <variable name> : VECTOR;
```

A vector stores three values of type REAL. Each value represents a location along the x, y or z axis. Values within the vector may be stored or retrieved through the use of an index in the range 1 to 3.

Example - Vector value retrieval

```
   PROCEDURE Test;
   VAR
      aVector : VECTOR;
   BEGIN
```

```
        aVector[1] := 1;
        aVector[2] := 1;
        aVector[3] := 0;
        Message(‘Vector - X: ’,aVector[1],’, Y: ’,aVector[2],’, Z: 
        ’,aVector[3]);
    END;
    RUN(Test);
```

The following vector operations are supported in VectorScript (‘v’ and ‘w’ are vectors and ‘k’ is a nonzero real number):

Negative	-v
Addition	v + w
Subtraction	v - w
Multiplication with a scalar	k * v
Division by a scalar	v / k
Dot Product	v • w
Cross Product	v * w

While vectors may appear to be similar to any other basic data type, because they hold several values and are similar to arrays, they cannot be passed as parameters to or returned from user defined subroutines.

SEARCH CRITERIA

VectorScript provides a number of procedures which allow the user to make use of the attributes of objects as a method of selecting them, as well as retrieving information from them. These procedures are known as inquiry, or search, routines.

Each graphic object within the document has attributes which can be read to identify the object. Some of these attributes include layer, class, linestyle, and object type. Inquiry routines make use of a user specified search criteria to go through the list of objects in the document and find any objects which match the specified criteria.

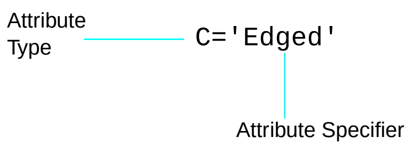
The example below illustrates a simple inquiry operation, counting all the objects in a particular class. In more complex documents this type of procedure can be extremely useful for retrieving all sorts of information.

Example - Inquiry procedure

```
Procedure CountParts;
VAR
   NumParts:LONGINT;
BEGIN
   NumParts:=Count((C='Plumbing Fixtures'));
   Message(NumParts,' fixtures were found');
END;
Run(CountParts);
```

In the example, note that the inquiry procedure's parameter. This specially formatted parameter is the search criteria. Each search criteria is comprised of two parts: the attribute type and the attribute specifier.

Example - Search criteria



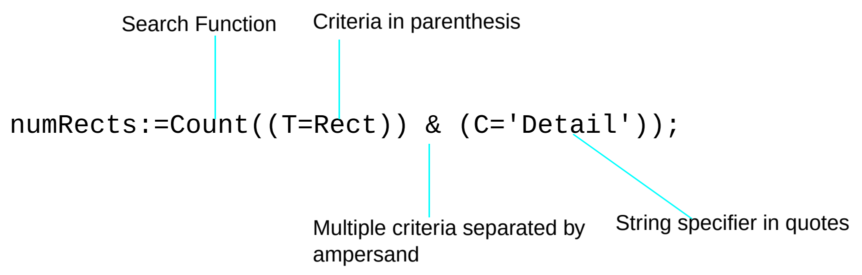
The first part, the attribute type, identifies what type of attribute is to be searched across for a match. Attribute types can include layers, classes, pen patterns, selection status, or many others. This part of the criteria 'narrows' the search, allowing for quick processing of objects. The second part, the attribute specifier, is the value to be matched, also known as the target value. The inquiry routine compares this value against the value assigned to the object, and performs the appropriate action if a match has occurred. In the case of the above example, when a match to the criteria is found, the running count of objects is incremented by 1. When the end of the object list is reached, the procedure returns a total count of objects matching the criteria.

Syntax

All search criteria conform to a few simple rules of syntax. These rules make it possible for VectorWorks to quickly search the object list by using an internal search function which uses a uniform set of rules for it's search.

- All individual search criteria must be enclosed in parentheses, even when specifying only one criteria. These parentheses are in addition to the normal parentheses which contain the parameters for the function or procedure.
- All text string attribute specifiers must be enclosed in quotes. This includes layer names, classes, etc. A good rule of thumb is if the specifier is something that can be modified by an editable text field in VectorWorks (layer names, for example), then it should be enclosed in quotes.

Example - Syntax



```
numRects:=Count((T=Rect)) & (C='Detail'));
```

Search Function
Criteria in parenthesis
Multiple criteria separated by ampersand
String specifier in quotes

Narrowing the Search

Search criteria can be combined to further narrow and define an inquiry. This allows for more flexibility in creating specific searches to find small sets or single objects, and is known as compound search criteria.

When specifying compound search criteria, each criteria is formatted according to normal rules of syntax, then separated by an '&' symbol or the reserved word AND. This delimiter includes the different criteria as part of a single search inquiry. For example, suppose a facilities manager had a document which contained polygons representing room areas for each room on every floor of an office building If it was necessary to derive a sum total of all the room areas on the second floor of the building, a single search criteria could be constructed to perform this operation. The example below shows the inquiry function call.

Example

```
Procedure AreaSecond;
```

```
      VAR
         TotArea:REAL;
      BEGIN
         TotArea:=Area((C='Room Areas') AND (L='Second Floor'));
         Message('Area total for 2nd floor : ',TotArea);
      END;
      Run(AreaSecond);
```

The compound criteria narrows the search, thereby allowing specific information to be extracted from the document.

Multiple Search Criteria

Search criteria can also be specified to search across multiple attribute specifiers for a single attribute type. In this way, subsets of the total document can be searched, providing another method of narrowing the overall search. This method of searching is known as multiple search criteria.

Multiple search criteria have a special formatting which allows VectorWorks' search function to process through the attribute specifiers as a list. In technical terms, the attribute specifiers are passed as a parameter list to the search function. The multiple criteria is specified as follows:

```
      (attrib type IN [ attrib specifier,attrib specifier,…])
```

As an example, suppose it was necessary to count all symbols named 'Part 2400' and 'Part 5230' in the document. This could be accomplished using two separate inquiries and adding the results, or could be performed using a multiple search criteria. The example below shows the multiple search inquiry.

Example - Multiple search criteria

```
      Procedure CountEmUp;
      VAR
         Total:LONGINT;
      BEGIN
         Total:=Count((S IN ['Part 2400','Part 5230']));
         Message('Total count was ',Total);
      END;
      Run(CountEmUp);
```

Even more complex searches can be created by combining the two methods just described. Suppose it were necessary to format a drawing for export to DXF to ensure a good import into AutoCAD®, and that all objects with a pen pattern of -5 on layers ‘New Construction’ and ‘Proposed Site Mods’ had to be selected for further processing. An inquiry could be constructed to specifically select these objects. The inquiry for this is listed below.

Example - Complex search inquiry

```
Procedure ProcessSelect;
BEGIN
   SelectObj((PP=5) AND (L IN [‘New Construction’,’Proposed Site
  Mods’]));
END;
Run(ProcessSelect);
```

Records and Fields

Another way of using inquiry routines involves using records and fields. Using records and fields as attribute specifiers adds significant power to the inquiry routines, since the data within records can be used as a basis for searches.

The syntax for searching by record is very similar to the syntax for multiple criteria, except that the name of the record in quotes is used as the specifier. An example of a record name as attribute specifier is shown below.

Example

```
Procedure NumRex;
VAR
   RecCount:LONGINT;
BEGIN
   RecCount:=Count(R IN [‘Part Data’]);
   Message(‘Total Recs :’,RecCount);
END;
Run(NumRex);
```

This example would count all the ‘Part Data’ records attached to objects in the document.

Fields can also be used to create inquiries. The example below shows the format for a record-field attribute specifier.

Example - Record Field attribute specifier

```
Procedure NumRex;
VAR
```

```
         RecCount:LONGINT;
      BEGIN
         RecCount:=Count((‘Part Data’.’Cost’));
         Message(‘Total Recs :’,RecCount);
      END;
      Run(NumRex);
```

If this example seems a bit odd to you, it should. There is no inherent advantage to using a record field specifier to perform a count of records, since each ‘Part Info’ record will have a ‘Cost’ field. In the next section, though the advantages of record field specifiers will become apparent.

Operators in Attribute Specifiers

Inquiry routines in VectorScript support one additional feature which adds yet more flexibility and power to their abilities. Record field attribute specifiers support the use of relational operators to compare field data to a specific value. When used with these operators, this method of performing inquiries takes on new importance. For example, suppose it was necessary to select all fixtures less than a certain price in order to update their pricing. An inquiry could be constructed which would perform just such an operation. The following example illustrates this inquiry.

Example - Rec Field Search with Operators

```
      Procedure ChoosePart;
      BEGIN
         SelectObj((‘Part Data’.’Cost’ < 100.00));
      END;
      Run(ChoosePart);
```

Using this inquiry, only objects whose ‘Cost’ field has a value less than 100.00 will be selected. The power of this is evident, since it allows objects to be matched according to very specific criteria. This method can also be combined with the compound and multiple search criteria methods to producer inquiries which match specific objects or sets of objects.

Other Search Criteria

In addition, there are a couple of other criteria which add some additional flexibility to inquiry routines.

Select All Objects

This search criteria can be used to process every object in the document, regardless of visibility, layer, or lock status.

Example

```
      Count(All);
```

counts all objects.

Visibility Status

Visibility status can be used as a criteria as well. When this criteria is used, the inquiry routine will search all visible or invisible objects.

Example

```
      Count((V=False));
```

counts all invisible objects.

Selection Status

Selection status may be used to further narrow searches, or as a criteria by itself. The criteria will specify either all selected or all deselected objects.

Example

```
      Count((SEL=False));
```

counts all deselected objects.

```
      Count((L='First Floor') AND (SEL=True));
```

counts the slected objects on layer ‘First Floor’.

Search Criteria Parameters

Attribute Type Identifiers

	Attribute Type Identifier	Example
Arrowhead	AR	(AR=1)
Class Name	C	(C='Tile')
Every Object	All	All
Fill Background	FB	(FB=23)
Fill Foreground	FF	(FF=42)
Fill Pattern	FP	(FP=3)
Layer Location	L	(L='Basement')
LineWeight	LW	(LW=2)
LineStyle	PP	(PP=2)
Object Name	N	(N='Brick')
Object Record	R	(R IN['Doors'])
Object Type	T	(T=Rect)
Pen Background	PB	(PB=Black)
Pen Foreground	PF	(PF=Black)
Pen Pattern	PP	(PP=1)
Selected Status	Sel	(Sel=True)
Symbol Name	S	(S='Window')
Visibility	V	(V=True)
InSymbol	InSymbol	InSymbol

Attribute Field Identifiers

Attribute Field	Attribute Field Identifier	Example
Object Name	String of 20 or less characters	(N='Brick')
Class Name	String of 20 or less characters	(C='Tile')
Layer Location	String of 20 or less characters	(L='Basement')

Attribute Field	Attribute Field Identifier	Example
Fill Pattern	FP followed by fill pattern number	(FP=3)
LineWeight	LW followed by line weight number	(LW=2)
LineStyle	PP followed by pen pattern number	(PP=2)
Object Type:		
Rectangle	Rect	(T=Rect)
Oval	Oval	(T=Oval)
Polygon	Poly	(T=Poly)
Polyline	Polyline	(T=Polyline)
Arc	Arc	(T=Arc)
Quarter Arc	QArc	(T=QArc)
Line	Line	(T=Line)
Text	Text	(T=Text)
Rounded Rectangle	RRect	(T=RRect)
2D Locus	Locus	(T=Locus)
3D Locus	Locus3D	(T=Locus3D)
Free Hand Line	FHand	(T=FHand)
Dimension	Dimension	(T=Dimension)
Symbol	Symbol	(T=Symbol)
Sweep	Sweep	(T=Sweep)
Mult. Extrude	MXtrd	(T=MXtrd)
Worksheet	SprdSheet	(T=SprdSheet)
Group	Group	(T=Group)
Mesh	Mesh	(T=Mesh)
Extrude	Xtrd	(T=Xtrd)
Roof / Floor	Slab	(T=Slab)
Wall	Wall	(T=Wall)
Layer Link	LayerLink	(T=LayerLink)
Poly3D	Poly3D	(T=Poly3D)
PICT	Pict	(T=Pict)
Bitmap	Bitmap	(T=Bitmap)
Light	Light	(T=Light)

Attribute Field	Attribute Field Identifier	Example
Symbol Name	String of 20 or less characters	(S=Window)
Selected Status		(Sel = True)
Selected Status		(Sel = False)