VectorScript Language Guide

Welcome to the VectorScript Language Guide.

VectorScript is the scripting language component of VectorWorks. Similar to Pascal, VectorScript is actually a "superset" of the Pascal language, extending the capabilities of that language with a broad range of features which access the power and flexibility of the VectorWorks engine.

To navigate to the topic you are interested in, either select it from the table of contents on the left, or use the Acrobat Find and Search features. A comprehensive index is also included, which can be accessed by clicking Index from the table of contents.

VectorScript Language Guide

Registration & Updates

The VectorWorks disks are warranted subject to the conditions of the License Agreement for a period of six (6) months from the date of purchase by the end user. A completed Registration Card must be returned to Nemetschek N.A., Inc. to officially register your copy of VectorWorks.

Only registered users are entitled to technical support, the Nemetschek N.A. newsletter, maintenance releases, and reduced cost upgrades.

Defective master disks are replaced free of charge to the end user for six (6) months after purchase. Thereafter, master disks will be replaced for a nominal service fee set by Nemetschek N.A., Inc.

Nemetschek N.A., Inc. will make available from time to time upgrades to the purchased program for nominal charges. Such upgrades, along with the original master copy of the program, shall be considered one program, subject in its entirety to the License Agreement.

License Agreement

Nemetschek North America, Inc., hereafter referred to as NNA, grants the buyer a non-exclusive license to use the software in the package according to the terms set forth below. The software is protected by copyright laws and international copyright treaties, as well as by other intellectual property laws and treaties.

This program has been purchased for a single, specific operating system per serial number. It is licensed for installation on one machine for each serial number.

The Buyer May:

Operate this software on one computer at a time. Make one back-up copy of the software, which is automatically subject to this agreement. Modify VectorScript routines provided with this software.

The Buyer May Not:

1: Make this software available to any person or entity other than employees who must use this software as specified above.

2: Modify or merge the software with another program, except for personal use as described above.

3: Disassemble, decompile, reverse engineer, or attempt in any fashion to discover the source code of the software.

4: Sub-license, sell, lend, rent, or lease any portion of the software. The buyer may, after notifying NNA, permanently transfer all (but no portion thereof) of the software to another person or entity, who in turn is subject to this agreement.

5: Operate the software on more than one computer at a time. (Site licenses are available from NNA for multi-station use. All sites in a site license must be used at one location.) NOTE: This software involves valuable proprietary rights of NNA and others. There is no transfer to the buyer of any title to, or ownership of, this software; nor is there transfer of any patent, copyright, trade secret, trade name, trademark, or other proprietary rights related to the software. The buyer may not violate these rights and must take appropriate steps to protect NNA's rights. NNA may at any time replace, modify, alter, improve, enhance, or change the software. The license and the buyer's right to use the software terminate automatically if the buyer violates any part of this agreement. In the event of termination of buyer's right to use the software, all copies of the software must be destroyed or immediately returned to NNA.

Upgrades

All upgrades or sequential versions of the program obtained under upgrade agreements or offered at a later date in consideration of this purchase will be considered one program under this license agreement. Under no circumstances will the providing of upgrades be considered as permission for this program to reside on more than one computer at any time, nor may the buyer sub-license, sell, lend, rent, or lease any portion of former versions of the software. Again, the license and the buyer's right to use the software terminate automatically if the buyer violates any part of this agreement. In the event of termination of buyer's right to use the software, all copies of the software must be destroyed or immediately returned to NNA.

General

NNA is not responsible for maintaining the software or for helping the buyer in the use of said software except through the Registered User Support Service. This agreement constitutes the entire agreement and supersedes any prior agreement between NNA and the buyer.

In case of differences between the license agreement in this manual and the license agreement in the software, the license agreement in the software applies.

Student and Educational Sales:

Student and Education copies are sold under certain restrictions set at the time of sale. The user agrees to abide by the restrictions set forth for these versions.

Table of Contents

Introduction to VectorScript

In this Chapter:

- Some Background On VectorScript

VectorScript is the scripting language component of the VectorWorks 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 chapter provides a brief overview of the VectorScript language; it explains what VectorScript can do and what it can't, and provides information on features new to this version of the 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. The core VectorScript language continues to be developed by Nemetschek North America, in parallel with the development of the VectorWorks product.

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 document. 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 documents. These interfaces allow you to retrieve and set geometric attributes of the document such as 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 document 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 documents.

What VectorScript Can't Do

VectorScript has an impressive range of capabilities; however, they are mostly confined to the scope of VectorWorks and VectorWorks documents. 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 documents or outside of a VectorWorks document 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;
```

Identifies the script to the VectorScript compiler

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

BEGIN
    myMessage:='VectorScript';

    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
    myMessage : STRING;

BEGIN
    myMessage:='VectorScript';
    Message(kGREETING,myMessage);
    Wait(5);
    SysBeep;
    ClrMessage;
END;
Run(FirstExample);
```

Declares data storage for the script

The source code of the script

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 of this manual will explain the various parts of a script and their underlying concepts in greater detail.

New Features in VectorScript 10

VectorScript in VectorWorks 10 contains a number of significant new features and changes. The chart below lists what is new to the language in this release, and where it can be found in this guide:

Name	Purpose	Location
New data types	Improved readability and maintainability of scripts	"VectorScript Data Types" on page 3-4
New syntax for comments	Allows block comment of code that already contains {} comments	"Comments" on page 2-2
New layout manager for custom dialogs	Improved ease for layout of dialog box controls	"Custom Dialogs" on page 9-2
New custom dialog controls	New controls available include: multi-line text, multi-column list, gradient slider, and image preview popup	Chapter 9
Plug-in objects support dash and font styles	Improved user interface for plug-ins	Chapter 10
Plug-in objects support disabling or hiding of parameters	Improved user interface for plug-ins	"Setting Parameter Visibilty" on page 10-14
International Plug-in objects	Improved portability of drawings that contain plug-in objects. When opening a drawing that was created by a different localized version of VectorWorks, the plug-in objects are recognized correctly	

Name	Purpose	Location
Many small changes	Many small changes to improve performance and consistency of VectorScript	Not applicable
New functions	Many new function have been added to VectorScript	VectorScript Function Reference
New preference selectors	Various new preference selectors have been added to support DXF translation, gradient and image fills, and other features.	Appendix F

Using the Rest of this Manual

The remainder of the VectorScript Language Guide is divided into four parts. Part 1, which immediately follows this chapter, is devoted to documenting the basic syntax and structure of the VectorScript language. Chapters 2 through 6 make for bland but necessary reading, as they cover topics necessary when learning any new programming language. Chapters 7 through 9 cover more advanced language topics, and are important to understanding how to design and implement more complex scripts.

Part 2 of this guide documents VectorScript plug-in technology. This technology, which became available in VectorWorks 8, allows you to create parametrically defined objects which can be used and edited like built-in VectorWorks object types. You can also use VectorScript plug-ins to define tool items and tools created with VectorScript, which can be integrated into your workspace and used like any other command or tool. Chapters 10 and 11 cover the fundamentals of creating plug-ins, providing the information needed to use plug-in technology effectively. Chapters 12 through 16 provide more detailed information on each of the plug-in types, illustrating the basics of using each type.

Part 3 of the guide documents the development tools provided with VectorWorks which are targeted at working with the VectorScript language. These include practical techniques to use with any type of plug-in; these techniques address issues that will come up in everyday use as you become more proficient at using plug-ins in your work. Chapter 17 discusses the various script types and how to work with them inside of VectorWorks. The features and use of the VectorScript editor are discussed; this is the primary means for editing your scripts. This includes a discussion on the VectorScript debugger, an extremely useful tool for detecting and fixing errors in your scripts, and the Plug-in editor, which lets you easily create and edit the basic settings of your VectorScript plug-ins.

Finally, the appendices in Part 4 is a reference section on various topics about VectorScript. These topics address specific issues or important information commonly needed by most VectorScript users.

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, 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 document and export it using the Export VectorScript option. Once you have exported the document, use a text editor to open the document. What you will see is a VectorScript representation of the complete VectorWorks document: objects, layers, classes, document settings, and so on. You can compare this script code to the source document 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 document to see how your changes affect the document. 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 document. 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 document 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

In this Chapter:

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 chapter explains the basic lexical structure of VectorScript.

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).

Integer Literals

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.

Floating-point Literals

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 A-1 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.

String Literals

```
'VectorScript'          'Nemetschek North America'
'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.

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

VectorScript Keywords

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

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.

Other Keywords

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.

VectorScript Special Symbols

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

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 later in this guide.

Variables, Constants, and Data Types

In this Chapter:

- Variables
- Constants
- VectorScript Data Types

Chapter 2 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 chapter explains how to use variables and constants in your scripts, and provides detailed information on the various data types available in VectorScript.

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 can be used to define constants in scripts.

Functions Supported in the Constant Definition Block

Abs()	Sqr()	Sqrt()	Ord()	Chr()
Trunc()	Round()	Sin()	Cos()	Tan()
Asin()	Acos()	Atan()	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 8-1).

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

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 `-32767` 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 -2,147,183,647 to 2,147,183,647.

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 1.9 x 10e-4951 to 1.1 x 10e4932.

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 2-3 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 4-7.

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`, `HANDLE`, and `VECTOR`.

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 2-5, 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.

Arrays in VectorScript

In this Chapter:

- Static Arrays
- Dynamic Arrays

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 chapter will explain the syntax and conventions for using arrays in your scripts.

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 5-1 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 5-1 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

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 7-7 for details on these statement types).

Vectors and Array Notation

As mentioned earlier in this chapter, 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 STRINGs in VectorScript.

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 table below lists all VectorScript API functions with CHAR array support.

VectorScript Functions with CHAR Array Support

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

Structures

In this Chapter:

- Creating Structures
- Accessing Values in a Structure

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.

Note: *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

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

In this Chapter:

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

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.

Simple Expressions

`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.

Comparison Operators

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.

Logical Operators

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 3-1, 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 4-1, 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

In this Chapter:

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

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 8-1 and “User-Defined Functions” on page 8-4.

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 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 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 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

In this Chapter:

In addition to the over 700 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 chapter 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 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 chapter.

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 1-3, 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 your 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.

User Interface

In this Chapter:

VectorWorks provides several ways for a script to present a user interface to display or gather information from the user. These include Help Tags, Tool Tips, Messages, Predefined Alerts, and Custom Dialogs. This chapter briefly introduces these features, and then describes Custom Dialogs in detail.

The simplest user interface feature that VectorScript plug-ins should support is Help Tags or Tool Tips. This feature simply identifies the plug-in by name (and an additional short description) when the user hovers the cursor over a tool icon or a menu item. Plug-ins are discussed in detail in "Using VectorScript Plug-ins" on page 10-1.

The "VectorScript Message" palette is another simple user interface feature. A script can call the "`Message()`" function to display one line of information to the user. The function takes multiple arguments, and will concatenate the pieces together. This feature can be used for status or progress information. Since it is a palette, not an alert, it does not interrupt the user's workflow.

Predefined Alerts

To notify the user of an error condition, provide a warning, or ask for confirmation, a script can use one of the several predefined alerts. With one function call the script can easily present a modal alert dialog which requires the user's attention before he or she can continue. For example:

```
AlrtDialog('You must select an object first.');
```

Another predefined alert will display a string which is typically a question, and provide "Yes" and "No" buttons:

```
response := YNDialog('Do you wish to continue?')
```

There are several functions that allow the user to enter values. For example, the function `StrDialog` allows the user to enter a string and the function `PtDialog` allows the user to enter a point value. See the VectorScript Function Reference for a complete list of these predefined alert functions.

Custom Dialogs

VectorScript provides the custom dialog API for scripts whose interface needs may exceed what is provided by the predefined dialogs available in the language. Scripts may create dialogs using any combination of controls (up to 512 controls per dialog) in layouts that can be tailored to meet your specific interface needs. VectorScript allows up to 32 dialogs per script, which create sophisticated interfaces for menu commands and tools. All the components required to build and manage dialogs for handling complex data entry and user interaction are provided.

Topics discussed in this chapter include the dialog control components, dialog definition and layout, as well as handling user interaction. The chapter also addresses the use of external resource files for storing image and string data and how to use them in creating custom dialogs.

Note: *This manual mainly discusses the custom dialog system that was introduced with VectorWorks 8.x. It is sometimes referred to as the "Modern Dialog" system or the "Layout Manager" dialog system. For a limited time, VectorWorks will continue to support existing scripts which may use the previous dialog system. These functions are referred to as "Classic Dialogs" in the VectorScript Function Reference.*

Custom Dialog Concepts

Dialog box interfaces are a means of retrieving information from the user for use by the script during execution. In order to do this, dialogs need to be able accept data entry (in various formats) and provide meaningful interaction and feedback for the user. Using VectorWorks, you have probably encountered dialogs whose interface is tailored to a specific task (such as creating a layer or setting document scale) and which provide feedback based on the data you have entered. These dialogs use the same underlying concepts that you will be using in creating custom dialogs for your scripts.

Controls

Every custom dialog is comprised of **dialog controls**, items which accept user input of one kind or another. Dialog controls are designed using easily understood metaphors which allow the user to quickly comprehend how a dialog control operates. Once the user understands these simple concepts, it becomes easy for the user to quickly enter data and define complex combinations of settings for a given task. Controls are also designed to provide interactive feedback for the user which guides and informs them as they interact with the dialog.

Controls are organized within the dialog window by means of a **dialog layout**, which positions and orients the controls for display. The dialog layout provides a logical structure for the controls, allowing the user to quickly process information contained in the dialog as well as facilitating data entry into the dialog.

VectorScript provides a rich set of predefined dialog controls for use in custom dialogs. Along with definition functions for each control, VectorScript provides functions for defining and managing the dialog layout, as well as functions for managing control-related data and for creating associated help for each control.

Events

From the script side, the interaction between the user and the dialog is viewed as a series of **events**. Each action the user initiates (such as a keystroke or a mouse click) is viewed as a discrete event which is passed to and processed by the script. The actions taken by the script in response to an event vary from script to script, and are defined according to what the script is designed to accomplish. This flexibility in handling of events is what makes it possible to apply a relatively small set of dialog features to a wide range of script applications.

Processing of user events in VectorScript is accomplished through the use of a structured subroutine known as the **event handler function**. The event handler function contains all the code needed to manage the operation of the dialog while it is displayed.

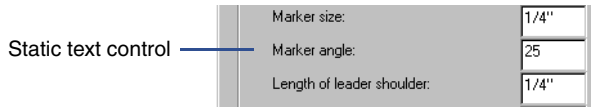
Custom Dialog Controls

VectorScript provides a wide range of control types for use in creating custom dialogs. In addition to basic control types such as editable text fields and radio

buttons, VectorScript also provides specialized controls such as sliders, color palettes, and edit fields which support numeric data entry. The following section lists the custom dialog controls currently available in VectorScript.

Static Text

Static text controls display a non-modifiable text string in the dialog. They are used as labels for other controls, or to display informational text.



Static text strings are left-justified by default; limited right-justification can be obtained by using alignment functions provided by the API. Static text controls support updating of the control text during script run-time.

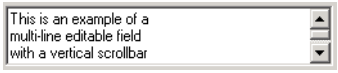
Edit Text

Edit text controls display a single-line editable field in which the user can enter or modify text.

The text value contained in the control can be retrieved using functions provided by the API. Text contained within an the control can also be updated during run-time. Text in edit text controls is always left-justified.

Edit Text Box

This is a multi-line editable field with a vertical scroll bar.



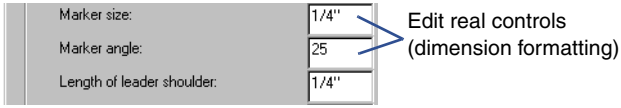
Edit Integer

Edit integer controls are a specialized type of edit control designed for handling numeric input. Edit integer controls return values directly as an `INTEGER` value, eliminating the need for string-number conversions.

Edit integer controls also support in-line expressions which result in a numeric value.

Edit Real

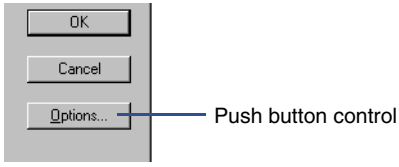
Edit real controls are a specialized type of edit control designed for handling numeric input. Values from edit real controls are returned directly as a `REAL` value, eliminating the need for string-to-number conversions.



Edit real controls can be configured to display the field value in one of several formats, such as dimensions or angular values. Edit real controls also support in-line expressions which result in a numeric value.

Push Button

Push button controls display a standard dialog button. The control is automatically sized based on the specified text string.

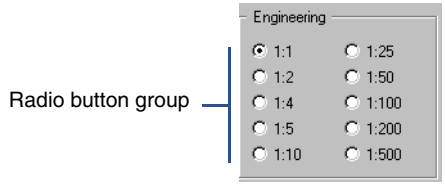


Push button control text cannot be updated during script run-time.

Radio Button

Radio button controls display a standard radio button option control.

Radio buttons are traditionally used in pairs or groups of three to display a set of related options where only one of the settings is active at any time. Related radio button controls are referred to as a **radio button group**.



Check Box

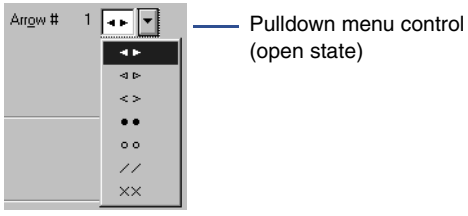
Check box controls display a standard check box option control.

Check boxes are traditionally used to display options that can be set independently of other option items in a dialog.

Pulldown Menu

Pulldown menu controls display one or more selection options in a menu format. The user may select one item from the available options as the active control option.

When in its closed state, the active menu option is displayed in the control. When the control is selected, all menu options are displayed, with the active option highlighted:

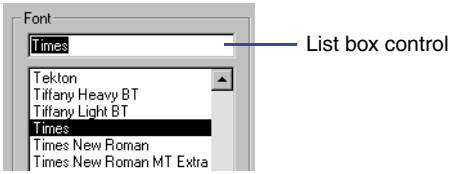


Pulldown menu options can also be navigated by highlighting or tabbing into the control and using the arrow keys to move up and down the list of options.

VectorScript provides API functions for retrieving and managing pulldown menu control options.

List Box

List box controls display a menu containing one or more selection options in a list box format. The user may select an option from the available list items as the active control option.



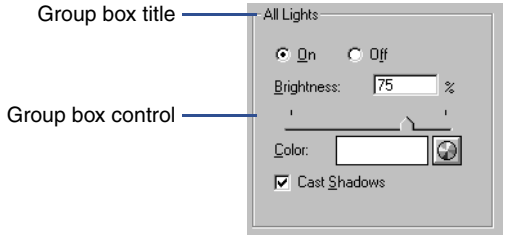
The user-selected option is highlighted in the list box view. List box options can be navigated by highlighting or tabbing into the control and using the arrow keys to move up and down the list of items available in the control. VectorScript API functions for retrieving and managing pulldown menu control options also work with list box controls.

As of VectorWorks 10, list boxes can have multiple columns, each with its own column width. By default, list boxes are created with one column. To add a column, use the VectorScript function `AddListBoxTabStop`, which takes a tab stop as a parameter. Each tab stop is given as a character position. Hence, each succeeding tab stop must be at a greater character position than the previous one.

Once all tab stops have been set up, data can then be added to the list box (all tab stops must be set before data can be added). Data is added in the usual way, using calls to `InsertChoice`. To align text at a tab stop, tab characters are inserted in the string passed to `InsertChoice`. The string for an entire line must be passed to `InsertChoice` all at once; it is not possible to pass just a part of a line.

Group Box

Group boxes are used to associate related items in a dialog box. Other controls, such as radio buttons, pulldown menus, and even other group boxes, can be embedded within a group box control.



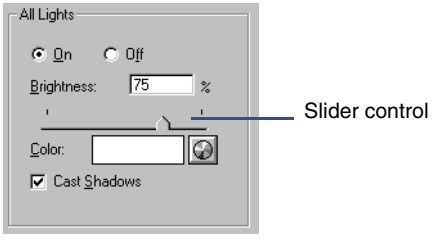
The size of the control is determined by the size of the controls which are embedded in the group box. The title of a group box is optional; group boxes defined without titles will display with a complete box border. Group boxes do not return a data value.

Group boxes can also be configured as invisible to group items as a layout unit within the dialog box.

Group boxes and the controls contained within them can be treated as a single control when performing dialog layout. Adjustments to the group box control will automatically adjust any controls contained within the group box.

Slider

Slider controls allow the user to select from a range of allowable values by positioning the control's slider bar indicator.



Slider controls are displayed with a fixed width, and are only displayed in a horizontal orientation.

Slider controls display range increments as tick marks located under the slider bar. The range increment is a fraction of the maximum value specified for the slider; the number of marks displayed can vary from 1 to 10, depending on the specified value.

Image Pane

Image pane controls display an image retrieved from a VectorScript resource file:

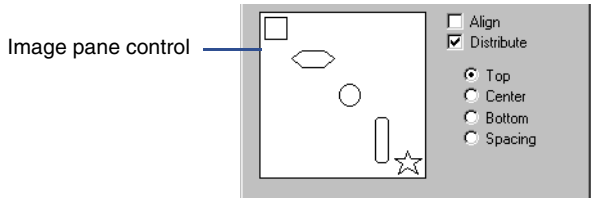
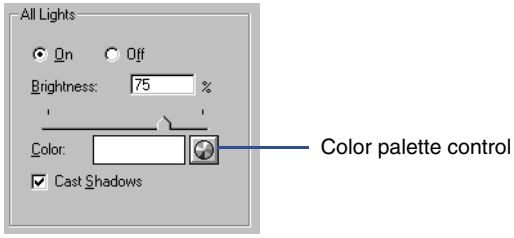
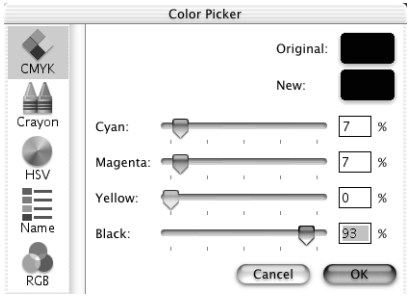


Image pane controls are sized to the dimensions of the graphic image being displayed. The graphic displayed in the image pane control can be updated during script run-time by setting the active image resource for the control.

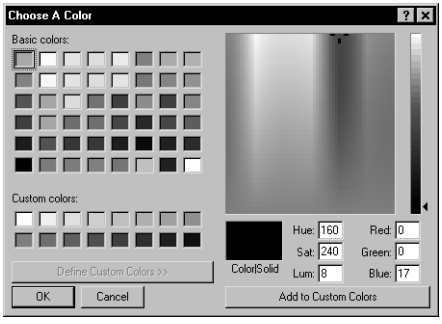
Color Palette

Color palette controls display a system color palette when clicked. The selected color value is returned for use in the script.





System color (Macintosh)

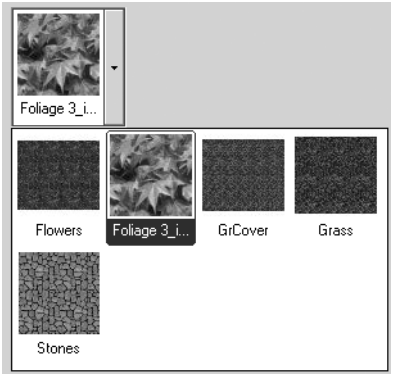


System color (Windows)

The value returned by the color palette control is a decimal representation of a hexadecimal color value. This value must be converted to corresponding RGB values for use with VectorScript color functions.

Image Popup

Image popup controls allow the user to display a selectable preview list of resources.



Gradient Slider

The gradient slider can be used to indirectly manipulate gradient resources.

Creating a Custom Dialog

To create custom dialogs, VectorScript utilizes a "layout manager" which handles all the details of positioning and sizing of controls. If you have written custom dialogs using previous versions of VectorWorks or MiniCAD, you know that the dialog was treated as a canvas, where dialog controls were created and positioned using absolute coordinates. This was often a tedious process, and dialogs created on one platform often did not transfer to other platforms without significant adjustment. Modern custom dialogs in VectorScript treat the dialog as a container for the components of the dialog. Using this methodology allows the details of control sizing and positioning to be handled by the application and results in dialogs which are consistent across platforms and easier to create.

Modern custom dialogs create dialogs in two stages. In the first stage, controls are added to the dialog container; this usually involves a series of control definition function calls which specify the controls to be displayed along with

their default properties. Once all the controls for a dialog have been added to the dialog container, they are organized for final on screen display.

Organizing controls in modern custom dialogs is radically different from the old canvas method in older versions. Controls are arranged by specifying their position relative to other controls, rather than specifying their exact location.

While you may specify character widths and heights for certain controls, for the most part the details of positioning each control are handled for you by the application.

Defining the Dialog Controls

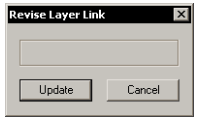
The first step in creating a new VectorScript custom dialog is to define the dialog window and its basic properties. To do this, we will use the custom dialog API function `CreateLayout()`, which creates the dialog window and defines the title, default button, and help display properties of the dialog. `CreateLayout()` then returns an identifier which will be used to add controls to the dialog. This identifier is also used elsewhere in the script to refer to the dialog for control positioning and event handling.

Example:

```
id := CreateLayout('Revise Layer Link',TRUE,'Update','Cancel');
```

The function creates a new empty dialog, entitled **Revise Layer Link**, which contains a help text area and two default buttons (**Update** and **Cancel**). The following script creates the dialog:

```
procedure CreateDialog;
VAR
id: LONGINT;
result : LONGINT;
BEGIN
id := CreateLayout('Revise Layer Link', TRUE, 'Update', 'Cancel');
result := RunLayoutDialog(id,NULL);
END;
RUN(CreateDialog);
```



This is the basic dialog container in which the rest of the dialog definition will be created.

`CreateLayout()` allows you some flexibility in creating the dialog container. If, for instance, you do not wish to provide help text in a dialog (in a confirmation dialog, for example) you can suppress the help text area by specifying FALSE in the help text display parameter of `CreateLayout()`:

```
id := CreateLayout('Revise Layer Link',FALSE,'Update', 'Cancel');
```

Dialog buttons can also be suppressed if not needed. Using the example, if the dialog did not require a Cancel button, you could suppress it simply by specifying a blank string for the button parameter:

```
id := CreateLayout('Revise Layer Link',FALSE,'Update','');
```

The default button for the dialog can also be suppressed in this fashion:

```
id := CreateLayout('Revise Layer Link',FALSE,'','Cancel');
```

Note: *It is possible to suppress both default buttons for a dialog. In this instance, if your code does not provide some alternate means of dismissing the dialog, you will be unable to exit the dialog.*

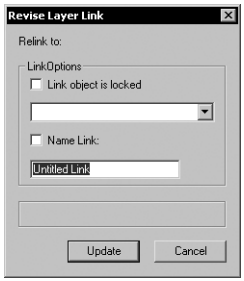
Once you have defined the dialog and its basic properties, you can begin adding the controls to the dialog. A control is added to a custom dialog by calling the appropriate definition function for the control, referencing the dialog in which the control will be displayed using the identifier supplied by `CreateLayout()`. In our example, we will be adding a pulldown menu to display layers that can be selected for the link, controls to let us specify link properties, as well as some additional controls for descriptive text and to organize the dialog. The resulting code is shown below:

```
id := CreateLayout('Revise Layer Link',TRUE,'Update', 'Cancel');

CreateStaticText(id,4,'Relink to:',-1);
CreatePulldownMenu(id,5,32);
CreateGroupBox(id,6,'Link Options',TRUE);
CreateCheckBox(id,7,'Link object is locked');
CreateCheckBox(id,8,'Name Link:');
CreateEditText(id,9,'Untitled Link',26);
```

Each control that will be a part of the dialog is defined with a call to a definition function. The definition for each control specifies the dialog in which it should appear, a unique number identifying the control, and the default properties for the control. The pulldown menu, for example, is created

using the function `CreatePulldownMenu()`, specifying the control ID of 5 and a width of 32 characters.



Once the controls have been defined, you can optionally add help text for some or all controls. Help text provides the user with an easy means of identifying what a control does from within the dialog, and is usually recommended for all but the most basic dialogs.

The function `SetHelpString()` is used to add help for a specific control. The function associates a help string with a control; if the cursor is moved over the control when the dialog is displayed, the associated help string will automatically be displayed in the help text area of the dialog. The dialog control definition code for the example with help strings added is shown below:

```
id := CreateLayout('Revise Layer Link',TRUE,'Update', 'Cancel');

CreateStaticText(id,4,'Relink to:',-1);
CreatePulldownMenu(id,5,32);
CreateGroupBox(id,6,'Link Options',TRUE);
CreateCheckBox(id,7,'Link object is locked');
CreateCheckBox(id,8,'Name Link:');
CreateEditText(id,9,'Untitled Link',26);

SetHelpString(1,'Update the selected layer link.');
   SetHelpString(2,'Cancel the operation and exit.');
   SetHelpString(4,'New layer to be displayed by selected link.');
   SetHelpString(5,'New layer to be displayed by selected link.');
   SetHelpString(7,'Lock the link object after it has been updated.');
   SetHelpString(8,'Apply an object name to the layer link.');
   SetHelpString(9,'Apply an object name to the layer link.');
```

In the example, note that we have repeated certain help text strings. We did this in order to provide useful help for the item whether the cursor was over the actual control or over the label associated with the control. Also, help text was omitted for the group box control; group boxes do not have associated help text.

Defining the Dialog Layout

Positioning dialog controls is generally a two step process, where an initial arrangement specifies the relative position of each control and then any special alignments are specified.

Dialog items are arranged by setting an initial anchor control and then specifying a chain of controls relative to the first control. Layouts and group items are the only two objects that can have anchor controls. Anchor controls are set using either `SetFirstLayoutItem` or `SetFirstGroupItem`. The next item is placed relative to the anchor item using either `SetBelowItem` `SetRightItem`. Using these calls, a chain of controls can be created with each item relative to the other. Group items are just like other items in that any control including another group can be placed to the right of or below another group.

The initial arrangement generally places items so that their left and top edges are aligned. To specify other alignments use the `AlignItemEdge` call. In `AlignItemEdge` you specify an edge and an alignment group. All objects in the same alignment group are aligned together. `AlignItemEdge` also allows you to specify whether you want an object to shift or resize when performing the alignment.

Running the Dialog

After creating the controls and arranging the layout, the script is ready to run the dialog. The `RunLayoutDialog()` function will show the dialog on screen and begin handling the user interaction with the dialog. The dialog will look appropriate for the computer platform it is running on—Macintosh or Windows.

Handling Dialog Events

The script can respond to user events by defining its own event handling function and passing the name of that function to the `RunLayoutDialog` call. When the user presses a button or clicks in a list, for example, VectorWorks will call the event handling function. The function will receive

the control item number and any appropriate data. The procedure will be called with an item of `SetupDialogC` before the dialog is displayed so that the script can initialize its controls.

```
Procedure HandleEvents( VAR item : LONGINT; data : LONGINT);
Begin
    case item of
        SetupDialogC:
        Begin
            InsertChoice( kPullDown, 0, 'choice 0');
            InsertChoice( kPullDown, 1, 'choice 1');
        End;

        kCancelButton:
        Begin

        End;

        kOKButton:
        Begin

        End;

    End;

End;
```

Using VectorScript Plug-ins

In this Chapter:

VectorWorks 8 introduced the concept of VectorScript plug-ins, which allow scripts to be directly integrated into a VectorWorks workspace and be made available to any VectorWorks document. The three types of plug-ins—**menu commands** (.vsm), **tools** (.vst), and **objects** (.vso) — allow scripts to integrate into both workspace menus and tool palettes, as well as other VectorWorks features such as the Resource Browser.

In addition to better integration into the VectorWorks environment, plug-ins also provide new script functionality in the form of **plug-in objects**. Plug-in objects created with VectorScript can be used to create entirely new classes of items that can streamline and enhance the design/drafting process for documents. They support standard VectorWorks core technologies such as snapping, classing, and advanced object editing, giving them essentially the same status as VectorWorks built-in object types.

VectorScript plug-ins also provide enhanced portability and platform independence for scripts, allowing them to be easily moved to VectorWorks installations on either Macintosh or Windows systems.

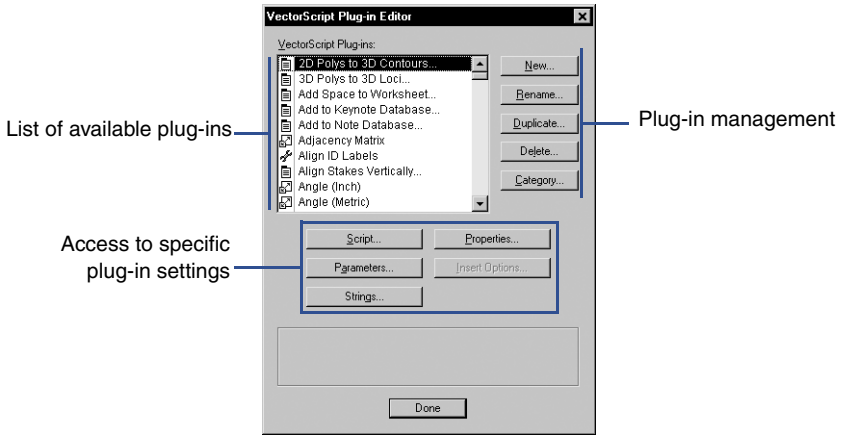
VectorScript plug-ins can also be localized for use in other countries. The names and strings that are displayed can be translated to another language. Drawings containing plug-in objects can be exchanged between users in different countries.

This chapter explains the basic concepts related to creating and using VectorScript plug-ins with VectorWorks.

Creating and Using Plug-ins

VectorScript plug-ins are created using the VectorScript Plug-in Editor, which can be opened by selecting

Organize > Scripts > Create Plug-in. The plug-in editor provides access to all the settings needed to define any type of VectorScript plug-in, and can also be used to edit existing plug-ins as well. The editor interface provides a listing of all plug-ins currently available to VectorWorks, as well as tools for managing individual plug-in items. Access to the various settings of a plug-in is available by clicking on one of the buttons in the main editor dialog; doing so will display detailed information on the selected setting, which can then be modified as desired.



For information on using the Workspace Editor to add VectorScript plug-ins to a workspace, see Appendix B in the VectorWorks User's Guide.

For additional details on creating or editing specific plug-in types, see Chapters 11 through 16.

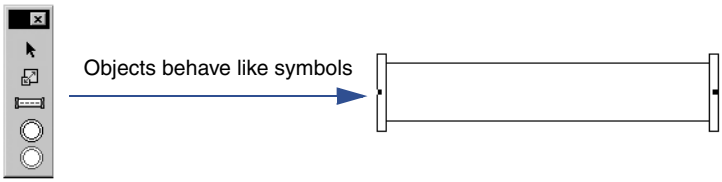
Once a new plug-in has been created using the Plug-in Editor, it can be made available for use in VectorWorks by using the Workspace Editor to add the plug-in item to one or more workspaces. Once the item has been added to a workspace, it is available to any open document in VectorWorks.

Using the Different Types of Plug-ins

A key feature of VectorScript plug-ins is their smooth integration into the VectorWorks product interface. VectorScript plug-in menu commands and tools work just like any built-in VectorWorks tool or menu item. Like built-in menu commands, VectorScript menu commands can be set to require certain document conditions such as 2D/3D view orientation or a selected set of items in order to activate. VectorScript tools, like their built-in counterparts, make use of the SmartCursor and other tool-centric VectorWorks features in order to provide full functionality for these items.

Plug-in objects have characteristics of both VectorWorks tools and VectorWorks symbols. Plug-in objects can be added to a VectorWorks tool

palette and resemble tool items, but in use they will place instances of the object in the document much like the symbol tool will place symbols in a document.



Object instances in a document can be modified by using the Object Info palette to edit the parametric values that are used to define the object. These values, which can be edited individually or globally, give plug-in objects enormous flexibility as to how they can be displayed within a document.

Plug-in objects can also be used in conjunction with the Resource Browser to create preconfigured object instances that need minimal editing after placement. Libraries of different object configurations based on a single plug-in object can be easily created and retrieved through the Resource Browser.

How Plug-ins Work

VectorScript plug-ins combine regular VectorScript script code with a plug-in "wrapper," an encoded header which contains the information that defines the characteristics and behaviors of the plug-in. Information such as the category of the plug-in, properties which define how the plug-in is activated by VectorWorks, or any other information needed by the plug-in to function within the VectorWorks application framework is included within the header which "wraps" the script.

When VectorWorks is launched, it searches the Plug-ins folder for any VectorScript plug-in that is included in the active workspace and registers the information necessary to activate and manage the plug-in.

Note: *A plug-in must reside in the Plug-ins folder when the application is launched or the workspace is activated for it to be available in the current VectorWorks session.*

When a menu command or tool item is selected, the script and any information needed by the plug-in is loaded into memory, and the plug-in script executes. VectorWorks uses information provided by the plug-in to

provide the user interactions (such as constraints) and document environment for the menu command or tool can perform its defined actions.

VectorScript objects work similarly to VectorScript menu commands or tools, but their scripts can also be invoked through events that occur in the document. Placed object instances can be edited or modified using either the SmartCursor or Object Info palette, and these changes will cause the script defining the object to execute in order for the object to redraw. Global document changes which force a regeneration of the document can also cause the scripts of objects placed in the document to execute.

Understanding Plug-In Parameters

At the core of VectorScript plug-ins is the concept of **plug-in parameters**, values which are used to define plug-in objects and which can be used to store persistent values associated with all types of plug-ins. Plug-in parameters are the "glue" which holds VectorScript plug-ins together.

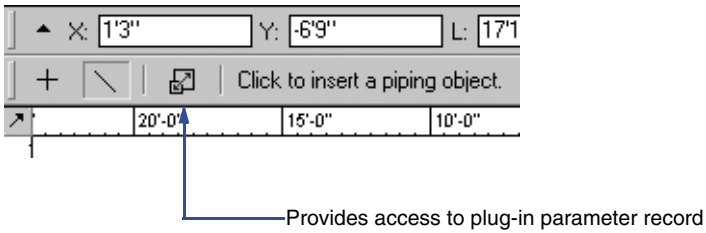
How Parameters Work

Plug-in parameters are stored in a **parameter record**, which is a specialized type of VectorWorks record format. Parameter records are required by VectorScript objects, but they can be created and associated with any type of VectorScript plug-in.

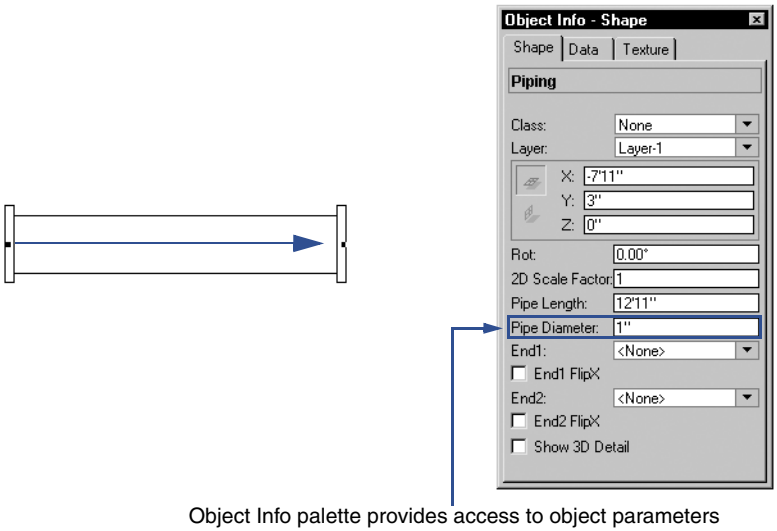
Values are stored in the parameter record by means of **parameter fields**. These fields act like regular record fields and can be accessed for reading and writing by the plug-in script. Parameter fields have distinct data types for storing different kinds of information; information on specific parameter types may be found later in this section.

The values stored in the parameter record of a VectorScript plug-in can be used in one of several ways. Objects use the stored parameter values to provide the data necessary to define and create the appearance of the object instance in the document. Menu commands and tools use parameter records to store information for subsequent use with the plug-in item.

Parameter records can be associated with the plug-in item itself, or, in the case of objects, can be associated with each instance of an object placed in a VectorWorks document. Parameter records associated with plug-in items are not visible in the Object Info palette or in the Resource Browser, and cannot be edited from either location. Tool items and objects do provide access to the parameter record of the plug-in by clicking the Plug-in Preferences button displayed in the mode bar when the tool or object is active.



Parameter records associated with object instances placed in a document can be edited by selecting the object instance and editing the parameter field values through the Shape pane of the Object Info palette.



Parameter Types

VectorScript provides ten parameter field types for use with plug-ins:

- Integer
- Boolean
- Number
- Text
- Popup
- Radio Button
- Dimension

- X-Coordinate
- Y-Coordinate
- Control Point

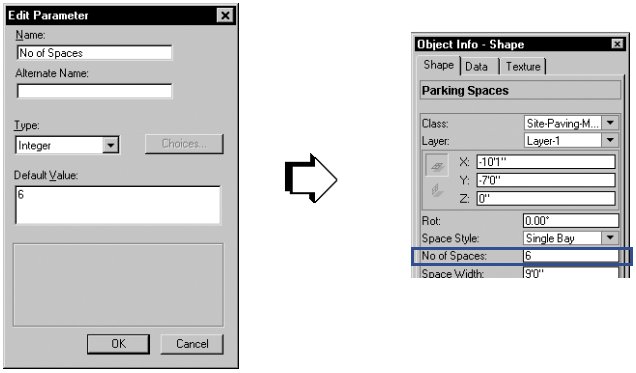
Parameter records can have multiple parameter fields of the same type, or combinations of parameter fields of different types. The following sections document each parameter type in detail.

Integer

Integer parameters store a single `INTEGER` data value.

An integer parameter value is displayed in the Object Info palette in an editable field, and can be edited as desired. Integer parameter fields support calculations in the field, fractional values entered into an integer parameter field will be rounded to the nearest value.

Integer parameters do not support unit marks or unit conversion.

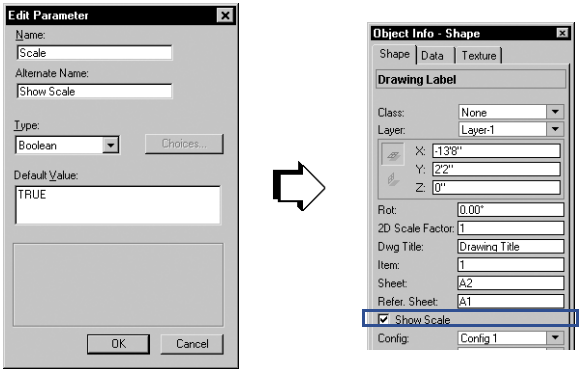


Boolean

Boolean parameters store a single `BOOLEAN` data value.

A boolean parameter is displayed in the Object Info palette as a check box, with the state of the check box indicating the `TRUE-FALSE` state of the value (`TRUE` = checked, `FALSE` = unchecked).

Boolean parameters do not support unit marks or unit conversion.

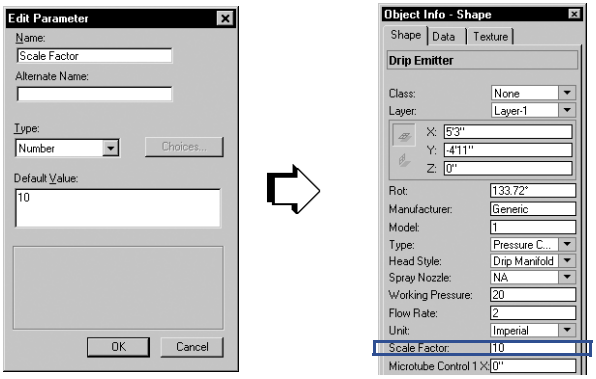


Number

Number parameters store a single `REAL` data value.

A number parameter value is displayed in the Object Info palette in an editable field, and can be edited as desired. Number parameter fields support calculations in the field, and fractional values entered into a number parameter field will be displayed using the current units fractional display setting.

Number parameters do support unit marks or unit conversion.

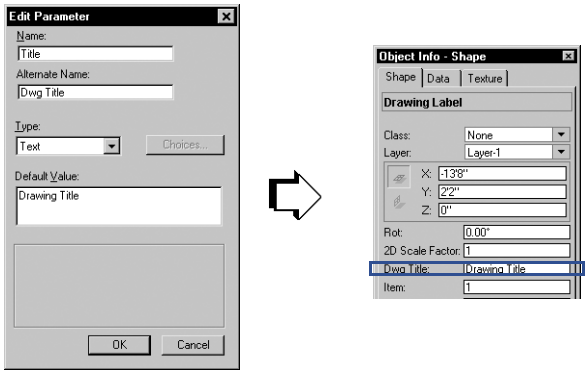


Text

Text parameters store a single string data value. The stored value may be up to 32K in length.

A text parameter value is displayed in the Object Info palette in an editable field, and can be edited as desired. For text values which are greater than 255

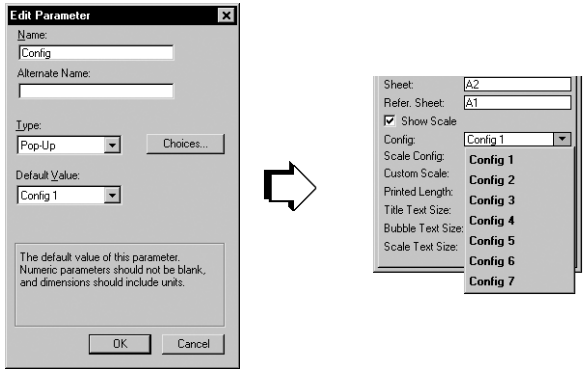
characters in length, use a `DYNARRAY OF CHAR` to store the value during script execution.



Popup

Popup parameters store a single `STRING` data value that is selected from a predefined list of values. The list of available values is defined in the parameter definition dialog, and cannot be modified during script execution.

Popup parameter values are displayed in the Object Info palette as popup menu listing the defined value options. The active parameter value (the value which is stored in the parameter) is indicated by the value displayed in the popup when the control is not selected, and by a bullet next to the item when the control is selected. To modify the value, select the desired parameter value from the popup control.



Radio Button

Radio button parameters store a single `STRING` data value that is selected from a predefined list of values. The list of available values is defined in the parameter definition dialog, and cannot be modified during script execution.

Radio button parameter values are displayed in the Object Info palette as series of radio buttons in a group box, with one radio button for each defined value. The active parameter value (which is stored in the parameter) is indicated by the selected radio button. To modify the value, select the radio button corresponding to the desired value.

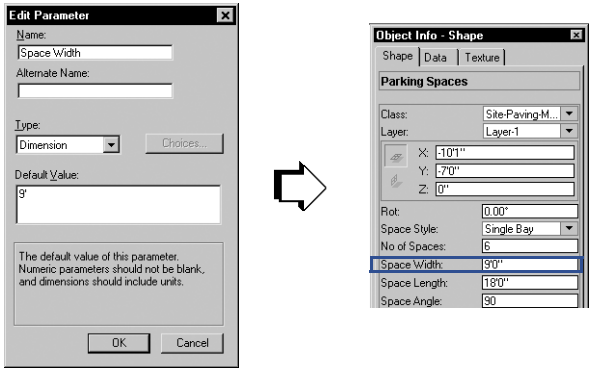
Dimension

Dimension parameters store a dimension data value as a `REAL` numeric value.

A dimension parameter value is displayed in the Object Info palette in an editable field, and the value can be edited as desired. Dimension parameter fields support calculations in the field, and fractional values entered into a dimension parameter field will be displayed using the current unit's fractional display setting.

Dimension parameters support the use of unit marks with values; values stored in one unit format will be automatically converted to an equivalent value if the document unit setting is modified.

Dimension parameters are not sensitive to changes in the user origin of a document.



X-Coordinate

X-coordinate parameters store a coordinate data value as a `REAL` numeric value.

A coordinate parameter value is displayed in the Object Info palette in an editable field; the value can be edited as desired. Coordinate parameter fields support calculations in the field, and fractional values entered into a coordinate parameter field will be displayed using the current units fractional display setting.

Coordinate parameters support the use of unit marks with values; values stored in one unit format will be automatically converted to an equivalent value if the document unit setting is modified.

Coordinate parameters are sensitive to changes in the user origin of a document, and are designed to be used with geometric data that is related directly to locations within a VectorWorks document. Values displayed in coordinate fields will be corrected for any changes in the document user origin.

Y-Coordinate

Y-coordinate parameters store a coordinate data value as a `REAL` numeric value.

A coordinate parameter value is displayed in the Object Info palette in an editable field; the value can be edited as desired. Coordinate parameter fields support calculations in the field, and fractional values entered into a coordinate parameter field will be displayed using the current unit's fractional display setting.

Coordinate parameters support the use of unit marks with values; values stored in one unit format will be automatically converted to an equivalent value if the document unit setting is modified.

Coordinate parameters are sensitive to changes in the user origin of a document, and are designed to be used with geometric data that is related directly to locations within a VectorWorks document. Values displayed in coordinate fields will be corrected for any changes in the document user origin.

Control Points

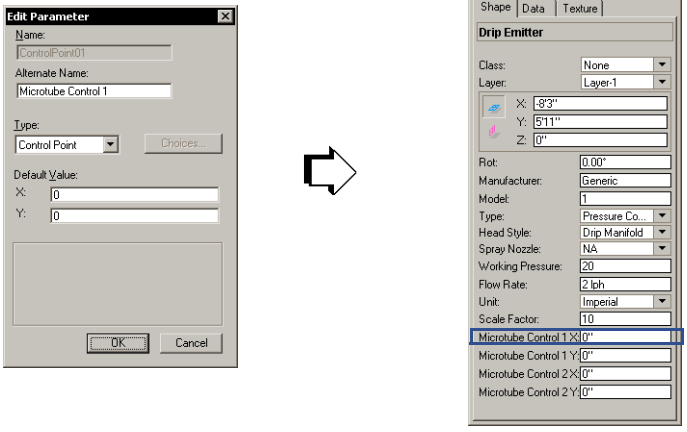
Control point parameters are a specialized parameter type designed to create control points in plug-in objects. A control point is similar to a selection handle and allows the user to click and drag to reshape the object. When created, a control point parameter consists of two linked coordinate parameters. The two parameters correspond to x- and y-coordinate fields for the control point.

Control point parameters are displayed in the Object Info palette as a pair of editable coordinate fields; the values can be edited as desired. Control points, like coordinate fields, support calculations in the fields. Fractional values entered into a control point parameter field will be displayed using the current units fractional display setting.

Control point parameters support the use of unit marks with values; values stored in one unit format will be automatically converted to an equivalent value if the document unit setting is modified.

Control point parameters are sensitive to changes in the user origin of a document, and values displayed in the field will be corrected for any changes in the document user origin.

Control point fields can be renamed by entering a display name in the alternate name field of the parameter. This value will be used as the control label in the Object Info palette; when referring to the parameter in a script, use the actual definition name of the parameter.



Accessing Parameters from Scripts

VectorScript provides a well-defined mechanism for directly accessing values in parameter records within plug-in scripts. This mechanism, known as **parameter referencing**, allows parameter values to be easily retrieved for use with scripts.

The generalized syntax for parameter references is as follows:

```
P<name of parameter>
```

Parameter names should be specified in all uppercase letters, with underscores representing any embedded spaces in the parameter name. For example, a dimension parameter named `Space Width` would be referenced in a script as:

```
PSPACE_WIDTH
```

Parameter references can be used to assign values to other identifiers in a script. Supported identifiers include variable, array, array element, and structure member identifiers. For example, assigning the value in the parameter to a variable would be defined as:

```
sp_width:= PSPACE_WIDTH;
```

Parameter references can also be used like constants in expressions or function arguments. For example, valid uses of `Space Width` parameter would include:

```
totalWidth:= 5 * PSPACE_WIDTH;
```

```
CalculateTotal(PSPACE_WIDTH,2);
```

Parameter references should always be treated as constant values. Parameter references do not accept value assignments, and parameter reference values cannot be modified.

Setting Parameter Values from Scripts

VectorScript uses the `SetRField()` function to write values to parameter records. VectorScript also provides two functions, `GetCustomObjectInfo()` and `GetPluginInfo()` which return the information needed by `SetRField()` to write values to parameter records.

Using `GetCustomObjectInfo()` or `GetPluginInfo()` with `SetRField()` is relatively straightforward. When writing a value back to the parameter record of an object instance, first use `GetCustomObjectInfo()` to obtain information about the object. Once this information has been retrieved, it can be used in conjunction with `SetRField()` to write the value back to the parameter record. The following example illustrates this technique:

```
BEGIN
       resultStatus:= GetCustomObjectInfo(objName,objHd,recHd,wallHd);


   IF resultStatus THEN BEGIN
sp_width:= PSPACE_WIDTH;
```

```
...
...
sp_width:= 5 * sp_width;
...
...
SetRField(objHd,GetName(recHd),'Space Width', Num2StrF(sp_width));
END;
END;
```

In the example, `GetCustomObjectInfo()` is called to obtain the name of the object and a handle to both the object instance and its associated parameter record. This information is then used with `SetRField()` to write the value to the parameter record field.

Note that when writing values to the parameter record, the actual name of the field, not the parameter reference, is used. Parameter references should only be used for retrieving data from the parameter record.

The example also points out one additional requirement for using `SetRField()`. In the example, the value in `sp_width` is a `REAL`, but `SetRField()` requires a `STRING` argument for the value being assigned to the record field. In this case, it will be necessary to convert the dimension value to a `STRING` for compatibility with the function call. The parameter record will convert the value back to the appropriate data type when it is stored.

The method for writing values to the parameter records of menu commands and tool items is almost identical to the method used for objects. In the case of menu commands and tool items, the function `GetPluginInfo()` should be used to obtain the plug-in name and a handle to the parameter record. The example below illustrates how the function is used with a menu command:

```
BEGIN
    ...
...
IF GetPluginInfo(cmdName,pRecHd) THEN
offvalue:= GetField(5);
numlines:= GetField(6);
cmdHd:= GetObject(cmdName);
SetRField(cmdHd,GetName(pRecHd),'Offset',Num2StrF(offvalue));
SetRField(cmdHd,GetName(pRecHd),'Lines',Num2Str(0,numLines));
```

```
END;
...
...
END;
```

In the example, `GetPluginInfo()` is used to obtain the name of the menu command and a handle to the parameter record. This information is used with `SetRField()` to write values to the parameter record of the menu command.

Parameter records for menu commands and tool items are very useful for storing information between uses of the command or tool item. For example, if a user modifies the default settings of a tool item, this information can be stored and reused on subsequent uses of the tool.

Note: *Like objects, records for menu commands and tool items are stored with a VectorWorks document; switching documents may cause the default settings for a command or tool item to change.*

Setting Parameter Visibilty

By default, all plug-in parameter users interface controls are enabled and visible. This behavior may be overridden using `SetParameterVisibility()` and `EnableParameter()`. Using `GetCustomObjectInfo()` or `GetPluginInfo()` with these procedures is relatively straightforward. First, use `GetCustomObjectInfo()` to obtain information about the object. Once this information has been retrieved, the plug-in parameter's user interface attributes can be specified. In each case, the parameter argument is the universal name of the plug-in's parameter. For example:

```
BEGIN
...
resultsStatus := GetCustomObjectInfo(objName,objHd,recHd,wallHD);
IF resultStatus THEN BEGIN

EnableParamter(objHD, 'Space Width'. FALSE);
{Disables the control for the Space Width parameter}

SetParameterVisibility(objHd, 'Space Depth'. FALSE);
{Hides the control from the Space Width parameter}
...
```

```
END;
END;
```

Like the plug-in object example above, plug-in menus and tools that use parameters can use `SetParameterVisibilty()` and `EnableParameter()`.

```
BEGIN
...
...
IF GetPluginINfo(cmdName,pRecHd) THEN
cmdHd:= GetObject(cmdName);
EnableParameter(objHd, 'Offset'. FALSE);
SetParameterVisibility(objHd, 'Lines'. FALSE);
END;
...
...
END;
```

Setting Default Parameter Visibility

Additionally, default parameter visibility may be set for objects, menus, and tools in the Create Plug-in/Edit Parameter dialog. By placing two leading underline characters as a prefix to the parameter's universal name, the parameter visibility is set to false. Using this technique hides a parameter in the default settings dialog for a plug-in. Parameters with this special universal name prefix will not be shown unless explicitly made visible using `SetParameterVisibility`.

VectorScript Menu Commands

In this Chapter:

VectorScript menu commands (`.vsm`) plug-ins allow scripts to be inserted into a VectorWorks workspace as a menu command item. VectorScript menu commands can be used like any standard menu command item, performing operations on the active VectorWorks document. As part of a workspace, menu command plug-ins (unlike document scripts) are available to any open VectorWorks document without the need for importing the script into the active document.

VectorScript menu commands support the standard behaviors expected from standard VectorWorks menu items. VectorScript menu commands can detect the view state of the active VectorWorks document, or can determine if a selection set exists which the menu command can act on.

This section documents the basic techniques needed to create and use VectorScript menu commands with VectorWorks.

Creating a Menu Command Plug-in

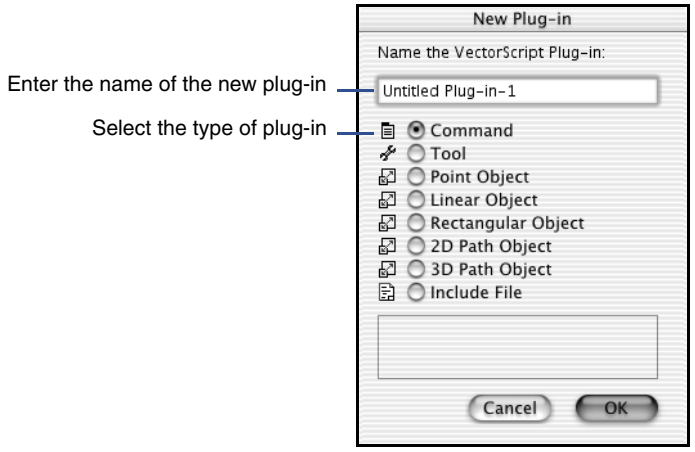
VectorScript menu commands are created using the Plug-in Editor to create and define the actual plug-in item. The Plug-in Editor provides a single interface for creating the plug-in script and editing the associated information that will be used by the menu command when used during a VectorWorks session.

Creating the Menu Command Plug-in

To create a menu command plug-in:

1. Select **Organize > Scripts > Create Plug-in**.

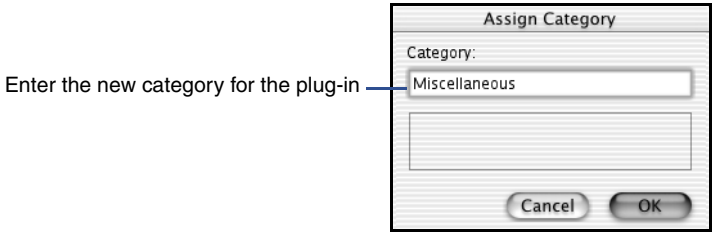
2. In the VectorScript Plug-in Editor dialog box, click the **New** button.
3. In the Assign Name dialog box, enter the name of the new plug-in item and select **Command** as the type of the plug-in. Click **OK** to create the plug-in item.



Setting the Category of the Menu Command

To set the category:

1. Click the **Category** button from the VectorScript Plug-in Editor dialog box.
2. In the Assign Category dialog box, enter the name of the category to be associated with the new plug-in item. Click **OK** to set the plug-in category to the new value.



The plug-in category is the heading under which the object may be found when selecting items in the Workspace Editor.

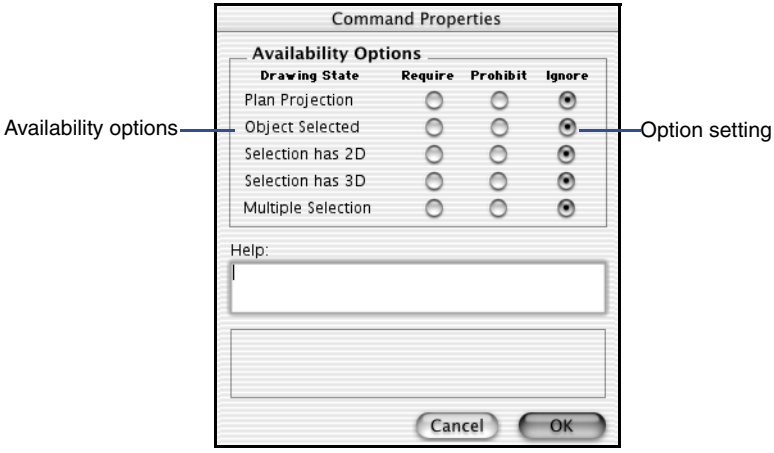
Setting Options for Menu Commands

Menu command plug-ins have several settings options which allow them to behave like standard VectorWorks menu commands. These settings, also known as **plug-in properties**, control behavior of the menu command with respect to the state of the document (selection status, view orientation) as well as defining the help text that will be available for the command.

Setting Document Properties for the Command

To set document properties:

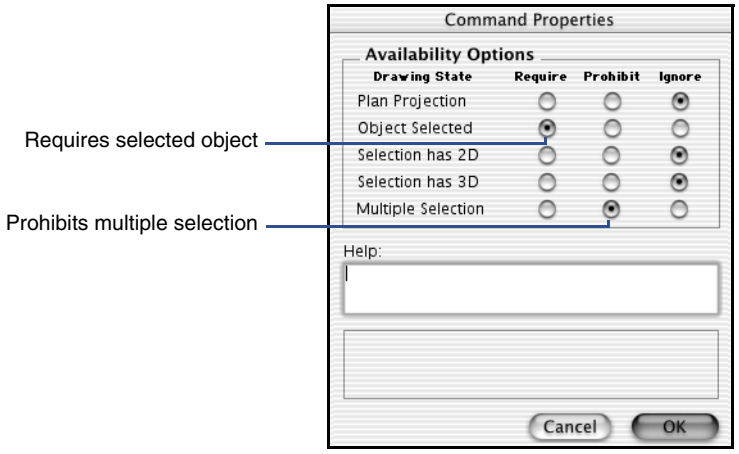
1. Click the **Properties** button from the VectorScript Plug-in Editor dialog box.
2. In the Command Properties dialog box, select the appropriate availability options for the plug-in.



Availability options for menu commands can be set to one of three states:

- **Require**, which will require the document state condition to exist for the command to be active,
- **Prohibit**, which will deactivate the command if the document state condition exists,
- **Ignore**, which will ignore the document state condition.

If a menu command is designed to act only on a single selected object, for example, availability options would be set to require object selection, but prohibit multiple selection.



The menu command will be disabled when the document state does not match the indicated option settings.

3. Click **OK** to save the new settings for the object.

Setting Help Text for the Menu Command

Help text describing the menu command item will display when the cursor is held over the command.

Note: *Help text for menu commands is currently only available on Macintosh systems.*

To create help text:

1. Click the **Properties** button from the VectorScript Plug-in Editor dialog box.

   The Command Properties dialog box opens.

2. Enter the desired help text in the **Help** field.

   The text will be displayed when help is enabled and the cursor placed over the menu command.

3. Click **OK** to save the new settings for the object.

Parameters and Menu Commands

Menu commands can have parameter records associated with the plug-in item. Such records can be used for persistent data storage between uses of the command, as well as providing default menu command values. A menu command which displays a dialog, for example, might need to store values entered by a user for later use. These values can be stored in the parameter record of the menu command and retrieved later when the command is again selected.

A default parameter record is created in the document when on the first use of the menu command with the active document. This default parameter record stores the command's default settings with the document. It is used by the menu command when it is activated for use.

Switching documents will display stored values associated with the new document or, if no parameter record exists, will display the default values of the parameter record as created by the plug-in item.

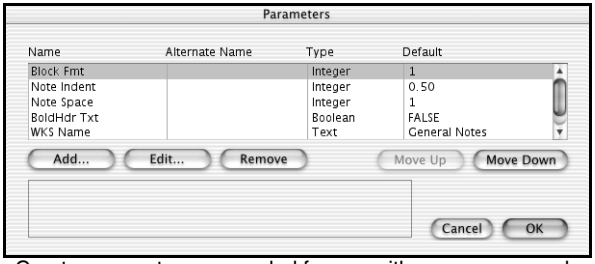
Creating a Parameter Record for a Menu Command

To create a parameter record:

1. Click the **Parameters** button from the VectorScript Plug-in Editor dialog box.

   The Parameters dialog box opens.

2. Create the desired parameter record settings for the plug-in. For more information on plug-in parameters, see "How Plug-ins Work" on page 10-3.



Create parameters as needed for use with menu command

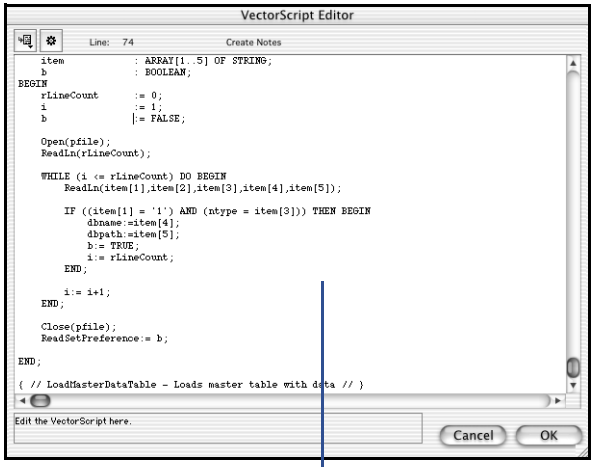
3. When all the parameters have been created, click **OK**.

Creating Script Code for a Menu Command

The script source code for the menu command can be created using the VectorScript editor or a third-party text editor. The source code is saved as part of the plug-in item.

To create script code:

1. Click the **Script** button from the VectorScript Plug-in Editor dialog box.
2. Enter the script source code in the VectorScript Editor window.



Enter script source code for the plug-in

3. When the script has been entered, click **OK** to save the script as part of the plug-in.

Working with Menu Commands

Adding a Menu Command to a Workspace

Once a menu command has been created, it will need to be added to one or more workspaces to be available for use with VectorWorks. Once the command is added to the workspace, it will immediately be available for use in the current VectorWorks session.

For details on editing workspaces, see Appendix B in the VectorWorks User's Guide.

To add a menu command to a VectorWorks workspace:

1. Select **File > Workspaces > Workspace Editor**.
2. In the list of available menu commands in the Workspace Editor, look for the category that was assigned to the menu command. Click the disclosure triangle to display the available items in the category.
3. Click and drag the menu command to the desired location in the workspace menu structure. If desired, add a key equivalent for the menu command.
4. Click **OK** to save the workspace with the added menu command item.

VectorScript Tool Items

In this Chapter:

VectorScript tool item (`.vst`) plug-ins allow scripts to be added to a VectorWorks workspace as a tool palette item. VectorScript tools, like standard VectorWorks tools, have support for the full range of VectorWorks feature technology. VectorScript tools make use of the SmartCursor, and can respond to document state conditions such as selection status or view orientation. As part of a workspace, VectorScript tools (unlike document scripts) are available to any open VectorWorks document without the need for importing the script into the active document.

This section documents the basic techniques needed to create and use VectorScript tool items with VectorWorks.

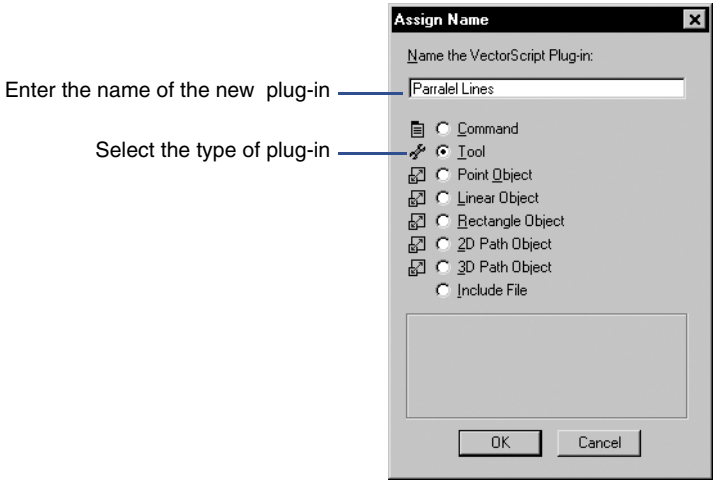
Creating a Tool Item Plug-in

VectorScript tool items are created using the VectorWorks Plug-in Editor to define and create the actual plug-in item. The Plug-in Editor provides a single interface for creating the plug-in script and defining the associated settings which affect the tool item when used during a VectorWorks session.

Creating the Tool Plug-in

To create a tool plug-in:

1. Select **Organize > Scripts > Create Plug-in**.
2. In the VectorScript Plug-in Editor dialog box, click **New**.
3. In the Assign Name dialog box, enter the name of the new plug-in item and select **Tool** as the type of the plug-in. Click **OK** to create the plug-in item.



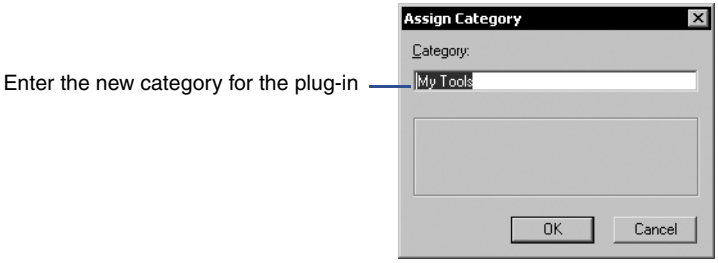
Setting the Tool Category

To set the tool category:

1. Click the **Category** button from the VectorScript Plug-in Editor dialog box.

   The Assign Category dialog box opens.

2. Enter the name of the category to be associated with the new tool item. Click **OK** to set the plug-in category to the new value.



The plug-in category is the heading under which the tool is referenced when selecting items in the Workspace Editor.

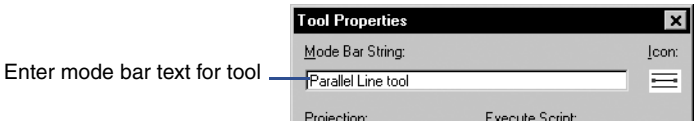
Setting Options for the Tool

Tool items have a number of settings which allow them to maintain behavior consistent with standard VectorWorks tools. These settings, also known as **plug-in properties**, control the conditions under which the tool is activated, as well as various display options for the tool.

Setting Mode Bar Text for the Tool

To set the mode bar text:

1. Click the **Properties** button in the Plug-in Editor dialog box.
2. In the Tool Properties dialog box, enter the desired descriptive text to be displayed in the mode bar in the **Mode Bar String** field.

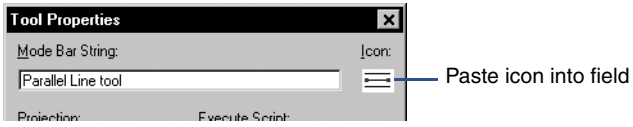


Mode bar text usually includes the name of the tool, and can include text indicating the action the user should perform.

Setting the Tool Icon

VectorScript plug-ins come preloaded with a default icon which is displayed when the plug-in is placed in a workspace tool palette. This icon can be replaced with a custom icon indicating the function of the plug-in item.

To create the icon, use a third-party icon editor. Tool icons use an 8-bit, 32-by-32 pixel field for defining the icon; the actual icon graphic, however, should be confined to an area 24 pixels wide by 18 pixels high, centered in the field. Once the icon is created, select the field and copy the icon to the clipboard.



To create the VectorScript tool icon:

1. Click the **Properties** button from the VectorScript Plug-in Editor dialog box.

The Tool Properties dialog box opens.

2. Click in the icon display field to select it. The icon field will be highlighted.
3. Paste the new tool icon into the icon field. The customized icon should be visible in the tool icon field of the dialog box.

Setting Activation Options for the Tool

Activation options for objects control the conditions under which the object script code will execute. VectorWorks presets these options to the optimal configuration for object interaction with the VectorWorks application.

Setting View Projection for the Tool

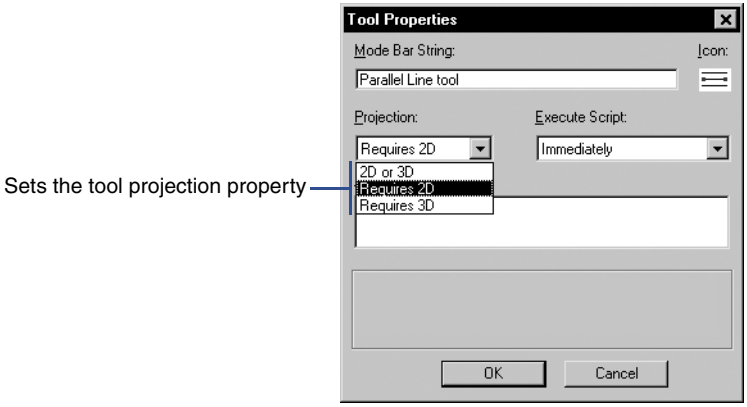
The projection property of the tool determines what view projection must be active for the tool to be used in the document. If the required projection is not active, the user will be prompted before the view orientation is switched to the correct projection.

To set the view projection property:

1. Click the **Properties** button in the Plug-in Editor dialog box.

   The Tool Properties dialog box opens.

2. Select the appropriate projection option for the plug-in.



Setting Script Execution Options for the Tool

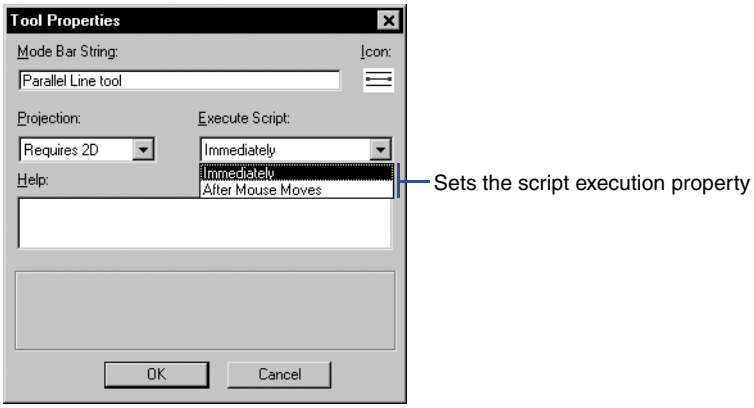
Tool items are set by default to execute immediately when selected. In some cases, however, it may be desirable to have the script execution wait for mouse movement (such as a tool which draws interactively based on user mouse movement).

To set the script execution property:

1. If the Tool Properties dialog is not already open, click the **Properties** button from the VectorScript Plug-in Editor dialog box.

   The Tool Properties dialog box opens.

2. Select the appropriate script execution option for the tool.



Setting Help Text for the Object

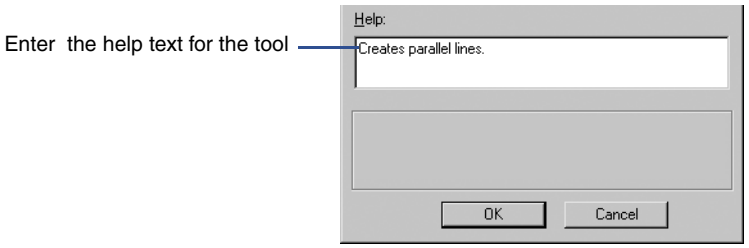
Help text describing the object will display when the cursor is held over the object icon in a tool palette.

To create help text for the object:

1. Click the **Properties** button from the VectorScript Plug-in Editor dialog box.

   The Tool Properties dialog box opens.

2. Enter the desired help text in the **Help** field. The text will be displayed when help text is enabled and the cursor placed over the tool item.



Parameters and VectorScript Tools

Tool items can have parameter records associated with the plug-in item. Such records can be used for persistent data storage between uses of the tool, as well as for setting default tool item values. A tool might, for example, provide several mode options in a popup list. Should the user wish to select a different mode for the tool, the new setting can be saved and reused on a subsequent use of the tool item.

A default parameter record is also created in the document when on the first use of the tool item in the active document. This default parameter record stores the tool item's default settings with the document. It is used by the tool when the tool is activated for use.

Switching documents will display stored values associated with the new document or, if no parameter record exists, will display the default values of the parameter record as created by the plug-in item.

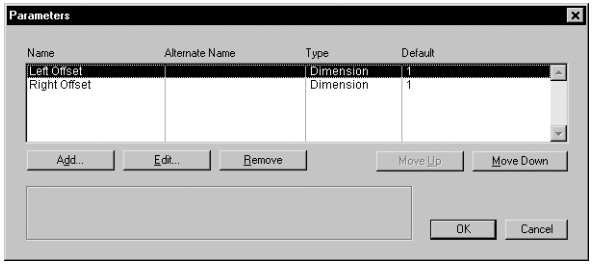
Creating a Parameter Record for a Tool

To create a parameter record:

1. Click the **Parameters** button from the VectorScript Plug-in Editor dialog box.

   The Parameters dialog box opens.

2. Create the desired parameter record fields for the plug-in. For details on plug-in parameters, see "Using VectorScript Plug-ins" on page 10-1.



3. When all parameters have been created, click **OK** to save them.

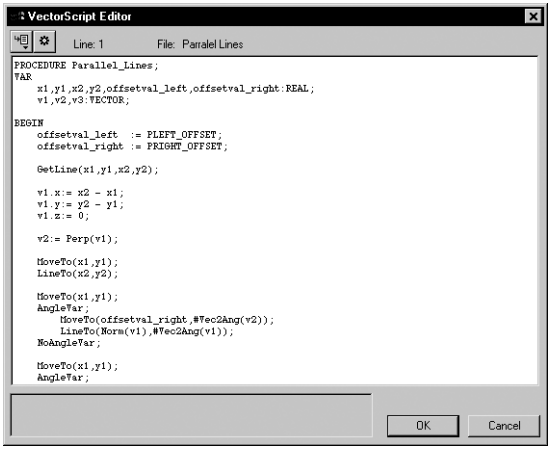
Creating the Tool Script

The script source code for the tool item plug-in can be created using the VectorScript editor, or it can be created with a third-party text editor and imported into the plug-in. The source code is then saved as part of the plug-in item.

Creating Script Code for a Tool

To create script code:

1. Click the **Script** button from the VectorScript Plug-in Editor dialog box.
2. Enter the script source code in the VectorScript Editor window.



3. When the script has been entered, click **OK** to save the script as part of the plug-in.

Working With Tool Items

Adding a Tool to a Workspace

Once a tool item plug-in has been created, it will need to be added to one or more workspaces to be available for use with VectorWorks. Once the tool has been added to the workspace, it will immediately be available for use in the current VectorWorks session.

To add a tool item to a VectorWorks workspace:

1. Select **File > Workspaces > Workspace Editor**.
2. In the list of available tools in the Workspace Editor, look for the category that was assigned to the tool item. Click the disclosure triangle to display the available items in the category.
3. Click and drag the tool item to the desired location on a tool palette. If desired, add a key equivalent for the tool item.
4. Click **OK** to save the workspace with the added tool item.

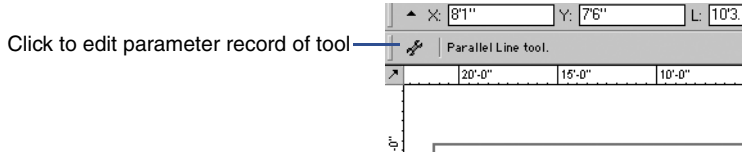
   Once the tool has been added, it should display the icon for the tool in the position where the tool was placed. To activate the tool item, simply click on it as if it were a standard VectorWorks tool.

For details on editing workspaces, see Appendix B in the VectorWorks User's Guide.

Setting Tool Item Defaults

VectorScript tool items can be integrated into a workspace and used in conjunction with other tools during a VectorWorks session. VectorScript tools can be selected from a tool palette, make use of the SmartCursor, and generally perform the same tasks as any standard VectorWorks tool.

Tools which have a defined parameter record will display a plug-in preferences button on the mode bar. This button provides access to the default parameter record of the tool.



Clicking the plug-in preferences button of the tool will display the preferences dialog box for the tool item. The default values stored in the parameter record will be displayed in the dialog box.

The parameter values can be edited as desired, then saved back to the parameter record by clicking **OK**. The new values will become the default settings for the tool on its next use.

VectorScript Point Objects

In this Chapter:

VectorScript parametric objects allow whole new classes of objects to be defined for use with VectorWorks. It is possible to create complex objects which can perform a wide array of tasks: standard architectural or mechanical elements, "smart" drawing components such as callouts or drawing borders, or other flexible objects which streamline the design process.

Parametric objects support standard VectorWorks core technologies such as snapping, classing, and advanced object editing. This support means that using parametric objects is no different than working with any of VectorWorks basic object types. VectorScript parametric objects are extremely flexible—object plug-ins can be created with up to 32,767 parameters for defining and editing the appearance of an object in the document. Objects are also portable; to share a parametric object, simply copy the object plug-in to the VectorWorks Plug-ins folder, add the object to a workspace, and the object is immediately available for use with VectorWorks.

The most basic type of VectorScript parametric object is the **point object**. Point objects are so named because they are defined by a single point click for placement in a VectorWorks document. This section documents the basic techniques needed to create and use point objects with VectorWorks.

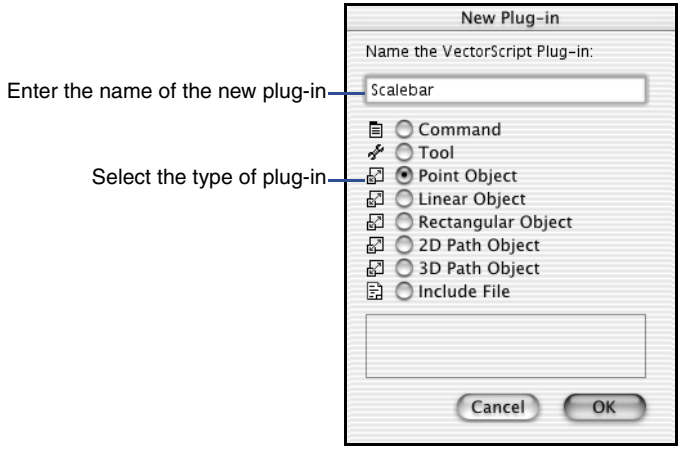
Creating a Point Object Plug-in

VectorScript parametric objects are created using the VectorWorks Plug-in Editor to create and define the actual plug-in item. The Plug-in Editor provides a single interface for creating the plug-in script and editing the associated settings objects will use when placed during a VectorWorks session.

Creating the Object Plug-in

To create a object plug-in:

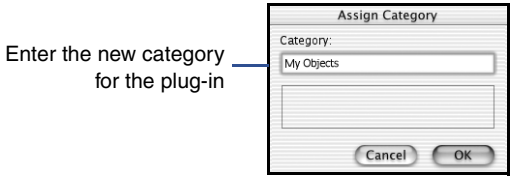
1. Select **Organize > Scripts > Create Plug-in**.
2. In the VectorScript Plug-in Editor dialog box, click **New**.
3. In the Assign Name dialog box, enter the name of the new plug-in item and select **Point Object** as the type of the plug-in. Click **OK** to create the plug-in item.



Setting the Object Category

To set the category:

1. Click the **Category** button from the VectorScript Plug-in Editor dialog box.
2. In the Assign Category dialog box, enter the name of the category to be associated with the new object plug-in item. Click **OK** to set the plug-in category to the new value.



The plug-in category is the heading under which the object may be found when selecting items in the Workspace Editor.

Setting Options for the Object

Point objects have a several settings which allow them to maintain behavior consistent with standard VectorWorks tools. These settings, also known as **plug-in properties**, control how the object is placed, as well as the various default values for the object when it is created.

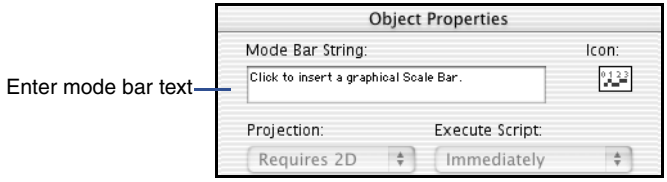
Setting Display Defaults for the Object

To set the display defaults:

1. Click the **Properties** button from the VectorScript Plug-in Editor dialog box.

   The Object Properties dialog box opens.

2. Enter the desired descriptive text to be displayed in the mode bar in the **Mode Bar String** field.



   Mode bar text usually includes the name of the tool, and can include text indicating the action the user should perform.

3. Click **OK** to save the new settings for the object.

Setting the Object Icon

VectorScript plug-ins come preloaded with a default icon which is displayed when the plug-in is placed in a workspace tool palette. This icon can be replaced with a custom icon indicating the function of the plug-in item.

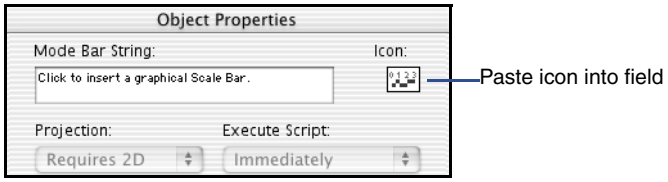
To create the icon, use a third-party icon editor. Tool icons use an 8-bit, 32-by-32 pixel field for defining the icon; the actual icon graphic, however, should be confined to an area 24 pixels wide by 18 pixels high, centered in the field.

To create the VectorScript object icon:

1. Click the **Properties** button from the VectorScript Plug-in Editor dialog box.

   The Object Properties dialog box opens.

2. Click in the icon display field to select it. The icon field is highlighted.



3. Paste the new object icon into the icon field. The customized icon is now visible in the icon field of the dialog box.

4. Click **OK** to save the new settings for the object.

Setting Activation Options for the Object

Activation options for objects control the conditions under which the object script code will execute. VectorWorks presets these options to the optimal configuration for object interaction with the VectorWorks application.

Setting the Default Class of the Object

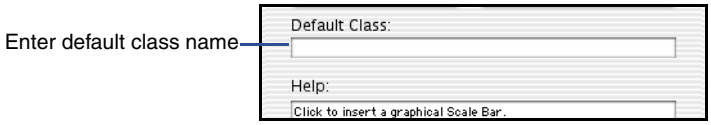
Objects can be specified to have a default class setting on insertion into a document. This preset default class allows objects to be automatically classed without the class being active, or requiring additional editing using the Object Info palette.

To define a default class for the object:

1. Click the **Properties** button from the VectorScript Plug-in Editor dialog box.

   The Object Properties dialog box opens.

2. Enter the desired class name in the default class field.



If the default class does not exist in the document when the object is placed, the class will be automatically created.

Setting Help Text for the Object

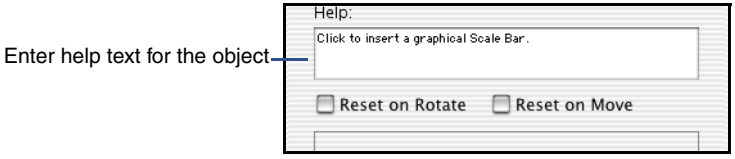
Help text describing the object will display when the cursor is held over the object icon in a tool palette.

To create help text for the object:

1. Click the **Properties** button from the VectorScript Plug-in Editor dialog box.

   The Object Properties dialog box opens.

2. Enter the desired help text in the **Help** field. The text will be displayed when help text is enabled and the cursor placed over the tool item.



3. Click **OK** to save the new settings for the object.

Setting Object Reset Options

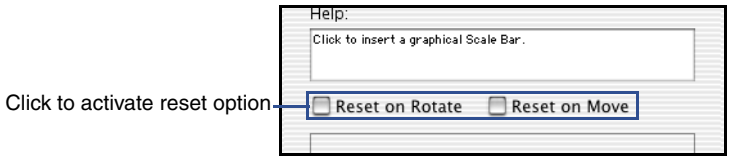
Objects have two properties which control when the geometry of the object will be recalculated and regenerated in the document. These properties are known as the **object reset options**.

By default, object geometry will only be recalculated if the object is edited via object parameters or control points. This is an important point, because when object geometry is recalculated, document default settings for attributes such as font, text size or line color will be reapplied to the object. If any of these settings have been modified since the object was placed or last edited, changes in the appearance of the object may occur. The default reset options allow objects to be manipulated without invoking object regeneration.

For instances where it is important that the object recalculate (for example, windows placed in a wall), objects can be optionally set to recalculate their geometry when the object has been moved or rotated.

To set the object reset options:

1. Click the **Properties** button from the VectorScript Plug-in Editor dialog box.
2. In the Object Properties dialog box, click on the desired object reset option to activate it.



   Selecting the **Reset on Rotate** option will cause the object to recalculate when the object is rotated in the document. Selecting the **Reset on Move** will cause the object to recalculate when the object is moved in either 2D or 3D, as well as when the object is cut and pasted into the document

3. Click **OK** to save the new settings for the object.

The new reset options for the object will take effect immediately.

Parameters and Point Objects

The parameters which define the appearance of a VectorScript point object are stored in a parameter record which is associated with each point object instance placed in the document. The parameters for each object instance may be modified by using the Object Info palette to access the values in the object parameter record.

A default parameter record is also created in the document when the first instance of an object is created in the active document. This default parameter record, which is distinct from the parameter records associated with object instances, stores the object default settings with the document. It is used when placing subsequent object instances to define the defaults for each new object instance.

Creating a Parameter Record for an Object

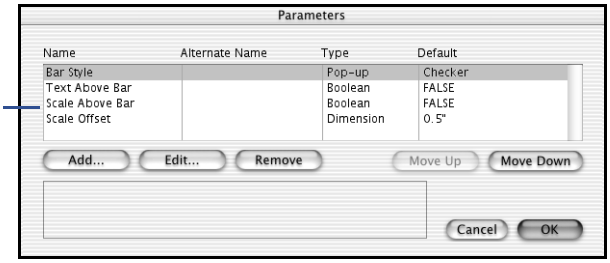
To create a parameter record:

1. Click the **Parameters** button from the VectorScript Plug-in Editor dialog box.

The Parameters dialog box opens.

2. Create the desired parameter record settings for the plug-in. For details on specific plug-in parameter types, see “Using VectorScript Plug-ins” on page 10-1.

Create parameters with default values for object



3. When all the desired parameters have been created, click **OK** to save the parameters.

When an object instance is placed in the document, the object’s parameter record can be edited using the Object Info palette.

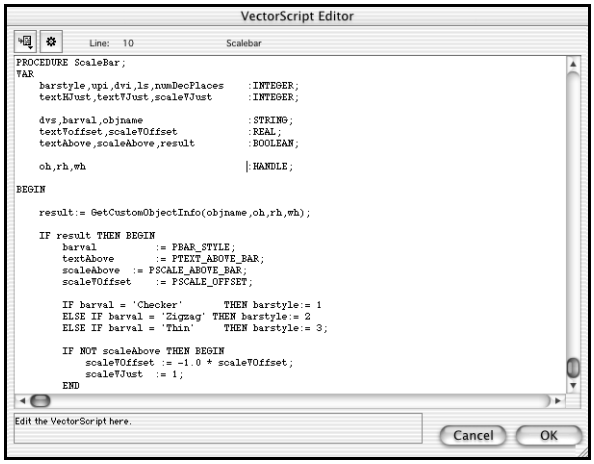
Creating the Object Script

The script source code for the object plug-in can be created using the VectorScript editor or a third-party text editor. The source code is saved as part of the plug-in item.

Creating Script Code for a Point Object

To create a script code:

1. Click the **Script** button from the VectorScript Plug-in Editor dialog box.
2. Enter the script source code in the VectorScript Editor window.



3. When the script has been entered, click **OK** to save the script as part of the plug-in.

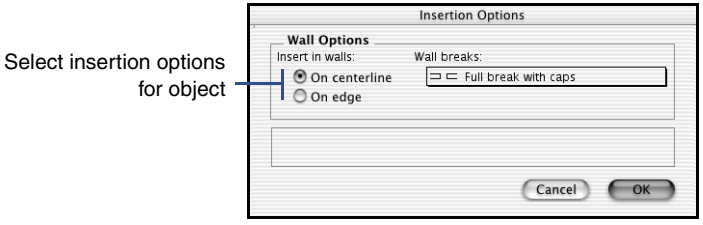
Setting Object Insertion Options

Objects, like symbols, can be assigned predefined insertion options for document placement. These options allow objects to properly interact with walls or other advanced VectorWorks object types.

Setting Insertion Options for a Point Object

To set insertion options:

1. Click the **Insert Options** button from the VectorScript Plug-in Editor dialog box.
2. In the Insertion Options dialog box, select the desired option settings for the object.



For objects which do not require insertion options, leave the options at the default settings.

Working with Point Objects

Working with objects incorporates elements of tool and symbol usage. Like tools, objects can be selected from tool palettes and make use of the SmartCursor, mode bar, and other core VectorWorks features. Like symbols, objects can be inserted into walls, and can optionally be configured to be available from the Object Browser.

Working effectively with point objects requires the knowledge of several basic techniques for managing and using objects with VectorWorks and VectorWorks documents.

Adding a Point Object to a Workspace

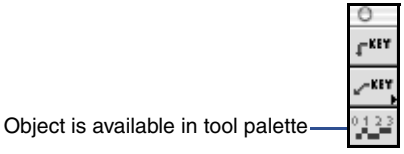
Once an object plug-in has been created, it will need to be added to one or more workspaces in order to be available for use with VectorWorks. Object plug-ins are functionally similar to tool items, and may be placed into tool palettes for use. Once the object has been added to the workspace, it will immediately be available for use in the current VectorWorks session.

To add an object to a VectorWorks workspace:

1. Select **File > Workspaces > Workspace Editor**.
2. Switch to the tools pane in the Workspace Editor, and then look for the category that was assigned to the new object plug-in. Click the disclosure triangle to display the available items in the category.
3. Click and drag the object plug-in to the desired location on a tool palette. If desired, add a key equivalent for the object.
4. Click **OK** to save the edited workspace.

For details on editing workspaces, see Appendix B in the VectorWorks User's Guide.

Once the object has been added, the object should be visible on the tool palette. To activate the object, click on the item as if it were a standard VectorWorks tool.



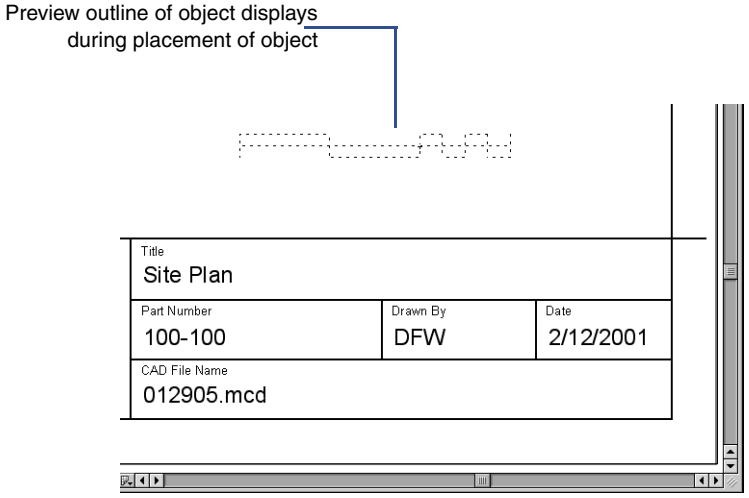
Placing Objects in Documents

Placing objects in documents is very similar to using any other VectorWorks tool. Objects, like VectorWorks tools, make use of the SmartCursor, mode bar, and constraints to allow precise placement of the object. Once placed, the object can be edited using the Object Info palette, and can be manipulated in much the same way as any of the basic VectorWorks object types.

Point objects, as their name indicates, are placed by defining a single point location in the document which is the insertion point for the object. The insertion point of the object corresponds to the origin `(0,0)` of the object definition.

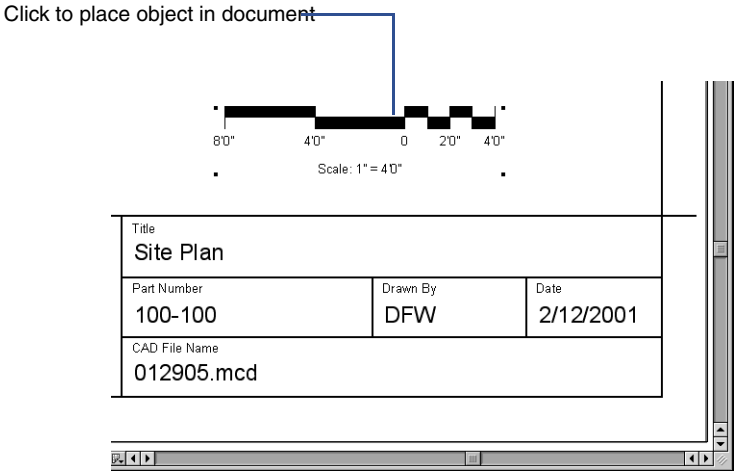
To place a point object in a document:

1. Open the tool palette containing the object and click the icon of the object to activate it.
2. Select the desired insertion mode, and if the object is to be inserted into a wall, optionally select the appropriate wall mode from the mode bar.
3. Move the cursor to the desired insertion location for the object.



   The cursor should drag a preview outline of the object to be placed, and respond to any active constraints.

4. Click to define the insertion location and place the object.



When the first instance of an object is placed in the document, the plug-in preferences dialog box will display, allowing default settings for subsequent object placement to be modified before the default parameter record is created in the document.

Default object parameter settings can be accessed at any time from the plug-in preferences button displayed in the mode bar when the object palette icon is selected.

Editing Objects in the Document

Once an object is placed in a document, the settings for the specific document can be modified at any time by selecting the object instance and editing the object through the Object Info palette.

To edit an object instance in the document:

1. Select the object instance to be edited.
2. Select **Window > Palettes > Object Info** to open the Object Info palette.
3. Edit the object instance as desired.

   The object specific parameters will be listed below a set of basic object editing controls which is available for any object.

Using Point Objects with the Resource Browser

The VectorWorks Resource Browser provides a streamlined process for browsing and selecting both symbols and objects for placement in a VectorWorks document. Browser items can be selected from the currently open document, or from other documents which act as "libraries" and which can be saved as Favorites. The Resource Browser interface provides an important tool for working with plug-in objects in VectorWorks.

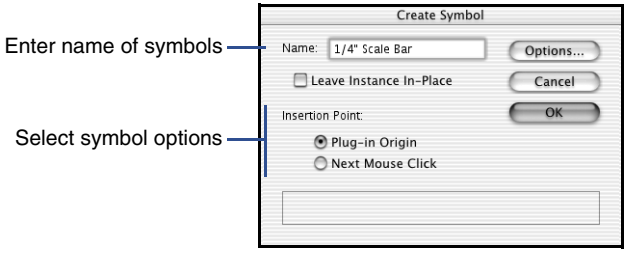
There are several methods for making point objects available through the Resource Browser. Objects can be saved as a regular (static) symbol, and can then placed in the document as needed. Objects can also be saved as **object symbols** which revert back to a plug-in when placed. Unlike static symbols, object symbols allow objects to be stored with any specified parameter combination as a default, while still retaining the ability for the object to be edited at any time after placement. Combinations of objects can also be saved as a **group symbol**, where the symbol is converted to a group on placement in the document. Group symbols, like object symbols, allow the component objects to retain their editable characteristics, but group the component items into a single entity for ease of editing.

Creating Static Symbols with Objects

To create static symbols:

1. Place an instance of the object to be saved in the active document.
2. Specify the desired parameters for the object.

   The parameters will be preserved and the object will retain the appearance that was displayed when the symbol is created.
3. Select **Organize > Create Symbol**. Enter the name for the new symbol and select the desired options.



4. Click **OK** to create the symbol.

The new symbol appears in the Resource Browser. If it is not visible, check to make sure the active document is selected for browsing.

The symbol can now be placed by clicking on the icon, and then selecting **Make Active** from the **Resources** menu.

An object that is contained within a static symbol can be reverted to a plug-in object. To revert the object:

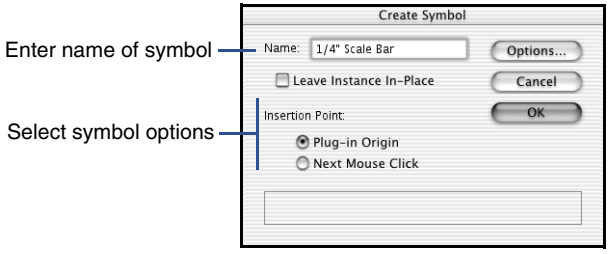
1. Select the symbol instance and choose **Organize > Convert to Plug-in**.
2. Select the appropriate conversion options and click **OK**. The symbol will revert to a plug-in object.

Creating Object Symbols

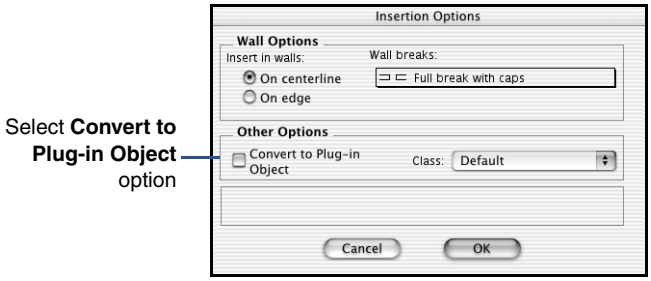
To create object symbols:

1. Place an instance of the object to be saved in the active document.
2. Specify the desired parameters for the object.

   The parameters will be preserved and the object will retain the appearance that was displayed when the symbol is created.
3. Select **Organize > Create Symbol**. Enter the name for the new symbol and select the desired options.



4. Click **Options** and select the **Convert to Plug-in** option in the Insertion Options dialog box. Select any other options appropriate for the object symbol, and then click **OK** to save the options.



5. Click **OK** to create the object symbol

The new symbol appears in the Resource Browser. If it is not visible, check to make sure the active document is selected for browsing.

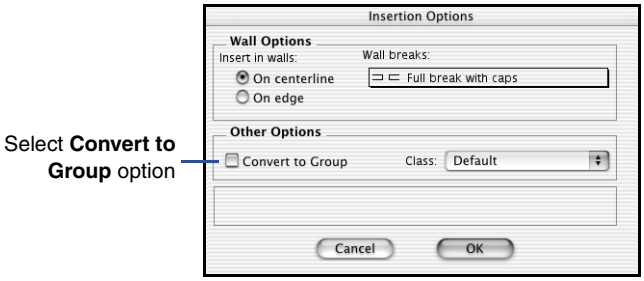
Note that an object symbol differs in appearance from static symbols, with the object symbol appearing as a red icon.

The symbol can now be placed by clicking on the icon, and then selecting **Make Active** from the **Resources** menu. When placed, the symbol will automatically revert to a plug-in object. The parameter configuration saved when the object symbol was created will act as the default settings for the object. The object parameter settings can be edited as desired after placement.

Creating Group Symbols with Objects

To create group symbols:

1. Place the items that will be components of the group symbol in the active document.
2. Specify the desired parameters for any objects, then select all the items to be included in the group symbol. Any objects will retain their displayed appearance when the group symbol is created.
3. Select **Organize > Create Symbol**. Enter the name for the new group symbol and select the desired options.
4. Click **Options** and select the **Convert to Group** option in the Insertion Options dialog. Select any other options appropriate for the object symbol, and then click **OK** to save the options.



5. Click **OK** to create the group symbol.

The new group symbol appears in the Resource Browser. If it is not visible, check to make sure the active document is selected for browsing.

Note that a group symbol also differs in appearance from static symbols, with the object symbol appearing as a blue icon.

The group symbol can now be placed by clicking on the icon, and then selecting **Make Active** from the **Resources** menu. When placed, the symbol will automatically convert to an object group. Any objects in the group can be modified by entering the group and editing the objects as desired.

VectorScript Linear Objects

In this Chapter:

VectorScript **linear objects** are the second of the four plug-in object types available in VectorWorks. Linear objects, like point objects, are named according to how they are defined in the VectorWorks document. Linear objects require a user-defined line to create the basic geometry of the object.

Like all other objects, linear objects support the standard VectorWorks core technologies, and behave similarly to VectorWorks basic object types. Linear objects can be edited using the Object Info palette, but can also be edited on-screen using the SmartCursor to modify the object instance

This section documents the basic techniques needed to create and use linear objects in your VectorWorks documents.

Creating a Linear Object Plug-in

VectorScript parametric objects are created using the VectorWorks Plug-in Editor to create and define the actual plug-in item. The Plug-in Editor provides a single interface for creating the plug-in script and editing the associated settings objects will use when placed during a VectorWorks session.

Creating the Object Plug-in

To create an object plug-in:

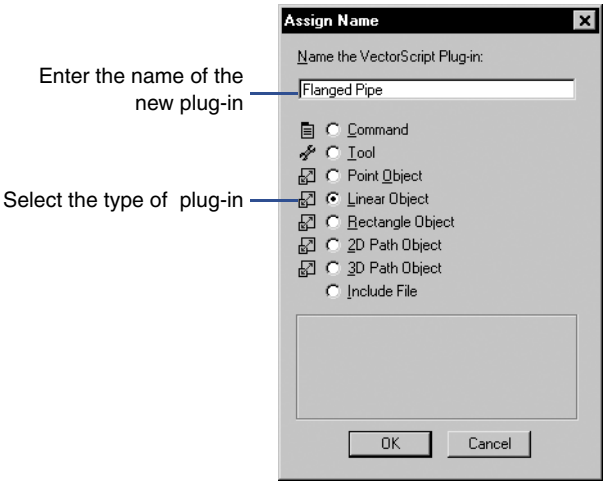
1. Select **Organize > Scripts > Create Plug-in**.

   The VectorScript Plug-in Editor dialog box opens.

2. Click the **New** button.

The Assign Name dialog box opens.

3. Enter the name of the new plug-in item and select **Linear Object** as the type of the plug-in. Click **OK** to create the plug-in.



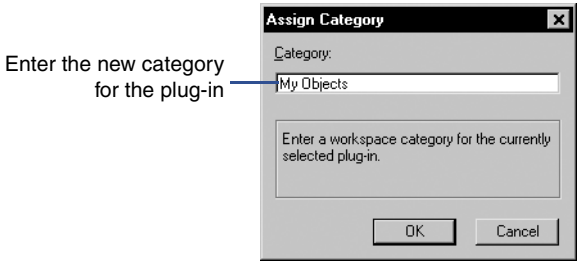
Setting the Object Category

To set the object category:

1. Click the **Category** button from the VectorScript Plug-in Editor dialog box.

   The Assign Category dialog box opens.

2. Enter the name of the category to be associated with the new object plug-in item. Click **OK** to set the plug-in category to the new value.



The plug-in category is the heading under which the object may be found when selecting items in the Workspace Editor.

Setting Options for the Object

Linear objects have several settings which allow them to maintain behavior consistent with standard VectorWorks tools. These settings, also known as **plug-in properties**, control how the object is placed, as well as the various default values for the object when it is created.

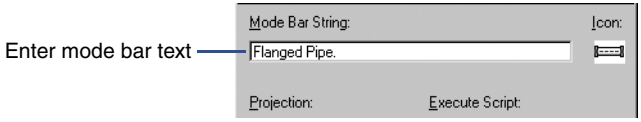
Setting Display Defaults for the Object

To set the display defaults:

1. Click the **Properties** button from the VectorScript Plug-in Editor dialog box.

   The Object Properties dialog box opens.

2. Enter the desired descriptive text to be displayed in the mode bar in the **Mode Bar String** field.



   Mode bar text usually includes the name of the tool, and can include text indicating the action the user should perform.

3. Click **OK** to save the new settings for the object.

Setting the Object Icon

VectorScript plug-ins come preloaded with a default icon which is displayed when the plug-in is placed in a workspace tool palette. This icon can be replaced with a custom icon indicating the function of the plug-in item.

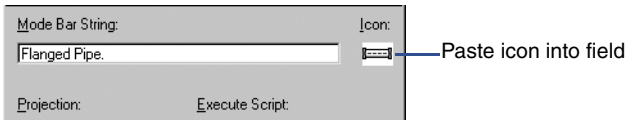
To create the icon, use a third-party icon editor. Tool icons use an 8-bit, 32-by-32 pixel field for defining the icon; the actual icon graphic, however, should be confined to an area 24 pixels wide by 18 pixels high, centered in the field.

To create the VectorScript object icon:

1. Click the **Properties** button from the VectorScript Plug-in Editor dialog box.

   The Object Properties dialog box opens.

2. Click in the icon display field to select it. The icon field is highlighted.



3. Paste the new object icon into the icon field. The customized icon should be visible in the icon field of the dialog box.
4. Click **OK** to save the new settings for the object.

Setting Activation Options for the Object

Activation options for objects control the conditions under which the object script code will execute. VectorWorks presets these options to the optimal configuration for object interaction with the VectorWorks application.

Setting the Default Class of the Object

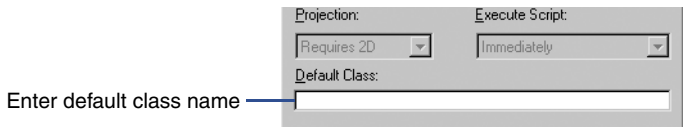
Objects can be specified to have a default class setting on insertion into a document. This preset default class allows objects to be automatically classed without the class being active, or requiring additional editing using the Object Info palette.

To define a default class for the object:

1. Click the **Properties** button from the VectorScript Plug-in Editor dialog box.

   The Object Properties dialog box opens.

2. Enter the desired class name in the default class field.



If the default class does not exist in the document when the object is placed, the class will be automatically created

Setting Help Text for the Object

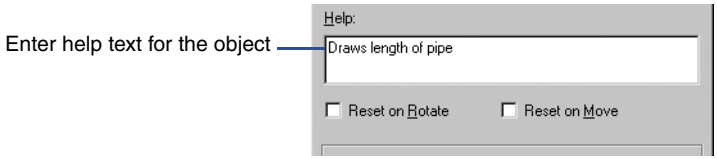
Help text describing the object will display when the cursor is held over the object icon in a tool palette.

To create help text for the object:

1. Click the **Properties** button from the VectorScript Plug-in Editor dialog box.

   The Object Properties dialog box opens.

2. Enter the desired help text in the **Help** field. The text will be displayed when help text is enabled and the cursor placed over the tool item.



3. Click **OK** to save the new settings for the object.

Setting Object Reset Options

Objects have two properties which control when the geometry of the object will be recalculated and regenerated in the document. These properties are known as the **object reset options**.

By default, object geometry will only be recalculated if the object is edited by object parameters or control points. This is an important point, because when an object geometry is recalculated, document default settings for attributes such as font, text size or line color will be reapplied to the object. If any of these settings have been modified since the object was placed or last edited, changes in the appearance of the object may occur. The default reset options allow objects to be manipulated without invoking object regeneration.

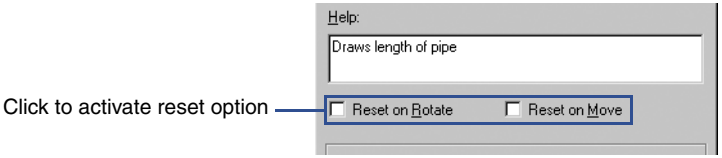
For instances where it is important that the object recalculate (for example, windows placed in a wall), objects can be optionally set to recalculate their geometry when the object has been moved or rotated

To set the object reset options:

1. Click the **Properties** button from the VectorScript Plug-in Editor dialog box.

   The Object Properties dialog box opens.

2. Click the desired reset option to activate it.



Selecting the **Reset on Rotate** option will cause the object to recalculate when the object is rotated in the document. Selecting **Reset on Move** will cause the object to recalculate when the object is moved in either 2D or 3D, as well as when the object is cut and pasted into the document.

3. Click **OK** to save the new settings for the object.

   The new reset options for the object will take effect immediately.

Parameters and Linear Objects

The parameters which define the appearance of a VectorScript linear object are stored in a parameter record which is associated with each point object instance placed in the document. The parameters for each object instance may be modified by using the Object Info palette to access the values in the object parameter record.

A default parameter record is also created in the document when the first instance of an object is created in the active document. This default parameter record, which is distinct from the parameter records associated with object instances, stores the object default settings with the document. It is used when placing subsequent object instances to define the defaults for each new object instance.

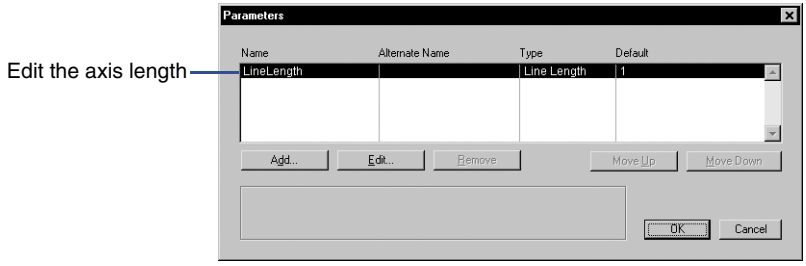
Parameter records for linear objects contain a predefined parameter, LineLength, which contains the length of the axis which defines the linear object. This predefined parameter may be edited, but cannot be deleted.

Creating a Parameter Record for an Object

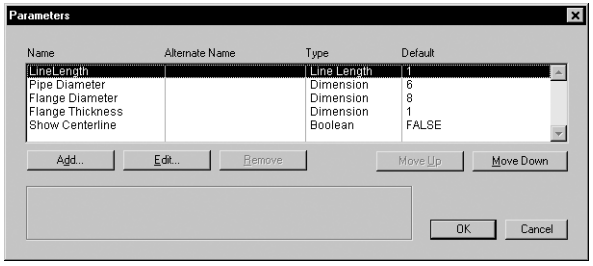
To create a parameter record:

1. Click the **Parameters** button from the VectorScript Plug-in Editor dialog box.

2. The Parameters dialog box should display the predefined LineLength parameter which contains the axis length of the linear object. If desired, specify the a new default value for the object axis.



3. Create the desired parameter record settings for the plug-in. For details on specific plug-in parameter types, see “Using VectorScript Plug-ins” on page 10-1.



Create parameters with default values for object

4. When all the parameters have been created, click **OK** to save the parameters.

When an object instance is placed in the document, the object’s parameter record can be edited using the Object Info palette.

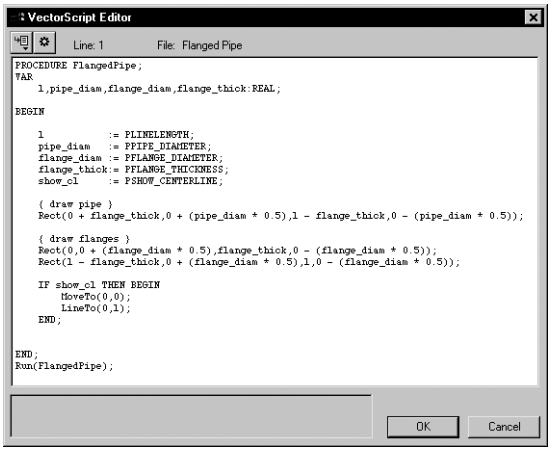
Creating the Object Script

The script source code for the object plug-in can be created using the VectorScript editor or a third-party text editor. The source code is saved as part of the plug-in item.

Creating Script Code for a Linear Object

To create script code:

1. Click the **Script** button from the VectorScript Plug-in Editor dialog box.
2. Enter the script source code in the VectorScript Editor window.



3. When the script has been entered, click **OK** to save the script as part of the plug-in.

Setting Object Insertion Options

Objects, like symbols, can be assigned predefined insertion options for document placement. These options allow objects to properly interact with walls or other advanced VectorWorks object types.

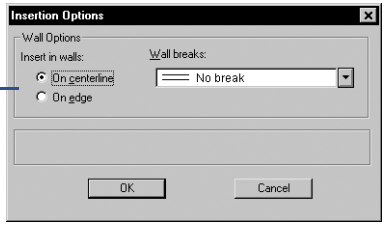
Setting Insertion Options for a Linear Object

To set insertion options:

1. Click the **Insert Options** button from the VectorScript Plug-in Editor dialog box.

   The Insertion Options dialog box opens.

2. Select the desired option settings for the object.



For objects which do not require specific insertion options, leave the options at the default settings.

Working with Linear Objects

Working with objects incorporates elements of tool and symbol usage. Like tools, objects can be selected from tool palettes and make use of the SmartCursor, mode bar, and other core VectorWorks features. Like symbols, objects can be inserted into walls, and can optionally be configured to be available from the Resource Browser.

Working effectively with point objects requires the knowledge of several basic techniques for managing and using objects with VectorWorks and VectorWorks documents.

Adding a Linear Object to a Workspace

Once an object plug-in has been created, it will need to be added to one or more workspaces in order to be available for use with VectorWorks. Object plug-ins are functionally similar to tool items, and may be placed into tool palettes for use. Once the object has been added to the workspace, it will immediately be available for use in the current VectorWorks session.

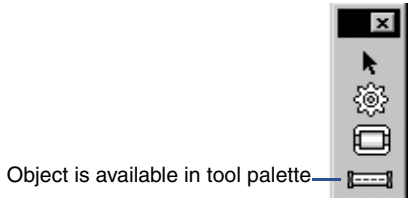
To add an object to a VectorWorks workspace:

1. Select **File > Workspaces > Workspace Editor**.
2. Switch to the tools pane in the Workspace Editor.
3. Locate the category that was assigned to the new object plug-in. Click the disclosure triangle to display the available items in the category.
4. Click and drag the object plug-in to the desired location on a tool palette. If desired, add a key equivalent for the object.

   Click **OK** to save the edited workspace.

For details on editing workspaces, see Appendix B in the VectorWorks User's Guide.

Once the object has been added, the object should be visible on the tool palette. To activate the object, click on the item as if it were a standard VectorWorks tool.



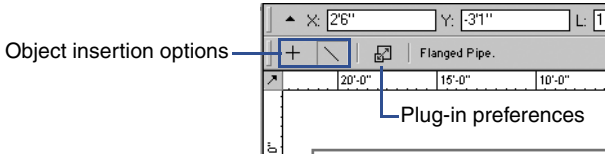
Placing Objects in Documents

Placing linear objects in documents is very similar to using any other VectorWorks tool. Objects, like VectorWorks tools, make use of the SmartCursor, mode bar, and constraints to allow precise placement of the object. Once placed, linear objects can be edited using the Object Info palette, or they can be edited using the SmartCursor to resize the object along its definition axis.

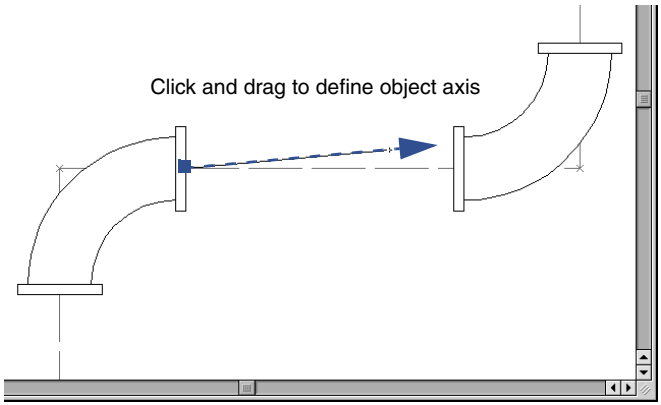
The insertion point of a linear object corresponds to the first click defining the object axis, and acts as the origin (0,0) of the object definition.

To place a linear object in a document:

1. Open the tool palette containing the object and click the icon of the object to activate it.
2. Select the desired insertion mode for the linear object.

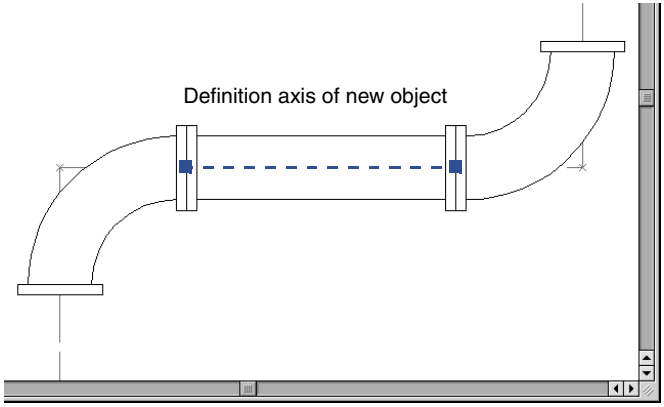


3. Move the cursor to the desired insertion location for the object, and then click to define the first point of the object definition axis.



The cursor should drag a preview line from the first point of the object axis, and respond to any active constraints.

4. Click to define the second point of the object axis and create the object.



When the first instance of an object is placed in a document, the plug-in preferences dialog box will be displayed. This dialog allows the settings for the default parameter record to be modified before it is created in the document. Subsequent object instances will use the modified settings as their defaults.

Default object parameter settings can be accessed at any time from the plug-in preferences button displayed in the mode bar when the object palette icon is selected.

Editing Linear Objects in the Document

Once a linear object is placed in a document, the settings for the specific object instance can be modified at any time by selecting the object and editing it through the Object Info palette. Linear objects may also be modified by redefining the object axis with the SmartCursor.

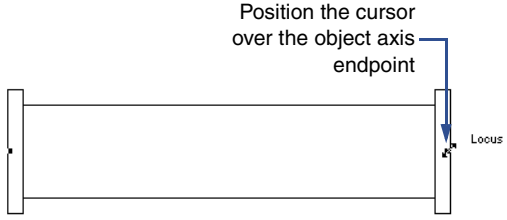
To edit an object instance in the document:

1. Select the object instance to be edited.
2. Select **Window > Palettes > Object Info** to open the Object Info palette.
3. Edit the object instance as desired.

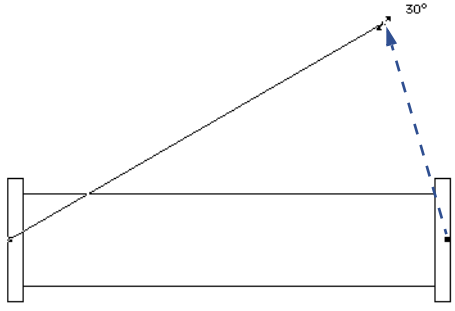
The object specific parameters will be listed below a set of basic object editing controls which are available for any object.

To edit the object using the object definition axis:

1. Choose the selection tool, then select the object instance to be edited.
2. Move the cursor over the object axis point to be modified and hold until the cursor changes to the reshape cursor.



3. Click on the axis point and drag to the desired location.



Drag axis point to new location

4. Release the mouse to define the new axis endpoint location.

Using Linear Objects with the Resource Browser

There are several methods for making linear objects available through the Resource Browser. Like point objects, linear objects can be saved as a regular (static) symbol, as an object symbol, or as part of a group symbol.

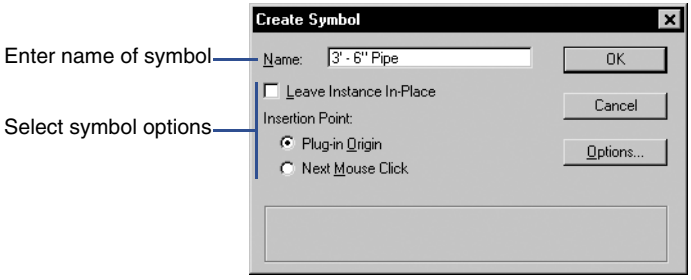
Creating Static Symbols with Linear Objects

To create static symbols:

1. Place an instance of the object to be saved in the active document.
2. Specify the desired parameters for the object.

   The parameters will be preserved and the object will retain the appearance that was displayed when the symbol is created.
3. Select **Organize > Create Symbol**.

   The Create Symbol dialog box opens.
4. Enter the name for the new symbol and select the desired options.



   The plug-in origin for a linear object will be the first object axis definition point. Linear objects in static symbols cannot be sized during placement.
5. Click **OK** to create the symbol.

The new symbol appears in the Resource Browser. If it is not visible, check to make sure the active document is selected for browsing.

The symbol can now be placed by clicking on the icon, and then selecting **Make Active** from the **Resources** menu.

An object that is contained within a static symbol can still be reverted to a plug-in object.

To revert the object:

1. Select the symbol instance and choose **Organize > Convert to Plug-in**.
2. Select the appropriate conversion options and click **OK**. The symbol will revert to a plug-in object.

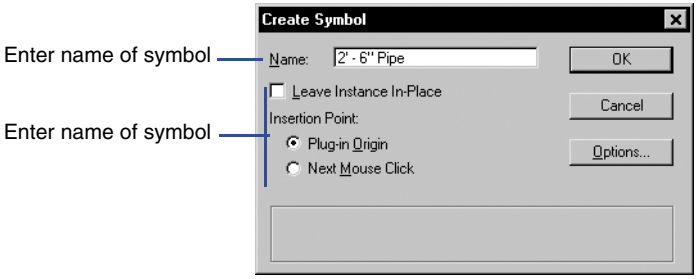
Creating Object Symbols with Linear Objects

To create object symbols:

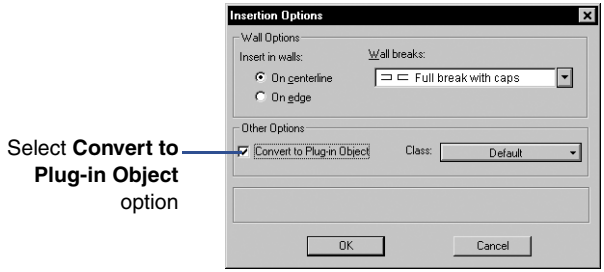
1. Place an instance of the object to be saved in the active document.
2. Specify the desired parameters for the object.

   The parameters will be preserved and the object will retain the appearance that was displayed when the symbol is created.
3. Select **Organize > Create Symbol.**

   The Create Symbol dialog box opens.
4. Enter the name for the new symbol and select the desired options.



5. Click **Options** and select the **Convert to Plug-in** option in the Insertion Options dialog. Select any other options appropriate for the object symbol, and then click **OK** to save the options.



Select **Convert to Plug-in Object** option

6. Click **OK** to create the object symbol

The new symbol appears in the Resource Browser. If it is not visible, check to make sure the active document is selected for browsing.

Note that an object symbol differs in appearance from static symbols, with the object symbol appearing as a red icon.

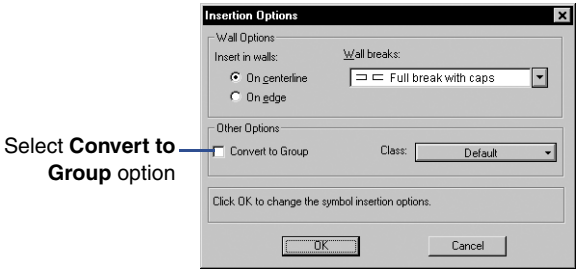
The symbol can now be placed by clicking on the icon and selecting **Make Active** from the **Resources** menu. When placed, the linear object symbols will automatically switch to the creation mode of the object, allowing the object axis to be defined. The new linear object instance will then be created in the document.

Creating Group Symbols with Linear Objects

To create group symbols:

1. Place the items that will be components of the group symbol in the active document.
2. Specify the desired parameters for any objects, and then select all the items to be include in the group symbol. Any objects will retain their displayed appearance when the group symbol is created.
3. Select **Organize > Create Symbol**.

   The Create Symbol dialog box opens.
4. Enter the name for the new group symbol and select the desired options.
5. Click **Options** and then **Convert to Group** in the Insertion Options dialog box. Select any other options appropriate for the object symbol, and then click **OK** to save the options.



6. Click **OK** to create the group symbol.

The new group symbol appears in the Resource Browser. If it is not visible, check to make sure the active document is selected for browsing.

Group symbol symbols differ in appearance from static symbols, with the symbol appearing as a blue icon.

The group symbol can now be placed by clicking on the icon, and then selecting **Make Active** from the **Resources** menu. When placed, the symbol will automatically convert to an object group. Any objects in the group can be modified by entering the group and editing the objects as desired.

VectorScript Rectangular Objects

In this Chapter:

VectorScript **rectangular objects** are the third of the four plug-in object types available in VectorWorks. Rectangular objects, like the other object types already introduced, are named according to how they are defined in the VectorWorks document. Rectangular objects utilize a user-defined rectangle to define and create the basic geometry of the object.

Rectangular objects support the standard VectorWorks core technologies, and behave similarly to the basic VectorWorks object types. Rectangular objects can be edited using the Object Info palette or they can be edited on-screen using the SmartCursor to modify the object instance.

This section documents the basic techniques needed to create and use rectangular objects in VectorWorks documents.

Creating a Rectangular Object Plug-in

VectorScript parametric objects are created using the VectorWorks Plug-in Editor to create and define the actual plug-in item. The Plug-in Editor provides a single interface for creating the plug-in script and editing the associated settings object will use when placed during a VectorWorks session.

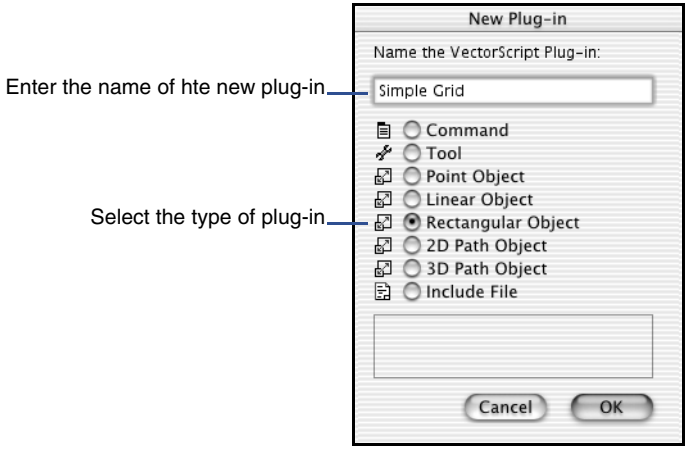
Creating the Object Plug-in

To create an object plug-in:

1. Select **Organize > Scripts > Create Plug-in**.
2. In the VectorScript Plug-in Editor dialog box, click **New**.

The Assign Name dialog box opens.

3. Enter the name of the new plug-in item and select **Rectangular Object** as the type of the plug-in. Click **OK** to create the plug-in item.



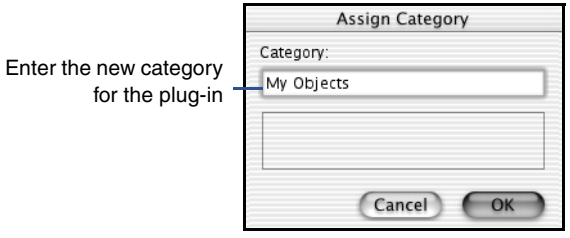
Setting the Object Category

To set the object category:

1. Click the **Category** button from the VectorScript Plug-in Editor dialog box.

   The Assign Category dialog box opens.

2. Enter the name of the category to be associated with the new object plug-in item. Click **OK** to set the plug-in category to the new value.



The plug-in category is the heading under which the object may be found when selecting items in the Workspace Editor.

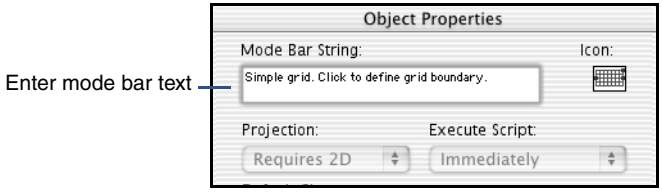
Setting Options for the Object

Rectangular objects have several settings which allow them to maintain behavior consistent with standard VectorWorks tools. These settings, also known as **plug-in properties**, control how the object is placed, as well as the various default values for the object when it is created.

Setting Display Defaults for the Object

To set the display defaults:

1. Click the **Properties** button from the VectorScript Plug-in Editor dialog box.
2. In the Object Properties dialog box, enter the desired descriptive text to be displayed in the mode bar in the **Mode Bar String** field.



   Mode bar text usually includes the name of the tool, and can include text indicating the action the user should perform.

3. Click **OK** to save the new settings for the object.

Setting the Object Icon

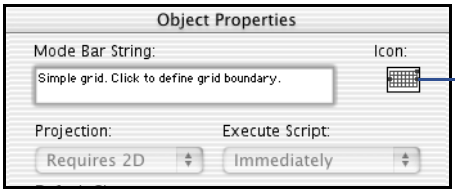
VectorScript plug-ins come preloaded with a default icon which is displayed when the plug-in is placed in a workspace tool palette. This icon can be replaced with a custom icon indicating the function of the plug-in item.

To create the icon, use a third-party icon editor. Tool and object icons use an 8-bit, 32-by-32 pixel field for defining the icon; the actual icon graphic, however, should be confined to an area 24 pixels wide by 18 pixels high, centered in the field.

To create the VectorScript object icon:

1. Click the **Properties** button from the VectorScript Plug-in Editor dialog box.

2. In the Object Properties dialog box, click in the icon display field to select it. The icon field is highlighted.



3. Paste the new object icon into the icon field. The customized icon should be visible in the icon field of the dialog box.
4. Click **OK** to save the new settings for the object.

Setting Activation Options for the Object

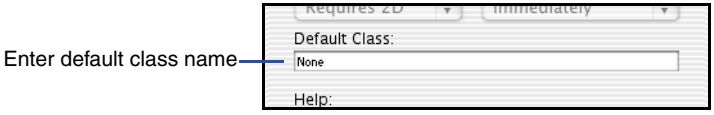
Activation options for objects control the conditions under which the object script code will execute. VectorWorks presets these options to the optimal configuration for object interaction with the VectorWorks application.

Setting the Default Class of the Object

Objects can be specified to have a default class setting on insertion into a document. This preset default class allows objects to be automatically classed without the class being active, or requiring additional editing using the Object Info palette.

To define a default class for the object:

1. Click the **Properties** button from the VectorScript Plug-in Editor dialog box.
2. In the Object Properties dialog box, enter the desired class name in the default class field.



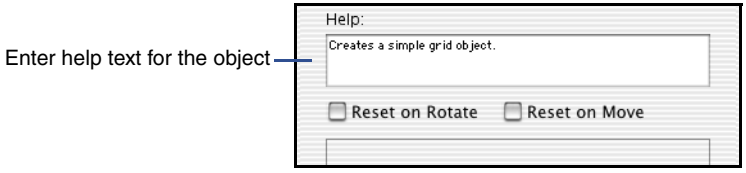
If the default class does not exist in the document when the object is placed, the class will be automatically created

Setting Help Text for the Object

Help text describing the object will display when the cursor is held over the object icon in a tool palette.

To create help text for the object:

1. Click the **Properties** button from the VectorScript Plug-in Editor dialog box.
2. In the Object Properties dialog box, enter the desired help text in the **Help** field. The text will be displayed when help text is enabled and the cursor placed over the tool item.



3. Click **OK** to save the new settings for the object.

Setting Object Reset Options

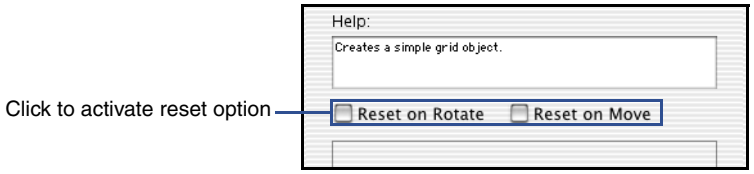
Objects have two properties which control when the geometry of the object will be recalculated and regenerated in the document. These properties are known as the **object reset options**.

By default, object geometry will only be recalculated if the object is edited using object parameters or control points. This is important, because when object geometry is recalculated, the document default settings for attributes such as font, text size or line color will be reapplied to the object. If any of these settings has been modified since the object was placed or last edited, changes in the appearance of the object may occur. The default reset options allow objects to be manipulated without invoking object regeneration.

For instances where it is important that the object recalculate (for example, windows placed in a wall), objects can be optionally set to recalculate their geometry when the object has been moved or rotated

To set the object reset options:

1. Click the **Properties** button from the VectorScript Plug-in Editor dialog box.
2. In the Object Properties dialog box, click the desired object reset option to activate it.



Selecting the **Reset on Rotate** option will cause the object to recalculate when the object is rotated in the document. Selecting **Reset on Move** will cause the object to recalculate when the object is moved in either 2D or 3D, as well as when the object is cut and pasted into the document

3. Click **OK** to save the new settings for the object.

   The new reset options for the object will take effect immediately.

Parameters and Rectangular Objects

The parameters which define the appearance of a VectorScript rectangular object are stored in a parameter record which is associated with each point object instance placed in the document. The parameters for each object instance may be modified by using the Object Info palette to access the values in the object parameter record.

A default parameter record is also created in the document when the first instance of an object is created in the active document. This default parameter record, which is distinct from the parameter records associated with object instances, stores the object default settings with the document. It is used when placing subsequent object instances to define the defaults for each new object instance.

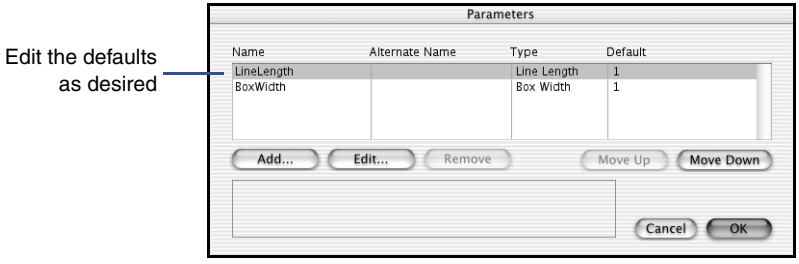
Parameter records for rectangular objects contain two predefined parameters, `LineLength` and `BoxWidth`. `LineLength` contains the length measured between the first and second mouse click during object placement. This value is used to define the initial length of the object instance. The second parameter, `BoxWidth`, is contains the length as measured between the second and third mouse click defined during object placement. The `BoxWidth` value is used to define the initial width of the rectangular object instance.

This predefined parameters for rectangular objects may be edited, but cannot be deleted.

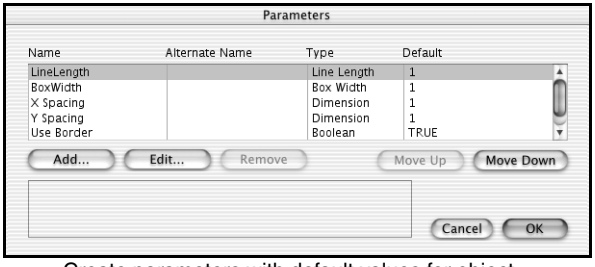
Creating a Parameter Record for an Object

To create a parameter record:

1. Click the **Parameters** button from the VectorScript Plug-in Editor dialog box.
2. The Parameters dialog box should display the predefined `LineLength` and `BoxWidth` parameters for the rectangular object. If desired, specify new default values for these parameters.



3. Create the desired parameter record settings for the plug-in. For details on specific plug-in parameter types, see “Using VectorScript Plug-ins” on page 10-1.



Create parameters with default values for object

4. When all the desired parameters have been created, click **OK** to save the parameters.

When an object instance is placed in the document, the object’s parameter record can be edited using the Object Info palette.

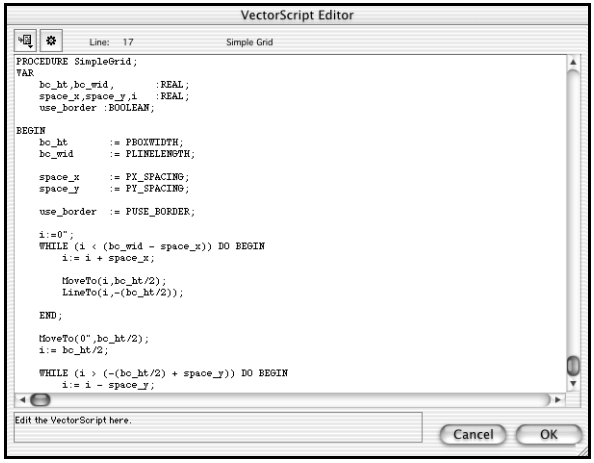
Creating the Object Script

The script source code for the object plug-in can be created using the VectorScript editor or a third-party text editor. The source code is saved as part of the plug-in item.

Creating Script Code for a Rectangular Object

To create script code:

1. Click the **Script** button from the VectorScript Plug-in Editor dialog box.
2. Enter the script source code in the VectorScript Editor window.



3. When the script has been entered, click **OK** to save the script as part of the plug-in.

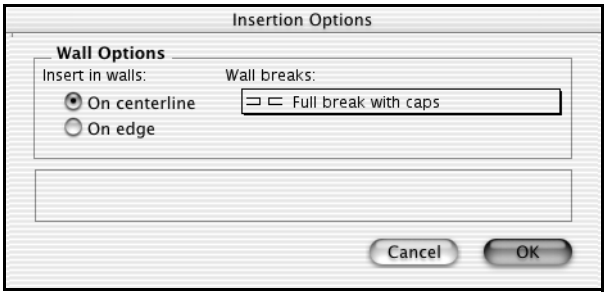
Setting Object Insertion Options

Objects, like symbols, can be assigned predefined insertion options for document placement. These options allow objects to properly interact with walls or other advanced VectorWorks object types.

Setting Insertion Options for a Rectangular Object

1. Click the **Insert Options** button from the VectorScript Plug-in Editor dialog box.

2. In the Insertion Options dialog box, select the desired option settings for the object.



Select insertion options for object

For objects which do not require specific insertion options, leave the options at the default settings.

Working with Rectangular Objects

Working with objects incorporates elements of tool and symbol usage. Like tools, objects can be selected from tool palettes and make use of the SmartCursor, mode bar, and other core VectorWorks features. Like symbols, objects can be inserted into walls, and can optionally be configured to be available from the Resource Browser.

Working effectively with rectangular objects requires the knowledge of several basic techniques for managing and using objects with VectorWorks and VectorWorks documents.

Adding a Rectangular Object to a Workspace

Once an object plug-in has been created, it will need to be added to one or more workspaces in order to be available for use with VectorWorks. Object plug-ins are functionally similar to tool items, and may be placed into tool palettes for use. Once the object has been added to the workspace, it will immediately be available for use in the current VectorWorks session.

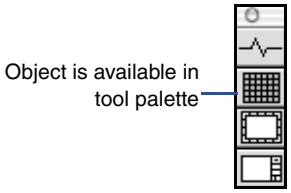
To add an object to a VectorWorks workspace:

1. Select **File > Workspaces > Workspace Editor**.

For details on editing workspaces, see Appendix B in the VectorWorks User's Guide.

2. Switch to the tools pane in the Workspace Editor, then look for the category that was assigned to the new object plug-in. Click the disclosure triangle to display the available items in the category.
3. Click and drag the object plug-in to the desired location on a tool palette. If desired, add a key equivalent for the object.
4. Click **OK** to save the edited workspace.

Once the object has been added, the object should be visible on the tool palette. To activate the object, click on the item as if it were a standard VectorWorks tool.



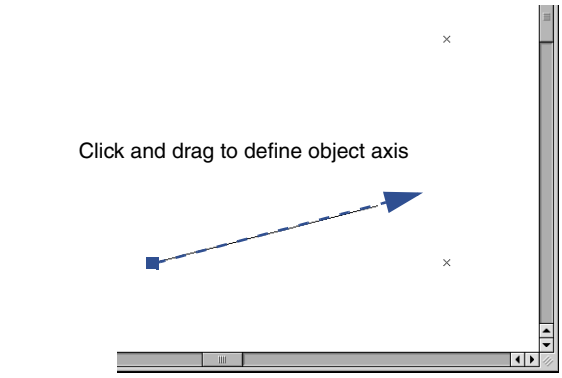
Placing Objects in Documents

Placing rectangular objects in documents is very similar to using any other VectorWorks tool. Objects, like VectorWorks tools, make use of the SmartCursor, mode bar, and constraints to allow precise placement of the object. Once placed, rectangular objects can be edited using the Object Info palette, or they can be edited using the SmartCursor to resize the object from any object handle.

The insertion point of a rectangular object corresponds to the first click defining the object axis, and acts as the origin `(0,0)` of the object definition.

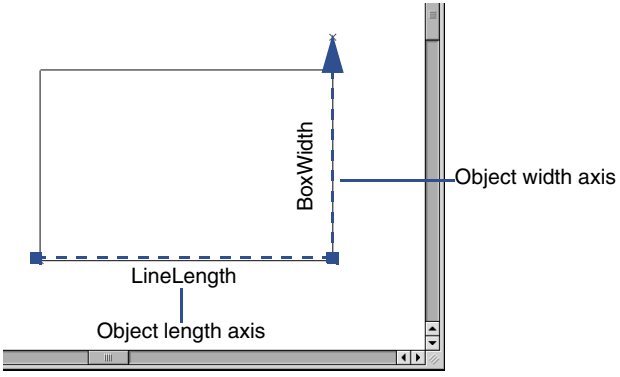
To place a rectangular object in a document:

1. Open the tool palette containing the object and click the icon of the object to activate it.
2. Select the desired insertion mode for the rectangular object.
3. Move the cursor to the desired insertion location for the object, then click to define the first point of the object definition axis.

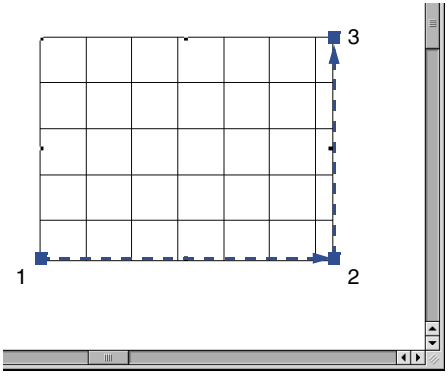


The cursor should drag a preview line from the first point of the length axis of the object, and respond to any active constraints.

4. Click to define the second point of the object length axis, then drag to define the width of the rectangular object instance.



5. Click a third time to define the width of the object instance. The object instance is then created in the document.



When the first instance of an object is placed in a document, the plug-in preferences dialog box will be displayed. This dialog allows the settings for the default parameter record to be modified before it is created in the document. Subsequent object instances will use the modified settings as their defaults.

Default object parameter settings can be accessed at any time from the plug-in preference button displayed in the mode bar when the object palette icon is selected.

Editing Rectangular Objects in the Document

Once a rectangular object is placed in a document, the settings for the specific object instance can be modified at any time by selecting the object and editing it through the Object Info palette. Rectangular objects may also be modified by resizing the object with the SmartCursor.

To edit an object instance in the document:

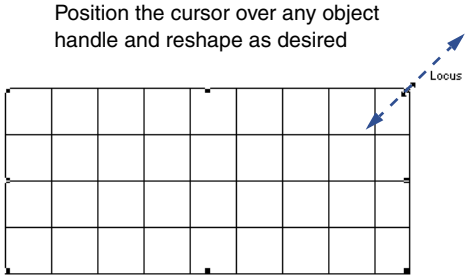
1. Select the object instance to be edited.
2. Select **Window > Palettes > Object Info** to open the Object Info palette.
3. Edit the object instance as desired.

The object specific parameters will be listed below a set of basic object editing controls which are available for any object.

To edit the object with the SmartCursor:

1. Choose the selection tool, then select the object instance to be edited.

2. Move the cursor over any object handle to be modified and hold until the cursor changes to the reshape cursor.



3. Click on the handle and drag to the desired location. Release the mouse to redefine the object.

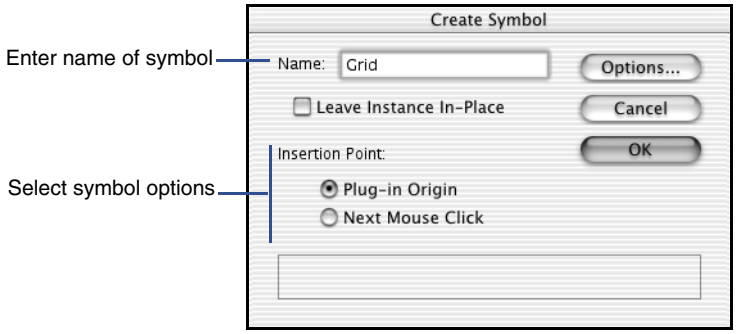
Using Rectangular Objects with the Resource Browser

There are several methods for making rectangular objects available through the Resource Browser. Like other objects, rectangular objects can be saved as a regular (static) symbol, as an object symbol, or as part of a group symbol.

Creating Static Symbols with Rectangular Objects

1. Place an instance of the object to be saved in the active document.
2. Specify the desired parameters for the object.

   The parameters will be preserved and the object will retain the appearance that was displayed when the symbol is created.
3. Select **Organize > Create Symbol**. Enter the name for the new symbol and select the desired options.



Remember that the plug-in origin for a rectangular object is the first point of the length axis. Rectangular objects in static symbols cannot be sized during placement.

4. Click **OK** to create the symbol.

The new symbol appears in the Resource Browser. If it is not visible, check to make sure the active document is selected for browsing.

The symbol can now be placed by clicking on the icon, and then selecting **Make Active** from the **Resources** menu.

An object that is contained within a static symbol can still be reverted to a plug-in object.

To revert the object:

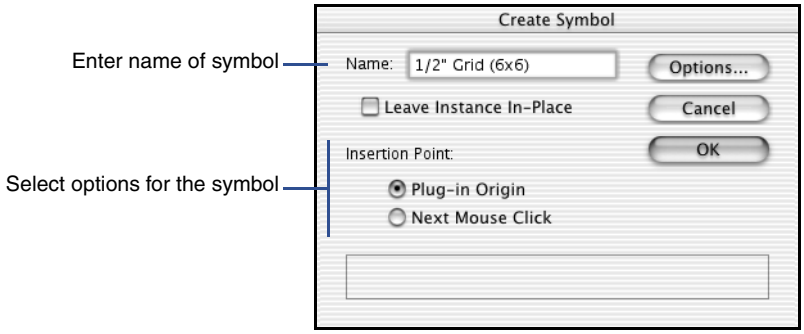
1. Select the symbol instance and choose **Organize > Convert to Plug-in**.
2. Select the appropriate conversion options and click **OK**. The symbol will revert to an object instance.

Creating Object Symbols with Rectangular Objects

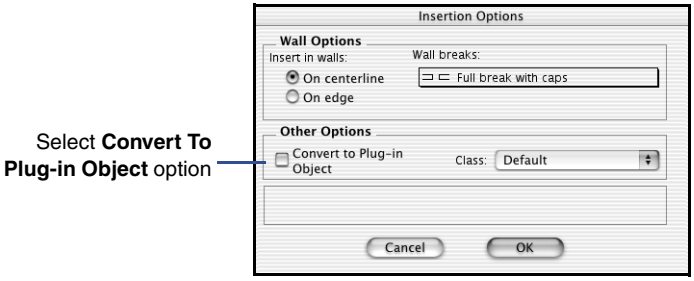
To create object symbols:

1. Place an instance of the object to be saved in the active document.
2. Specify the desired parameters for the object.

   The parameters will be preserved and the object will retain the appearance that was displayed when the symbol is created.
3. Select **Organize > Create Symbol**. Enter the name for the new symbol and select the desired options.



4. Click **Options** and then select the **Convert to Plug-in** in the Insertion Options dialog. Select any other options appropriate for the object symbol, then click **OK** to save the options.



5. Click **OK** to create the object symbol

The new symbol appears in the Resource Browser. If it is not visible, check to make sure the active document is selected for browsing.

Note that an object symbol differs in appearance from static symbols, with the object symbol appearing as a red icon.

The symbol can now be placed by clicking on the icon, and selecting **Make Active** from the **Resources** menu. When placed, the rectangular object symbols will automatically switch to the creation mode of the object, allowing the length and width to be defined. The new rectangular object instance will then be created in the document.

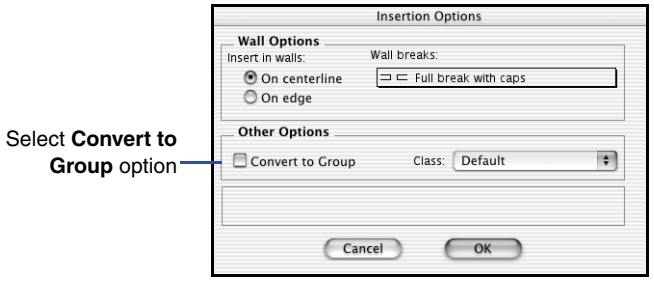
Creating Group Symbols with Rectangular Objects

To create group symbols:

1. Place the items that will be components of the group symbol in the active document.

2. Specify the desired parameters for any objects, and then select all the items to be include in the group symbol. Any objects will retain their displayed appearance when the group symbol is created.
3. Select **Organize > Create Symbol**.

   The Create Symbol dialog box opens.
4. Enter the name for the new group symbol and select the desired options.
5. Click **Options** and then select **Convert to Group** in the Insertion Options dialog. Select any other options appropriate for the object symbol, then click **OK** to save the options.



6. Click **OK** to create the group symbol.

The new group symbol appears in the Resource Browser. If it is not visible, check to make sure the active document is selected for browsing.

Group symbol symbols differ in appearance from static symbols, with the symbol appearing as a blue icon.

The group symbol can now be placed by clicking on the icon, and then selecting **Make Active** from the **Resources** menu. When placed, the symbol will automatically convert to an object group. Any objects in the group can be modified by entering the group editing the objects as desired.

VectorScript Path Objects

In this Chapter:

VectorScript **path objects** are the last of the four plug-in object types available in VectorWorks. Path objects, as with the other object types already introduced, are named according to how they are defined in the VectorWorks file. Path objects utilize a user-defined polygonal path to define and create the basic geometry of the object.

Path objects, like all other objects, support the standard VectorWorks core technologies, and behave similarly to the basic VectorWorks object types. Path objects can be edited using the Object Info palette, or they can be edited on-screen using the SmartCursor to modify the object instance.

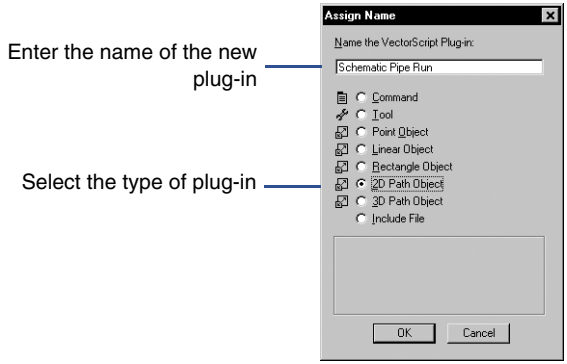
This section documents the basic techniques needed to create and use path objects in VectorWorks files.

Creating a Path Object Plug-in

VectorScript parametric objects are created using the VectorWorks Plug-in Editor to create and define the actual plug-in item. The Plug-in Editor provides a single interface for creating the plug-in script and editing the associated settings objects will use when placed during a VectorWorks session.

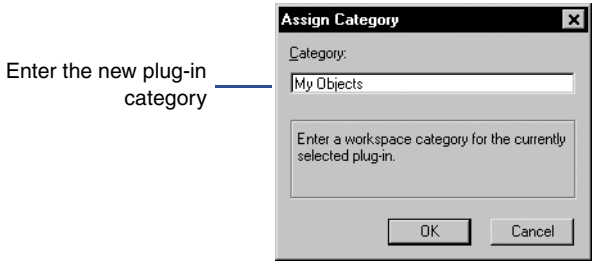
Creating the Object Plug-in

1. Select **Organize > Scripts > Create Plug-in**.
2. In the VectorScript Plug-in Editor dialog box, click **New**.
3. In the Assign Name dialog box, enter the name of the new plug-in item and select **2D Path Object** as the type of plug-in. Click **OK** to create the plug-in item.



Setting the Object Category

1. Click the **Category** button from the VectorScript Plug-in Editor dialog box.
2. In the Assign Category dialog box, enter the name of the category to be associated with the new object plug-in item. Click **OK** to set the plug-in category to the new value.



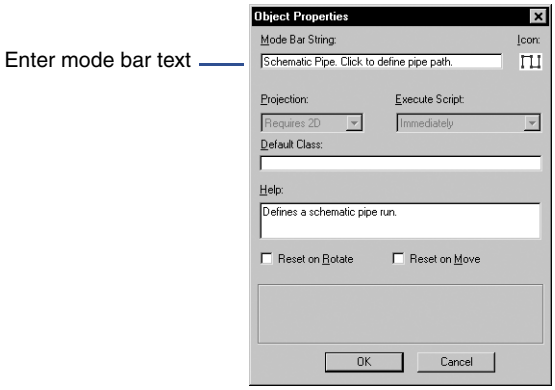
The plug-in category is the heading under which the object may be found when selecting items in the Workspace Editor.

Setting Options for the Object

Path objects have a several settings which allow them to maintain behavior consistent with standard VectorWorks tools. These settings, also known as **plug-in properties**, control how the object is placed, as well as the various default values for the object when it is created.

Setting Display Defaults for the Object

1. Click the **Properties** button from the VectorScript Plug-in Editor dialog box.
2. In the Object Properties dialog box, enter the desired descriptive text to be displayed in the mode bar in the **Mode Bar String** field.



   Mode bar text usually includes the name of the tool, and can include text indicating the action the user should perform.

3. Click **OK** to save the new settings for the object.

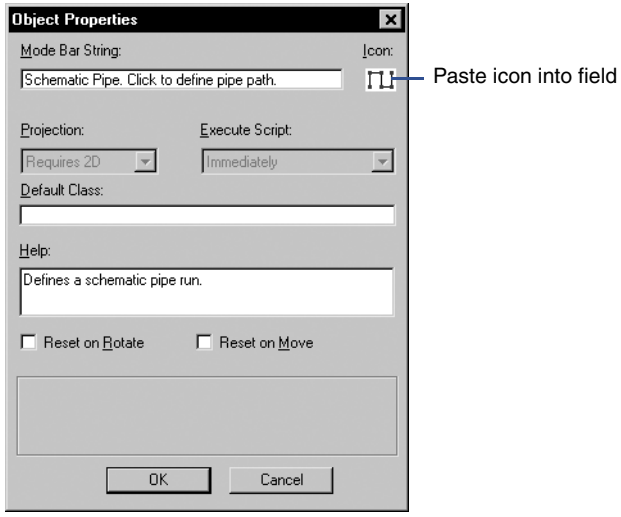
Setting the Object Icon

VectorScript plug-ins come preloaded with a default icon which is displayed when the plug-in is placed in a workspace tool palette. This icon can be replaced with a custom icon indicating the function of the plug-in item.

To create the icon, use a third-party icon editor. Tool and object icons use an 8-bit, 32-by-32 pixel field for defining the icon; the actual icon graphic, however, should be confined to an area 24 pixels wide by 18 pixels high, centered in the field.

To create the VectorScript object icon:

1. Click the **Properties** button from the VectorScript Plug-in Editor dialog box.
2. In the Object Properties dialog box, click in the icon display field to select it. The icon field is highlighted.



3. Paste the new object icon into the icon field. The customized icon should be visible in the icon field of the dialog box.
4. Click **OK** to save the new settings for the object.

Setting Activation Options for the Object

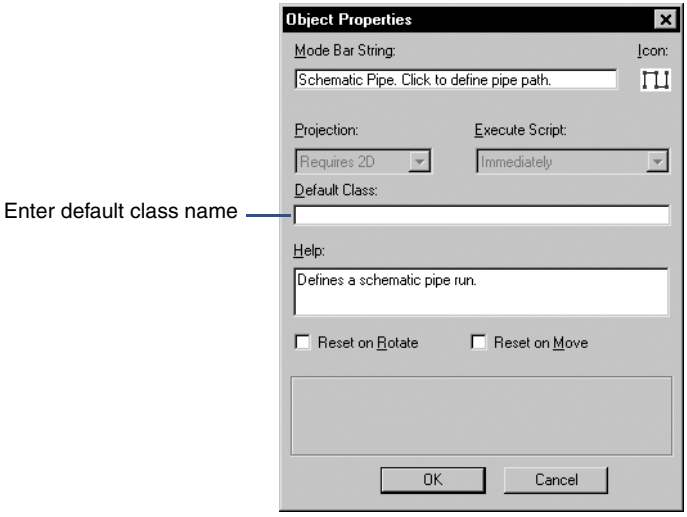
Activation options for objects control the conditions under which the object script code will execute. VectorWorks presets these options to the optimal configuration for object interaction with the VectorWorks application.

Setting the Default Class of the Object

Objects can be specified to have a default class setting on insertion into a file. This preset default class allows objects to be automatically classed without the class being active, or requiring additional editing using the Object Info palette.

To define a default class for the object:

1. Click the **Properties** button from the VectorScript Plug-in Editor dialog box.
2. In the Object Properties dialog box, enter the desired class name in the default class field.



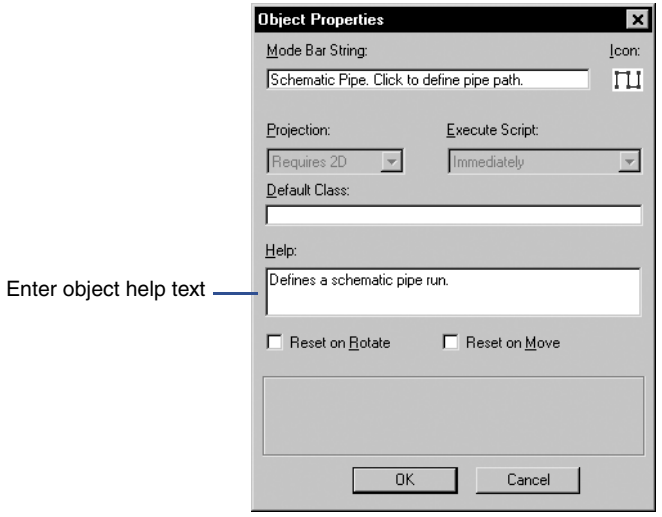
If the default class does not exist in the file when the object is placed, the class will be automatically created.

Setting Help Text for the Object

Help text describing the object will display when the cursor is held over the object icon in a tool palette.

To create help text for the object:

1. Click the **Properties** button from the VectorScript Plug-in Editor dialog box.
2. In the Object Properties dialog box, enter the desired help text in the **Help** field. The text will be displayed when help text is enabled and the cursor placed over the tool palette item.



3. Click **OK** to save the new settings for the object.

Setting Object Reset Options

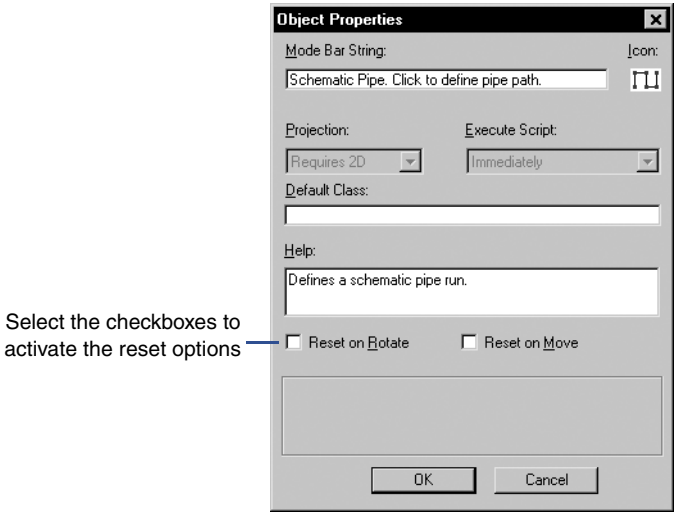
Objects have two properties which control when the geometry of the object will be recalculated and regenerated in the file. These properties are known as the **object reset options**.

By default, object geometry will only be recalculated if the object is edited using object parameters or control points. This is important, because when object geometry is recalculated, the document default settings for attributes such as font, text size or line color will be reapplied to the object. If any of these settings has been modified since the object was placed or last edited, changes in the appearance of the object may occur. The default reset options allow objects to be manipulated without invoking object regeneration.

For instances where it is important that the object recalculate (for example, windows placed in a wall), objects can be optionally set to recalculate their geometry when the object has been moved or rotated.

To set the object reset options:

1. Click the **Properties** button from the VectorScript Plug-in Editor dialog box.
2. In the Object Properties dialog box, click the desired object reset option to activate it.



Selecting the **Reset on Rotate** option causes the object to recalculate when the object is rotated in the file. Selecting **Reset on Move** will cause the object to recalculate when the object is moved in either 2D or 3D, as well as when the object is cut and pasted into the file

3. Click **OK** to save the new settings for the object.

The new recalculation options for the object take effect immediately.

Parameters and Path Objects

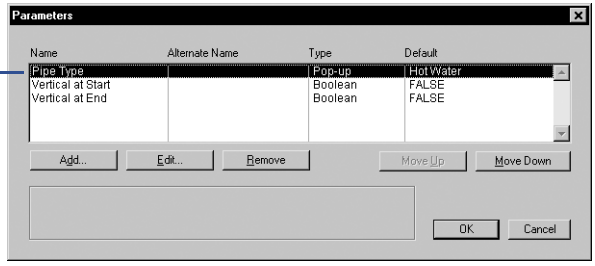
The parameters which define the appearance of a VectorScript path object are stored in a parameter record which is associated with each point object instance placed in the file. The parameters for each object instance may be modified by using the Object Info palette to access the values in the object parameter record.

A default parameter record is also created in the file when the first instance of an object is created in the active file. This default parameter record, which is distinct from the parameter records associated with object instances, stores the object default settings with the file. It is used when placing subsequent object instances to define the defaults for each new object instance.

Creating a Parameter Record for an Object

1. Click the **Parameters** button from the VectorScript Plug-in Editor dialog box.
2. In the Parameters dialog box, create the desired parameter record settings for the plug-in. For details on specific plug-in parameter types, see “Using VectorScript Plug-ins” on page 10-1.

Create parameter record settings with default values for the object



3. When all the desired parameters have been created, click **OK** to save the parameters.

When an object instance is placed in the file, the object’s parameter record can be edited in the Object Info palette.

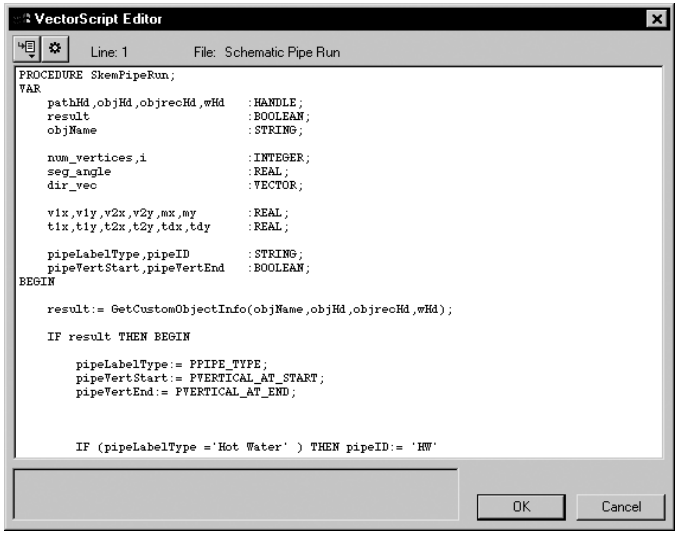
The Object Info palette also provides a convenient interface for editing the definition path for path objects.

Creating the Object Script

The script source code for the object plug-in can be created using the VectorScript editor or a third-party text editor. The source code is saved as part of the plug-in item.

Creating Script Code for a Path Object

1. Click the **Script** button from the VectorScript Plug-in Editor dialog box.
2. Enter the script source code in the VectorScript Editor window.



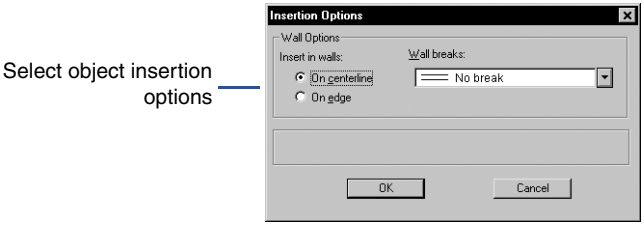
3. When the script has been entered, click **OK** to save the script as part of the plug-in.

Setting Object Insertion Options

Objects, like symbols, can be assigned predefined insertion options for placement. These options allow objects to properly interact with walls or other advanced VectorWorks object types.

Setting Insertion Options for a Path Object

1. Click the **Insert Options** button from the VectorScript Plug-in Editor dialog box.
2. In the Insertion Options dialog box, select the desired option settings for the object.



For objects which do not require specific insertion options, leave the options at the default settings.

Working With Path Objects

Working with objects incorporates elements of tool and symbol usage. Like tools, objects can be selected from tool palettes and make use of the SmartCursor, mode bar, and other core VectorWorks features. Like symbols, objects can be inserted into walls, and can optionally be configured to be available from the Object Browser.

Working effectively with path objects requires the knowledge of several basic techniques for managing and using objects with VectorWorks and VectorWorks files.

Adding a Path Object to a Workspace

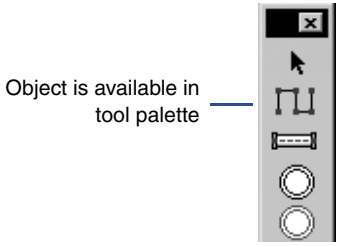
Once an object plug-in has been created, it will need to be added to one or more workspaces in order to be available for use with VectorWorks. Object plug-ins are functionally similar to tool items, and may be placed into tool palettes for use. Once the object has been added to the workspace, it will immediately be available for use in the current VectorWorks session.

To add an object to a VectorWorks workspace:

1. Select **File > Workspaces > Workspace Editor**.
2. Switch to the tools pane in the Workspace Editor, then look for the category that was assigned to the new object plug-in. Click the disclosure triangle to display the available items in the category.
3. Click and drag the object plug-in to the desired location on a tool palette. If desired, add a key equivalent for the object.
4. Click **OK** to save the edited workspace.

For additional details on editing workspaces, see Appendix B in the VectorWorks User's Guide

Once the object has been added, the object should be visible on the tool palette. To activate the object, click on the item as if it were a standard VectorWorks tool.



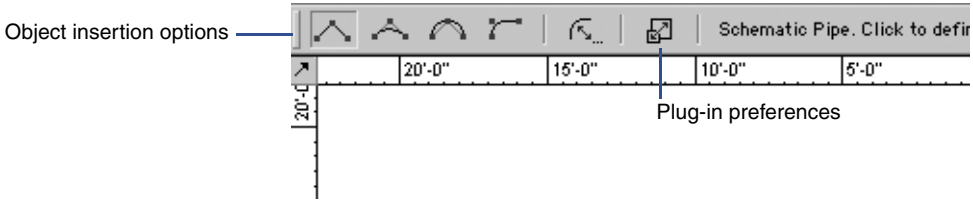
Placing Objects in Files

Placing path objects is very similar to using any other VectorWorks tool. Objects, like VectorWorks tools, make use of the SmartCursor, mode bar, and constraints to allow precise placement of the object. Once placed, path objects can be edited using the Object Info palette, or they can be edited using the SmartCursor.

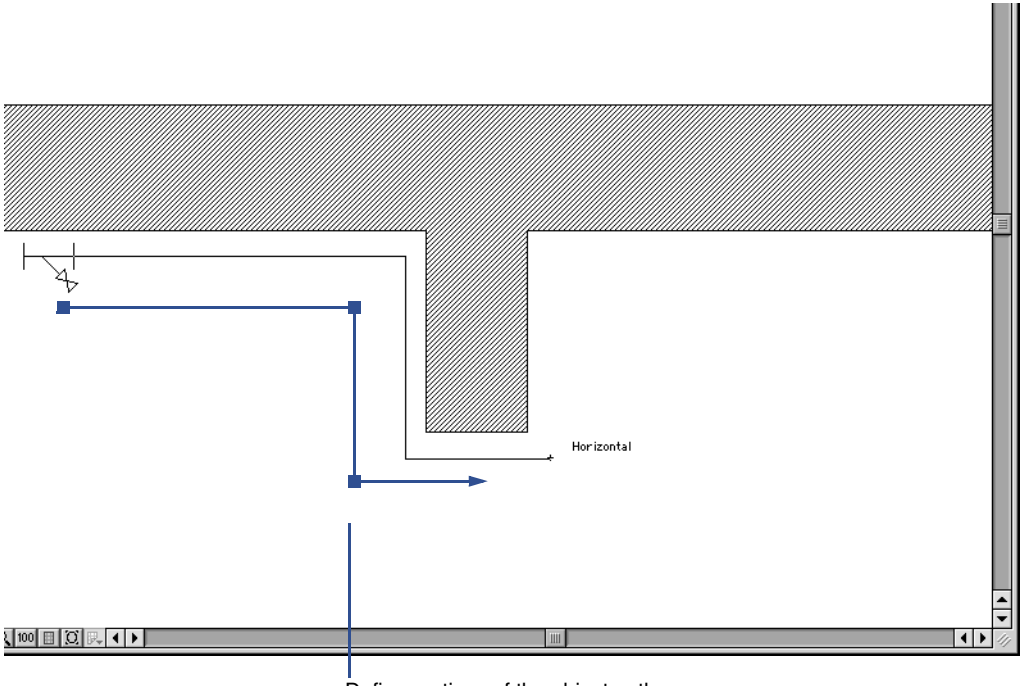
The insertion point of a path object corresponds to the first click defining the objects' path polygon, and acts as the origin (0,0) of the object definition.

To place a path object in a file:

1. Open the tool palette containing the object and click the object icon to activate it.
2. Select the desired insertion mode for the path object.

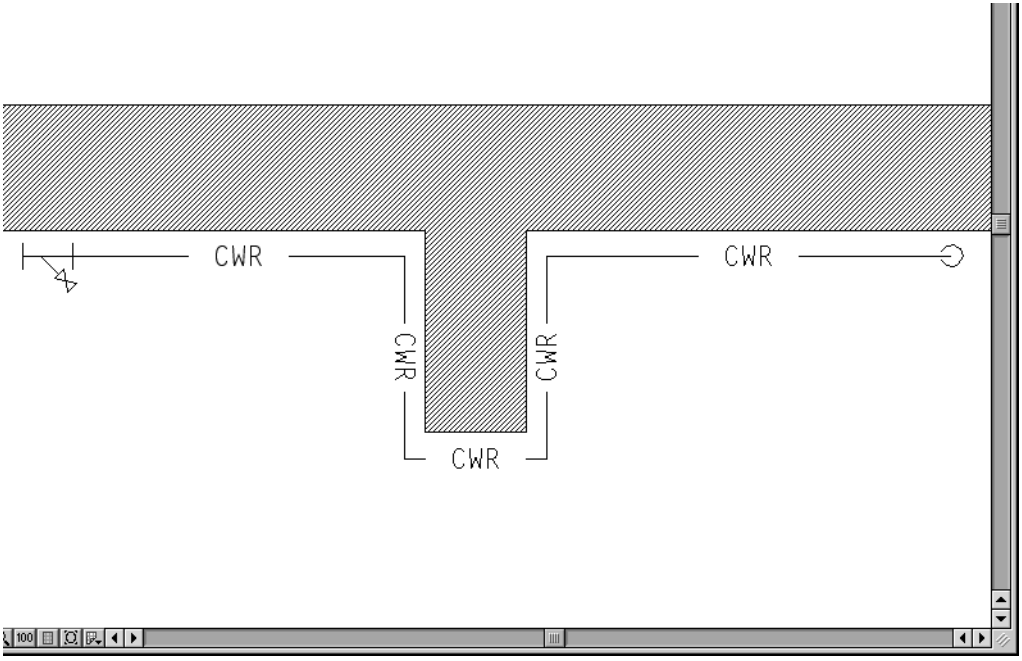


3. Move the cursor to the desired insertion location for the object, and then click to define the first point of the object path. Continue clicking to define the vertices of the path as you would if you were defining a polygon.



The cursor displays an image preview from the most recently defined vertex of the object path, and should respond to any active constraints.

4. Double-click to end defining the object path. The new path object will then be created.



When the first instance of an object is placed in a file, the plug-in preferences dialog is displayed. This dialog allows the settings for the default parameter record to be modified before it is created in the file. Subsequent object instances will use the modified settings as their defaults.

Default object parameter settings can be accessed at any time from the plug-in preferences button displayed in the mode bar when the object palette icon is selected.

Editing Path Objects

Once a path object has been placed, the settings for the specific object instance can be modified by selecting the object and editing it in the Object Info palette. Both object specific settings and object path information can be edited from the Object Info palette.

To edit an object instance:

1. Select the object instance to be edited.
2. Select **Window > Palettes > Object Info** to open the Object Info palette.

3. Edit the object instance as desired.

   The object specific parameters will be listed below a set of basic object editing controls which are available for any object. Below the object specific parameter controls are a set of controls that can be used to edit the object definition path.

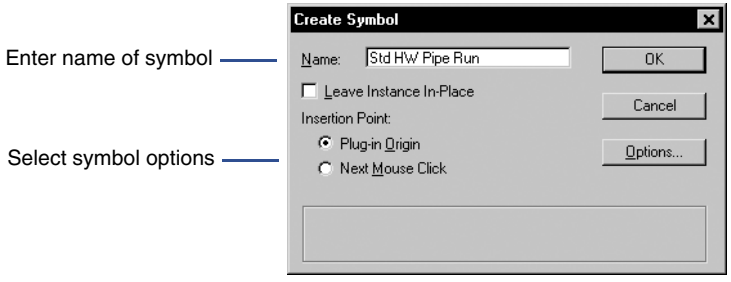
Using Path Objects with the Resource Browser

There are several methods for making path objects available through the Resource Browser. Like other objects, path objects can be saved as a regular (static) symbol, as an object symbol, or as part of a group symbol.

Creating Static Symbols with Path Objects

1. Place an instance of the object to be saved in the active file.
2. Specify the desired parameters for the object.

   When the symbol is created, the parameters will be preserved and the object will retain the appearance that was displayed.
3. Select **Organize > Create Symbol**. Enter the name for the new symbol and select the desired options.



   Remember that the plug-in origin for a path object is the first point of the object path. Path objects in static symbols cannot be defined during placement.
4. Click **OK** to create the symbol.

   The new symbol appears in the Resource Browser. If it is not visible, check to make sure the active file is selected for browsing.

The symbol can now be placed by clicking on the icon and selecting **Make Active** from the **Resources** menu.

An object that is contained within a static symbol can still be reverted to a plug-in object.

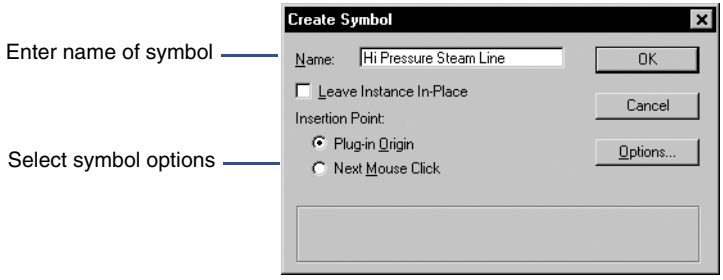
To revert the object:

1. Select the symbol instance and choose **Organize > Convert to Plug-in**.
2. Select the appropriate conversion options and click **OK**. The symbol will revert to an object instance.

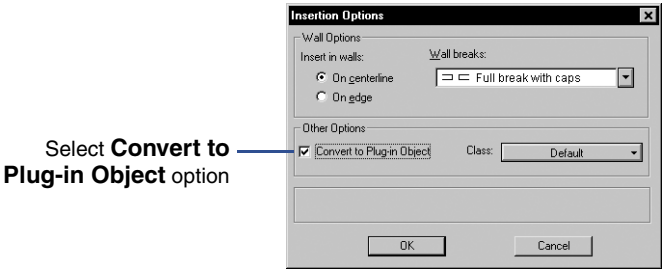
Creating Object Symbols with Path Objects

1. Place an instance of the object to be saved in the active file.
2. Specify the desired parameters for the object.

   When the symbol is created, the parameters will be preserved and the object will retain the appearance that was displayed.
3. Select **Organize > Create Symbol**. Enter the name for the new symbol and select the desired options.



4. Click **Options** and select the **Convert to Plug-in** option in the Insertion Options dialog box. Select any other options appropriate for the object symbol, then click **OK** to save the options.



5. Click **OK** to create the object symbol

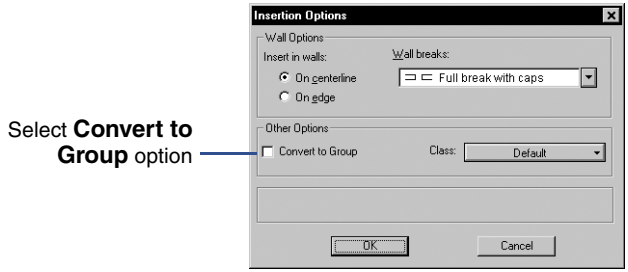
   The new symbol appears in the Resource Browser. If it is not visible, check to make sure the active file is selected for browsing.

Note that an object symbol differs in appearance from static symbols, with the object symbol appearing as a red icon.

The symbol can now be placed by clicking on the icon, and then selecting **Make Active** from the **Resources** menu. When placed, the path object symbols will automatically switch to the creation mode of the object, allowing the length and width to be defined. The new path object instance will then be created in the file.

Creating Group Symbols with Path Objects

1. Place the items that will be components of the group symbol in the active file.
2. Specify the desired parameters for any objects, and then select all the items to be included in the group symbol. Any objects will retain their displayed appearance when the group symbol is created.
3. Select **Organize > Create Symbol**. Enter the name for the new group symbol and select the desired options.
4. Click **Options** and select the **Convert to Group** option in the Insertion Options dialog. Select any other options appropriate for the object symbol, then click **OK** to save the options.



5. Click **OK** to create the group symbol.

The new group symbol appears in the Resource Browser. If it is not visible, check to make sure the active file is selected for browsing.

Group symbol symbols differ in appearance from static symbols, with the symbol appearing as a blue icon.

The group symbol can now be placed by clicking on the icon, and then selecting **Make Active** from the **Resources** menu. When placed, the symbol will automatically convert to an object group. Any objects in the group can be modified by entering the group and editing the objects as desired.

VectorScript Development Tools

In this Chapter:

VectorWorks provides a suite of development tools for creating and maintaining scripts and plug-ins. They include a script editor, a script debugger, and a plug-in editor for setting up and configuring VectorScript plug-ins. These tools are integrated into the VectorWorks application, and can be used directly from within the program.

Creating Scripts

VectorWorks provides several methods for creating, managing and using scripts. The most basic of these methods is to create a VectorWorks document and select the **File > Export > Export VectorScript** command. The command creates a script which can be run by the **File > Import > Import VectorScript** command, or by selecting the file in the Resource Browser, and then selecting **Run** from the **Resources** menu.

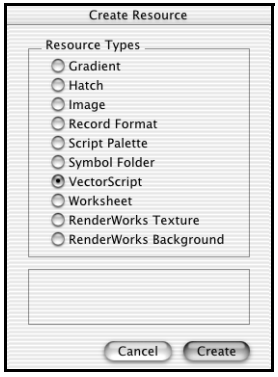
The traditional method for storing scripts, which has been a feature of VectorWorks since its original release as MiniCad, is in **document scripts**. Document scripts are stored with individual documents in **script palettes**, which organize the scripts and can be displayed or hidden as needed. Both document scripts and script palettes can be created and accessed from the Resource Browser.

Beginning with VectorWorks 8, scripts can also be created and stored as plug-ins. Plug-ins are used as a component of a workspace, and can be accessed by any document. Scripts in plug-ins can be used as tool items, tools, or parametric objects. Plug-ins are created and maintained using the plug-ins editor, which is accessed by selecting **Organize > Scripts > Create Plug-in**.

Creating a Document Script

To create a document script:

1. In the Resource Browser, select **New Resource** from the **Resources** menu.
2. Select the **VectorScript** option and click **Create**.



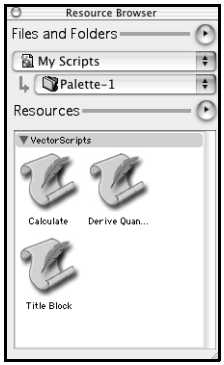
   Newly created scripts are located by default in the active script palette (the palette which is open and active, or which is open in the window of the Resource Browser). If no script palette is active, you will be prompted to select a location for the script. If no script palette exists in the document, a new palette will be created to contain the script. The Create Resource dialog box allows the creation of new Script Palettes as well.

3. Enter the name of the script.

   The script editor window will be displayed to create the script.

Editing an Existing Document Script (Resource Browser)

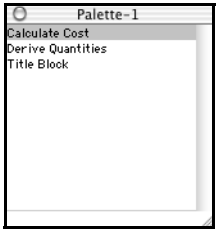
1. In the Resource Browser, select the script to be edited.



2. Select **Open** from the **Resources** menu. The VectorScript Editor opens, displaying the script source code.

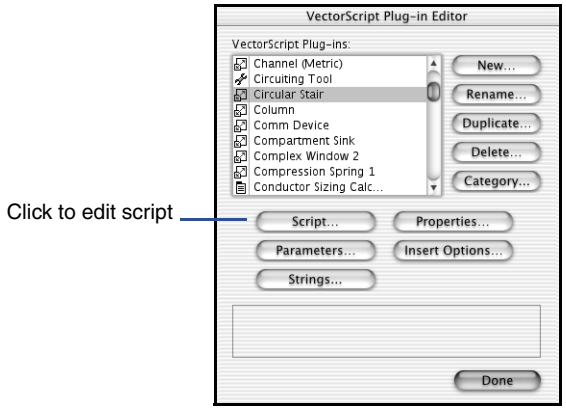
Editing an Existing Document Script (Script Palette)

1. Open a script palette containing the script to be edited.
2. Option (Mac) or Alt (Win) + double-click on the script to be edited. The VectorScript Editor opens, displaying the script source code.



Creating Scripts in the Plug-in Editor

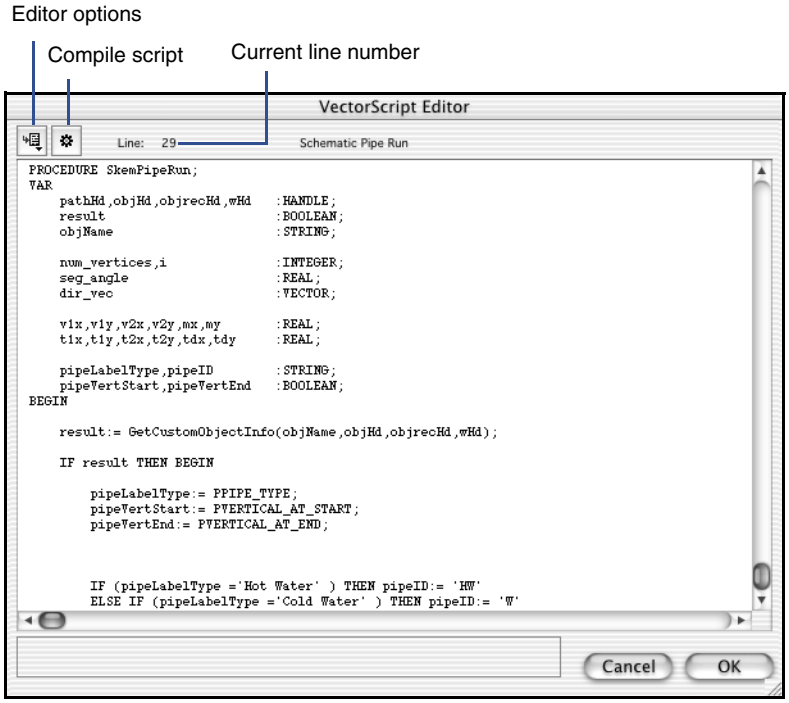
1. Select **Organize > Scripts > Create Plug-in.**
2. Click **New**, and then select the type and enter the name of the plug-in to be created.
3. Select the plug-in to be edited, and then click the **Script** button. The VectorScript Editor opens, displaying the plug-in script source code.



To edit an existing plug-in, open the plug-ins editor, select the plug-in to be edited, and click the **Script** button.

The VectorScript Editor

The VectorScript Editor provides a basic authoring environment for script development and maintenance. Its features allow you to edit and compile scripts, browse the API, view errors, as well as perform other tasks associated with creating scripts.

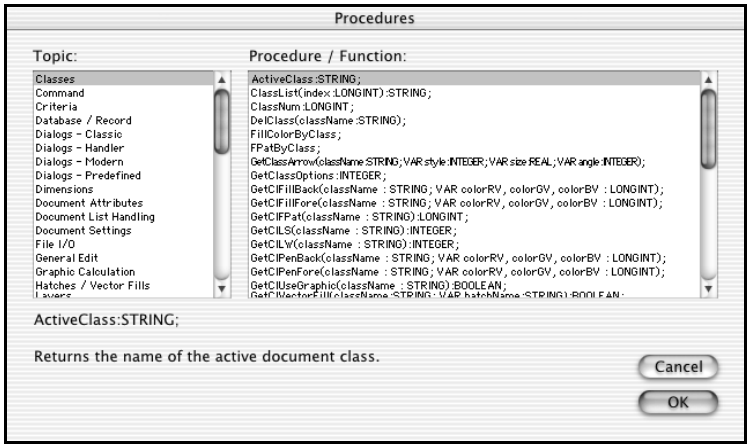


Editor Options

The Editor Options pop-up menu provides access to the extended features of the editor.

Procedure

The **Procedure** option provides access to a browser listing all the functions found in the VectorScript API. Functions are listed by category and provide a function prototype as well as a brief description of what operation is performed by the function.

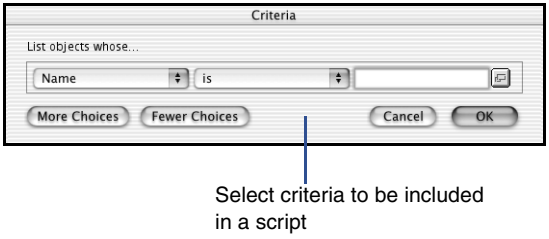


Inquiry

The **Inquiry** option provides a listing of all functions which use search criteria. When used in conjunction with the **Criteria** option (see below), custom queries or selections can easily be defined in a script.

Criteria

The **Criteria** option provides a convenient method of defining search criteria for use in scripts. The dialog, which is similar to the Custom Selection dialog, allows criteria to be chosen from a listing of available search options.



Tool / Attribute

The **Tool / Attribute** option provides a way of saving the current tool and attribute state information into a script.

Parameters

The **Parameters** option provides access to a plug-in objects' parameter list for editing.

Text File

The **Text File** option allows script source code to be imported from external text files.

Compile Script

The **Compile Script** button allows a script to be compiled directly from the the VectorScript Editor without the need to execute the script. If errors exist within the script which prevent successful compilation, they will be displayed and can be resolved without the need to exit the script editor.

Line Number

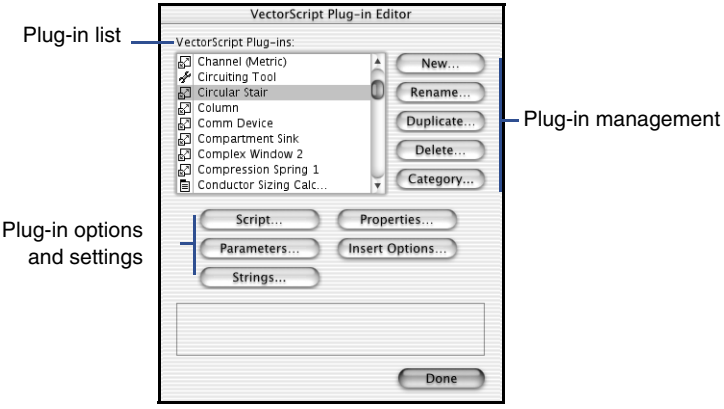
The current position of the cursor within the edit field of the VectorScript Editor is indicated by the line number displayed in the editor window.

VectorScript Plug-in Editor

The VectorScript Plug-in Editor is the VectorWorks interface for creating and editing VectorScript plug-in objects. The editor provides tools for editing scripts, preference settings, and parameter lists for all plug-in types.

Using the Plug-in Editor

To open the Plug-in Editor, select **Organize > Scripts > Create Plug-in**.

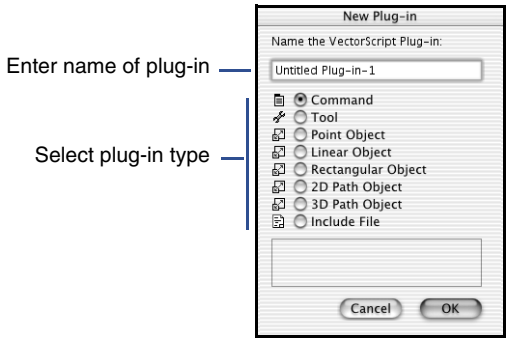


The main editor window displays a listing of all available plug-ins in the VectorWorks installation. The dialog also provides options for managing plug-in files, as well as for accessing the various settings and options available in each plug-in.

Managing Plug-ins

New

To create a new plug-in object, click the **New** button. When prompted, enter the name and select the type of plug-in to be created.



Plug-in names are limited to twenty characters in length. The appropriate plug-in extension will be appended to the plug-in name.

Rename

To rename an existing plug-in, select a plug-in from the list and click the **Rename** button. Enter the new name for the plug-in and click **OK**.

Duplicate

To create a copy of an existing plug-in, select a plug-in from the list and click the **Duplicate** button. Enter a name for the plug-in and click **OK**.

Delete

To delete an existing plug-in, select a plug-in from the list and click the **Delete** button.

Category

To specify a category for a plug-in, select a plug-in from the list and click the **Category** button. When prompted, enter the name of category that will be associated with the plug-in.

The plug-in category is the heading that the plug-in may be found under in the Workspace Editor.

Plug-in Option Settings

Script

To modify the script associated with a plug-in item, click the **Script** button. The VectorScript Editor will be displayed, allowing the script to be created or edited.

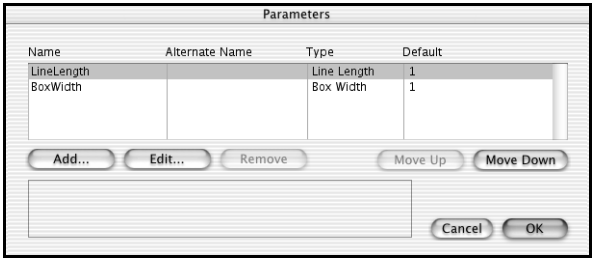
Properties

To modify the properties associated with a plug-in item, click the **Properties** button. The Properties dialog box specific to the type of the selected plug-in item will be displayed.

For more details on specific plug-in properties, see “Using VectorScript Plug-ins” on page 10-1.

Parameters

To create or modify parameters associated with the plug-in parameter record, click the **Parameters** button. The Parameters dialog box will be displayed, allowing specific parameter settings to be edited or created.



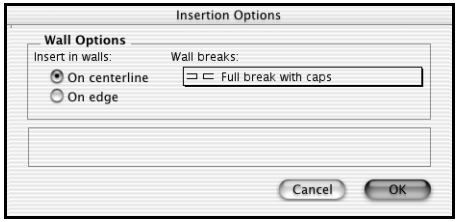
Parameter records may be created for any plug-in object type. Parameter records for object plug-ins store data which defines the display characteristics of the object; this information can also be edited from the Object Info palette. Parameter records for custom tools and tool items store default and status

related data for the item; this information can only be edited through the parameters dialog box.

For more details on plug-in object parameters, see “Using VectorScript Plug-ins” on page 10-1.

Insert Options

To set insertion options for object plug-ins, click the **Insert Options** button. The Insertion Options dialog box opens, where insertion options for the object can be specified.



Insertion options cannot be specified for menu command or tool item plug-ins. For more details on insertion options available for object plug-ins, see Chapters 11 through 16.

The VectorScript Debugger

VectorScript provides a powerful tool to assist in solving problems that may occur while developing scripts. This tool, known as a **source-level debugger**, controls the execution so that the internal operations of the script can be observed while the script is running. Using the debugger, it becomes possible locate and solve problems by moving through the script line by line to view the associated data, variables, and flow of script execution.

Launching the Debugger

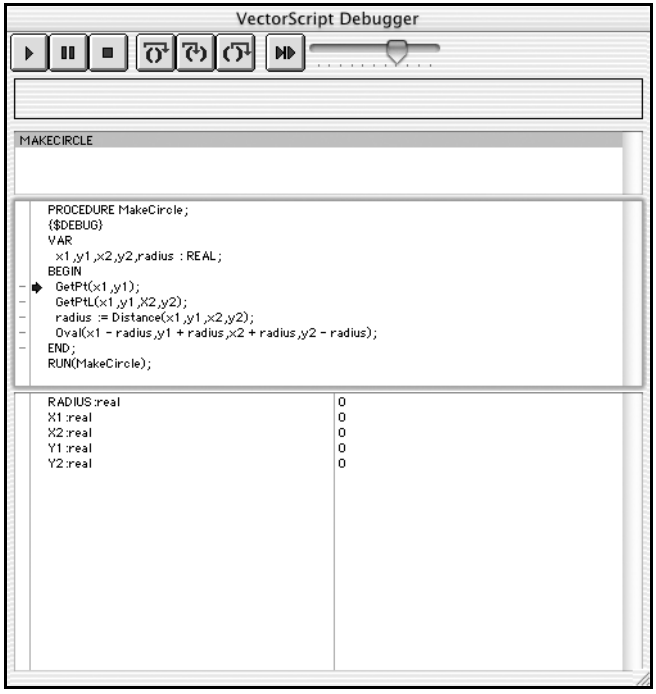
The VectorScript debugger is activated by using the `{$DEBUG}` compiler directive. This compiler directive, which can be placed anywhere within a script, instructs the compiler to activate and display the debugger window when the script is executed. For example,

```
PROCEDURE MakeCircle;
{$DEBUG}
VAR
```

```
    x1,y1,x2,y2,radius : REAL;

BEGIN
    GetPt(x1,y1);
    GetPtL(x1,y1,x2,y2);
    radius:= Distance(x1,y1,x2,y2);
    Oval(x1 - radius,y1 + radius,x2 + radius,y2 - radius);
END;
Run(MakeCircle);
```

This launches the VectorScript debugger and displays the window as shown:

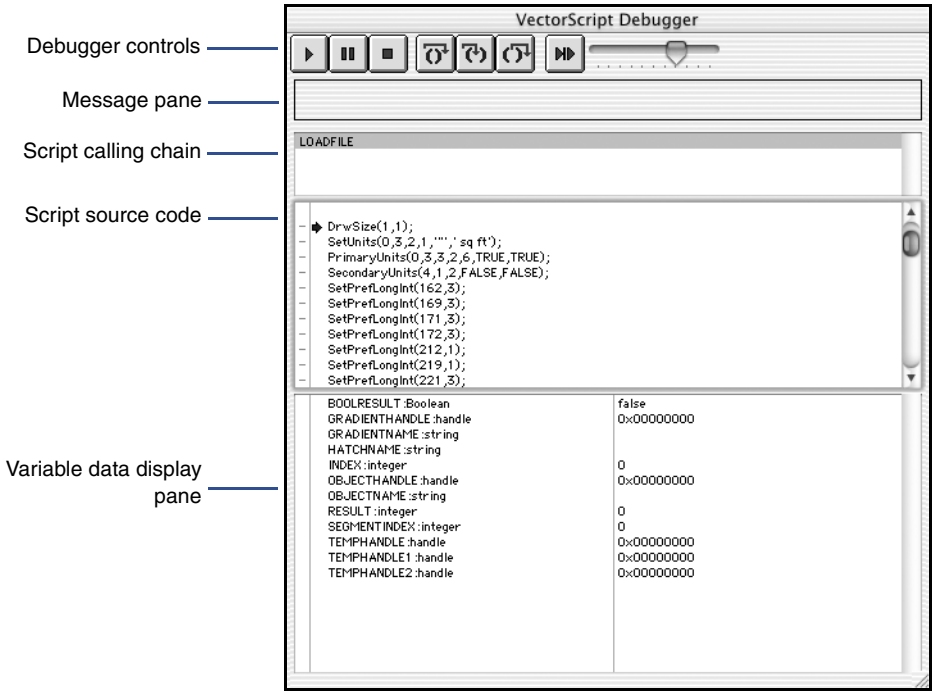


The VectorScript debugger allows a script to be executed in a line-by-line fashion, also known as "stepping" through the source code. The debugger performs this task beginning at the first line of the script and continuing through each line until the end of the script is reached.

When the debugger is launched, storage for variables and constants is defined and script execution is paused at the first line of code in the script body. The

debugger window is then displayed, providing a wide array of information on the script and the current state of execution.

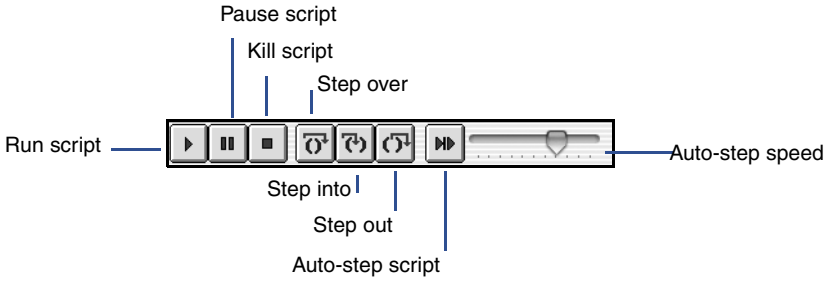
The Debugger Interface



The debugger window contains controls for managing script execution, as well as several areas for displaying various data about the script and the state of execution.

Debugger Controls

The debugger controls allow the execution of the script to be controlled during the debugging process. Script execution can be started, stopped, paused, or advanced one line at a time.



See “Controlling Execution” on page 17-14 for details on controlling script execution in the debugger.

Message Pane

The message pane displays information about the script. These messages may include prompts for user interaction with the script, script status information, or errors encountered in script execution.

Script Calling Chain

The script calling chain pane displays the current function calling chain of the script. Each subroutine name appears below the function calling it in the list; by highlighting the desired subroutine name, it is possible to determine which subroutines are being called and the execution location within those subroutines.

The script calling chain pane is resizable; to resize the pane, click and drag the bottom border of the pane.

Script Source Code

The script source code pane displays the source code of the script being debugged. The current location of execution in the script is indicated by the small blue arrow on the left-hand side of the pane. This arrow indicates the line of code that is about to be executed.

The script source code pane is resizable; to resize the pane, click and drag the bottom border of the pane.

Variable Data Display

The variable data display pane displays the current values stored in variables, arrays, and structures declared in the script. The display is updated as execution proceeds, so that values can be continuously watched for changes during execution.

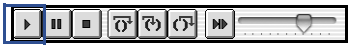
Arrays, elements, and structure members may be displayed by clicking on the disclosure triangle that appears next to the item.

The variable data display pane is resizable; to resize the pane, click and drag the top border of the pane. The data value area of the pane may also be resized; to resize this area, click and drag the vertical divider bar in the pane.

Controlling Execution

Running a Script

To run a script in the debugger, click the **Run** button.

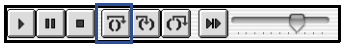


Running a script here is identical to running a script from a script palette or plug-in; when the script has completed execution, the debugger window will close.

Running scripts is primarily used in conjunction with setting a breakpoint in the debugger. For details on setting and using breakpoints, see “Using Breakpoints” on page 17-16.

Stepping Through a Single Line Of a Script

To execute a single statement in the script, click the **Step Over** button.



The **Step Over** button advances script execution by a single statement and refreshes the data display pane of the debugger. The script position indicator advances to indicate the new location of script execution. If the statement which is stepped through is a user defined subroutine, all the code within the subroutine is executed.

When stepping through a statement containing a user-interactive function call (such as `GetPt()` or `GetLine()`), the debugger will prompt the user for input. Custom dialog function calls will cause the dialog to become active until a dialog event occurs; control is then returned to the debugger.

Stepping Into a Subroutine

To advance execution into a user defined subroutine, click the **Step Into** button.

The **Step Into** button is used when a statement containing a call to a user-defined subroutine is reached. Whereas the **Step Over** button will execute all the code contained within the subroutine and move to the next statement in the calling function, **Step Into** will advance script execution to the first statement within the subroutine body.

The **Step Into** button performs a **Step Over** action with all other statements.

Stepping Out of a Subroutine

To advance execution from the current script location in a subroutine to the next statement in the calling function, click the **Step Out** button.

When stepping out of a subroutine, all statements which follow the current script location will be executed, and script execution will advance to the next statement in the calling routine.

Auto-Step Through Script

To automatically step through a script on a line-by-line basis, click the **Auto-Step** button.

The **Auto-Step** button will automatically advance the script at a speed determined by the Auto-Step slider control.

Pausing Script Execution

To pause script execution, click the **Pause Script** button.

The **Pause Script** button will stop auto-step execution at the current execution location in the script. Script execution can be resumed by clicking the **Auto-Step** button.

The **Pause Script** button can also pause execution of scripts in infinite loops; in some instances, however, **Pause Script** may not be able to stop such loops.

Stopping Script Execution

To terminate execution of a script, click the **Kill Script** button.

The **Kill Script** button will immediately terminate script execution. After the **Kill Script** button is pressed, the debugger window will close.

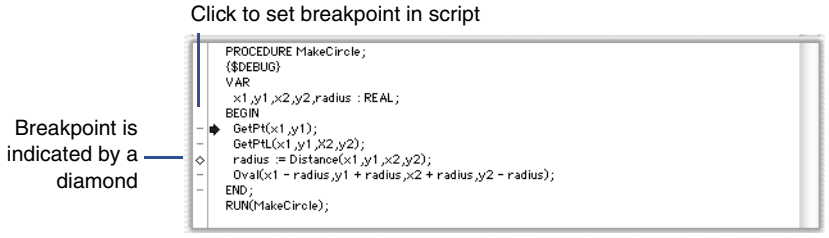
Using Breakpoints

A **breakpoint** suspends script execution and transfers control of the script to the debugger. When a breakpoint is encountered, execution is suspended just prior to the breakpoint, and the script execution pointer is positioned at the breakpoint location.

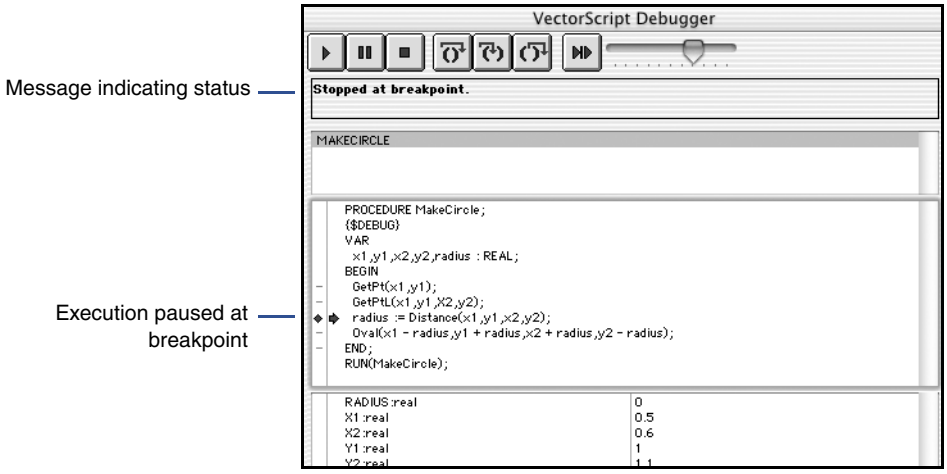
Breakpoints are often used in order to run scripts and stop them just prior to a statement or statements that are to be debugged. Once the script has stopped at a breakpoint, it may be stepped through manually or by using the auto-step feature.

Setting a Breakpoint

To set a breakpoint in the debugger, click the dash in the narrow column on the left side of the script source code pane. The new break point will be indicated by a small diamond at the break location.



Once a breakpoint has been set, the script can be executed using the **Run Script** button. Script execution will pause when the breakpoint is reached, as indicated by a highlighted breakpoint and execution pointer arrow. The message pane of the debugger will also indicate that a break point has been reached.



To continue execution, click either the **Run Script**, **Step Over**/**Into**/**Out**, or **Auto-Step** buttons.

Breakpoints which have been placed in a looping statement will cause the script to stop each time the breakpoint is encountered. When used in conjunction with the **Run Script** or **Auto-Step** buttons, such breakpoints can be used to observe conditions occurring within a loop during execution.

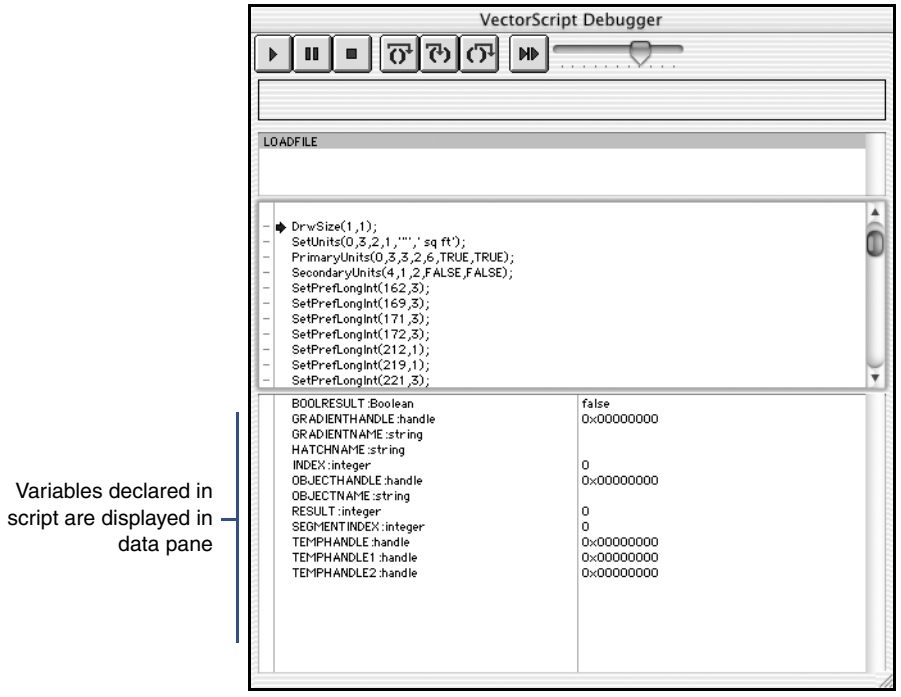
Care should be exercised when placing breakpoints within branching statements such as `IF..THEN` or `CASE` statements. If the breakpoint is in a branch outside the path of execution, the script will continue to execute.

Clearing Breakpoints

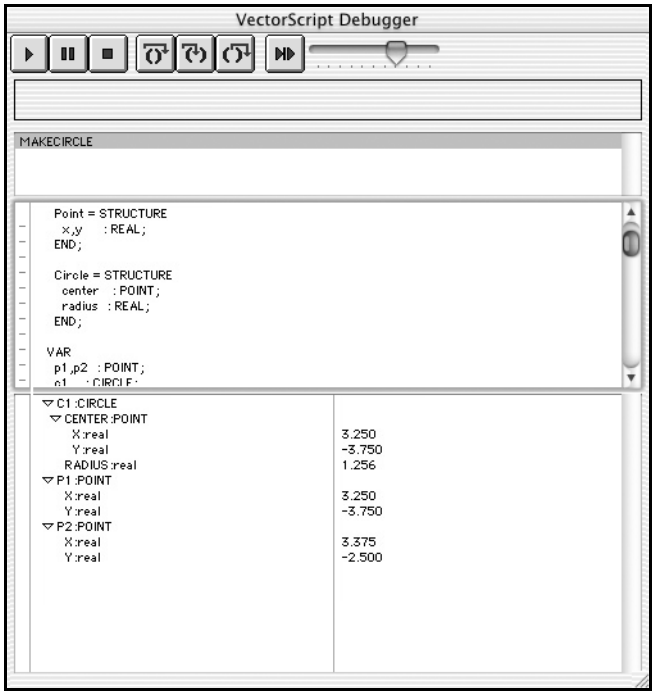
To clear a breakpoint in the debugger, click on the diamond indicating the breakpoint location while script execution is stopped or paused. The breakpoint will be removed from the script.

Viewing Data in the Debugger

Constant and variable data values may be observed during script execution in the data display pane of the debugger. This pane displays all data storage locations declared in the script, as well as the values contained within them. The data display pane is updated as each statement is executed, so changes in values can be observed as the script is running.



Vectors, array elements, and structure members can also be observed in the data display pane during script execution. Items containing more than one storage location are shown with a disclosure triangle to the left of the item name. To view the storage locations contained within the item, click the disclosure triangle; the individual locations and the values contained in them will be displayed below the item name.



Complex data types can also be viewed in the data pane

The view within data pane can be resized to accommodate data display. To resize the value display, move the cursor over the divider line, then drag the divider to the desired location.

Numeric and Data Formats

In this Chapter:

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

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

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.

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 sting length, spaces will be prepended to the value.

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.

Search Criteria

In this Chapter:

Search criteria are designed for use with VectorScripts' criteria API and with worksheets to filter and locate objects by the specified attribute values. Search criteria use the attributes of VectorWorks objects (layer, class, color, lineweight, etc...) as a means of selecting and manipulating subsets of items within the document.

Search Criteria Format

Syntax

Search criteria in VectorScript are composed of two parts: the **search attribute type specifier** and the **search value**. The search attribute specifier indicates which attribute will be used to filter objects in the document; the search value specifies the value to be found and matched by the search operation. For example, the search criteria term:

```
(C='Edged')
```

indicates that a search should be performed for any objects whose class is `Edged`. In the criteria term, the `C` attribute type indicates that the search should be performed on the class attribute of objects in the document. The search value `Edged` indicates what class will be a match in the search operation.

The general syntax for search criteria terms is:

```
(<search attribute type> = <search value>)
```

Parentheses are traditionally used to enclose and indicate individual search terms; they are not required.

Multiple Search Terms

Multiple criteria terms may be specified in order to narrow the search operation to a more specific subset of objects. Multiple search criteria are created using the & operator to chain individual search criteria terms. In the term

```
((L='New Construction') & (C='Phase 1'))
```

two search terms are combined to filter for a specific set of objects, in this case any objects on the layer `New Construction` whose class is `Phase 1`. To narrow the search even further, simply add additional search terms:

```
((L='New Construction') & (C='Phase 1') & (SEL=TRUE))
```

In the example, the selection status attribute type was added, so now only selected objects in the `Phase 1` class on layer `New Construction` will match the search.

Multiple Search Values

It is also possible to filter for multiple match values using search criteria. Multiple match values use the following syntax:

```
(<attribute type> IN [<search value>,<search value>,...])
```

When a search term is specified in this fashion, objects matching any value in the comma delimited value list will be included in the list of objects matching the search. For example:

```
(R IN ['Part Data','Subassembly Data','Assembly Data'])
```

A search using the above term will match any objects with an attached record matching one of the records in the search list.

Attribute Types

Markers (AR)

The marker attribute type will search for the indicated marker style. The search value should be one of the supported marker style flag selector values (in a range of 0 – 27).

Class (C)

The class attribute type will search for objects assigned to the specified class. The search value should be a `STRING` value which is up to 64 characters in length (literals and variables are supported).

Fill Background (FB)

The fill background attribute type will search for objects having the specified fill background. The search value should be a standard VectorWorks color index value (which can be obtained with `RGBToColorIndex()`).

Fill Foreground (FF)

The fill foreground attribute type will search for objects having the specified fill foreground. The search value should be a standard VectorWorks color index value (which can be obtained with `RGBToColorIndex()`).

Fill Pattern (FP)

The fill pattern attribute type will search for objects having the specified fill pattern. The search value should be the standard VectorWorks fill pattern selector value (in a range of 0 – 71).

Layer (L)

The layer attribute type will search for objects on the specified layer. The search value should be a `STRING` value which is up to 64 characters in length (literals and variables are supported).

Line Weight (LW)

The line weight attribute specifier will search for objects which have the indicated line weight. The search value should be an `INTEGER` value specifying the line weight.

Pen Pattern/Linestyle (PP)

The pen pattern/linestyle attribute specifier will search for objects having the indicated linestyle or pen pattern. The search value should be a standard linestyle or pen pattern selector value.

Object Name (N)

The object name attribute specifier will search for the object which is assigned the specified object name. The search value should be a `STRING` value which is up to 64 characters in length (literals and variables are supported).

Attached Record (R)

The record attribute specifier will search for objects which have the indicated record attached.

The record attribute specifier requires the use of the multiple criteria format to specify the record name. For example, to search for objects having the Part Data record attached, the search term would be:

```
(R IN ['Part Data'])
```

The record name must be a literal `STRING` value.

Object Type (T)

The object type attribute specifier will search for objects matching the specified object type. The search value must be one of the predefined object type selectors (see table at the end of this section for a complete listing).

Pen Background (PB)

The pen background attribute specifier will search for objects having the specified pen background. The search value should be a standard VectorWorks color index value (which can be obtained with `RGBToColorIndex()`).

Pen Foreground (PF)

The pen foreground attribute specifier will search for objects having the specified pen foreground. The search value should be a standard VectorWorks color index value (which can be obtained with `RGBToColorIndex()`).

Selection Status (SEL)

The selection status specifier will search for selected or deselected objects. The search value is a BOOLEAN value indicating the selection state (TRUE for selected, FALSE for deselected).

Symbol Name (S)

The symbol name attribute specifier will search for symbol instances based on the specified symbol name. The search value should be a STRING value which is up to 64 characters in length (literals and variables are supported).

Visibility (V)

The visibility attribute specifier will search for objects based on their visibility status. The search value is a BOOLEAN value indicating the visibility state (TRUE for visible, FALSE for invisible).

Specialized Searches

In addition to the standard attribute types available for use in search terms, VectorScript also provides specialized search attribute types for additional flexibility in searching a document.

Record Field Values

Record fields may be searched for specific matching values using a specialized attribute type to query the field value. The syntax for querying record fields is:

```
(<record name>.<field name>[< = | <> | > | >= | < | <= ><search value>])
```

The record and field names are STRING values and should be enclosed in single quotes. Any one of the optional comparison operators can be used to focus the search on a specific subset of items which have the attached record. For example:

```
('Assembly Data'.'Base Cost' < 250.00)
```

will search for any items with the attached record whose base cost is less than 250.00 dollars.

Search Symbol Instances (INSYMBOL)

The INSYMBOL attribute specifier will cause the search to enter any symbols encountered and perform a search on the symbols' definition. For example, suppose you are laying out a large office and wish to count the total number of desk components that will need to be purchased. Your document contains a mixture of individual desk and desk return symbols, plus symbols which are comprised of a combination of the two desk components. A search using the term

```
(S IN ['3660 Desk','3660 LH Return'])
```

will return an inaccurate count, as it does not include instances of those symbols which are themselves inside another symbol. Adding the INSYMBOL type specifier to the term:

```
((S IN ['3660 Desk','LH Return']) & (INSYMBOL))
```

will force the search to enter any symbols encountered and detect any nested instances of the symbols in the search term.

Symbol Flip Status (ISFLIPPED)

The ISFLIPPED attribute specifier will check the flipped status of symbols or other objects. For example, to perform a count of all flipped instances of a particular symbol:

```
((S=’3680 Door’) & (ISFLIPPED))
```

will find only those instances of the symbol which have been flipped. The ISFLIPPED specifier is useful for determining orientation of objects for editing or related tasks.

All Objects (ALL)

Using the ALL attribute type specifier will select all the objects in the document.

Search Criteria Tables

The VectorScript criteria attribute specifiers are listed in the following table.

Attribute Type	Type Specifier	Example
Marker	AR	INTEGER selector value
Class	C	64 character STRING
Fill Background	FB	Color index value
Fill Foreground	FF	Color index value
Fill Pattern	FP	INTEGER selector value
Layer	L	64 character STRING
Line Weight	LW	INTEGER value
Pen Pattern/Linestyle	PP	INTEGER value
Object Name	N	64 character STRING
Attached Record	R	64 character STRING
Object Type	T	Type selector (see table)
Pen Background	PB	Color index value
Pen Foreground	PF	Color index value
Selected status	SEL	BOOLEAN value
Symbol Name	S	64 character STRING
Visibility status	V	BOOLEAN value
Descend into symbols	INSYMBOL	n/a
Flipped status	ISFLIPPED	n/a
All objects	ALL	n/a

Object Type	Type Selector	Example
Line	LINE	T=LINE
Rectangle	RECT	T=RECT
Rounded Rectangle	RRECT	T=RRECT
Oval	OVAL	T=OVAL
Polygon	POLY	T=POLY
Polyline	POLYLINE	T=POLYLINE
Arc	ARC	T=ARC
Quarter Arc	QARC	T=QARC
Text	TEXT	T=TEXT

Object Type	Type Selector	Example
2D Locus	LOCUS	T=LOCUS
3D Locus	LOCUS3D	T=LOCUS3D
Freehand	FHAND	T=FHAND
Dimension	DIMENSION	T=DIMENSION
Symbol	SYMBOL	T=SYMBOL
Group	GROUP	T=GROUP
Extrude	XTRD	T=XTRD
Multiple Extrude	MXTRD	T=MXTRD
Sweep	SWEEP	T=SWEEP
Mesh	MESH	T=MESH
3D Polygon	POLY3D	T=POLY3D
Cone, Sphere, Pyramid	SOLID	T=SOLID
CSG Solid	CSGSOLID	T=CSGSOLID
Wall	WALL	T=WALL
Round Wall	ROUNDWALL	T=ROUNDWALL
Roof	ROOF	T=ROOF
Roof Element	ROOFELEMENT	T=ROOFELEMENT
Roof Face, Floor, Column	SLAB	T=SLAB
Worksheet	SPRDSHEET	T=SPRDSHEET
Layer Link	LAYERLINK	T=LAYERLINK
PICT Image	PICT	T=PICT
Bitmap Image	BITMAP	T=BITMAP
Plug-in Object	PLUGINOBJECT	T=PLUGINOBJECT

Compiler Directives

In this Chapter:

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

{$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 Plug-ins folder.

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 may be found in “The VectorScript Debugger” on page 17-10.

{$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.

Object Types

In this Chapter:

- Standard Types

Standard Types

The numeric types in the table below are useful for identifying what type of object is referenced by a handle. The function `GetType(h)` will return one of these numeric types. The Criteria values in the table below are used in search statements. They are used along with the criteria `T=` to search for objects of a specific type. For example, the following statement will count the number of rectangles in the active document: `Message(Count(T=RECT));`

Object	Type	Criteria
Line	2	LINE
Rectangle	3	RECT
Oval	4	OVAL
Polygon	5	POLY
Arc	6	ARC
Freehand	8	FHAND
3D Locus	9	LOCUS3D
Text	10	TEXT
Group	11	GROUP
Quarter Arc	12	QARC
Rounded rectangle	13	RRECT
Bitmap Image	14	BITMAP
Symbol in document	15	SYMBOL
Symbol definition	16	
2D Locus	17	LOCUS
Worksheet	18	SPRDSHEET
Polyline	21	POLYLINE
PICT Image	22	PICT

Object	Type	Criteria
Extrude	24	XTRD
3D Polygon	25	POLY3D
Layer link	29	LAYERLINK
Layer	31	
Sweep	34	SWEEP
Multiple extrude	38	MXTRD
Mesh	40	MESH
Mesh vertex	41	
Record definition (format)	47	
Record	48	
Document script [1]	49	
Script palette [1]	51	
Worksheet container	56	
Dimension	63	DIMENSION
Hatch definition [1]	66	
Wall	68	WALL
Column, floor, roof face	71	SLAB
Light	81	
Roof edge	82	
Roof object	83	ROOF
CSG solid (addition, subtraction)	84	CSGSOLID
Plug-in object	86	PLUGINOBJECT
Roof element	87	ROOFELEMENT
Round walls	89	ROUNDWALL
Symbol folder	92	
Texture	93	
Class definition [1]	94	
Solid (cone, sphere, ...)	95	SOLID
Texture definition (material)	97	
NURBS curve	111	
NURBS surface	113	
Image fill definition [1]	119	

Object	Type	Criteria
Gradient fill definition [1]	120	
Fill space [1]	121	

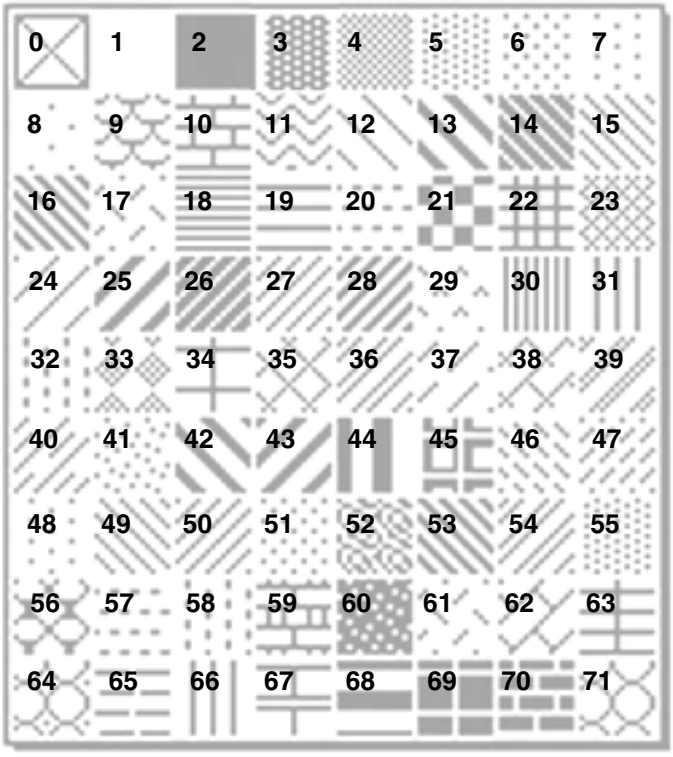
Note:

[1] *These special objects are not directly displayed in the document. They may contain definition information used by other objects or features.*

Selector Tables

In this Chapter:

Fill Patterns



Linestyles

Style	Style Selector	Example
Short Dash	-1	
Medium Dash	-2	
Long Dash	-3	
Very Long Dash	-4	
Dotted	-5	
Dash-Dot	-6	
Dash-Dash-Dot	-7	
Dash-Dot-Dot	-8	
Centerline	-9	
Break Line	-10	

Markers

Marker Type	Marker Location	Selector
Solid Arrow	None	0
	Start	1
	End	2
	Both	3
Hollow Arrow	None	4
	Start	5
	End	6

Marker Type	Marker Location	Selector
	Both	7
Open Arrow	None	8
	Start	9
	End	10
	Both	11
Dot	None	12
	Start	13
	End	14
	Both	15
Circle	None	16
	Start	17
	End	18
	Both	19
Slash	None	20
	Start	21
	End	22
	Both	23
Cross	None	24
	Start	25
	End	26
	Both	27

SetTool - CallTool Selectors

Tool	Selector	Tool	Selector
2D Selection Cursor	-240	3D View Rotate	-301
3D Selection Cursor	-349	Walkthrough	-302
Pan	-241	Flyover	-303
Zoom In	-242	Translate Working Plane	-304
Zoom Out	-243	Rotate Working Plane	-305
Text	-200	Set Working Plane	-306
Line	-201	Move Working Plane	-307
Arc	-202	3D Reshape	-308
Rectangle	-203	3D Symbol	-309
Polyline	-204	Extrude	-310
Oval	-205	Slab	-311
Fillet	-206	Align Plane	-312
2D Polygon	-207	3D Polygon	-313
Wall	-208	3D Mirror	-314
2D Symbol	-209	3D Rotate	-315
Constrained Dimension	-210	3D Locus	-316
Unconstrained Dimension	-211	Create Light	-317
Radial-Diam Dimension	-212	Round Wall	-318
Angular Dimension	-213	Create Sphere	-319
2D Reshape	-214	Create Hemisphere	-320
2D Rotate	-215	Create Cone	-321
Double Line	-216	Wall Join	-322
Rounded Rectangle	-217	Wall Heal	-323
Double Line Polygon	-218	Render Bitmap	-324
Chamfer	-219	NURBS Curve	-325
Freehand	-220	Curve Split	-326
2D Locus	-221	Connect/Combine	-327
2D Mirror	-222	NURBS Circle	-328
Leader	-223	NURBS Arc	-329
Rotated Rectangle	-224	Extract Curve	-342
Regular Polygon	-225	Blend Edge	-343
Clipping	-226	Loft	-344

Tool	Selector	Tool	Selector
Quarter Arc	-227	Project and Trim	-345
Center Mark	-228	Extract Surface	-346
Number Stamp	-229	Shell Solid	-347
3D View Translate	-300	Create Contours	-348

Record Field Data Type Selectors

Field Data Type	Selector
INTEGER	1
BOOLEAN	2
Number - General	3
Text	4
Number - Decimal	5
Number - Decimal with commas	6
Number - Scientific	7
Number - Fractional	8
Dimension	9
Angle	10
Date-Time	11

Record Field Display Style Selectors

Field Type	Display Style	Style Selector
BOOLEAN	TRUE	1
	FALSE	2
Number - Decimal	No. of decimal places	0 to 9
Number - Decimal with commas	No. of decimal places	0 to 9
Number - Scientific	No. of decimal places	0 to 9
Number - Fractional	Rounding value	2,4,8,16,32...
Angle	Degrees	1
	Degrees-Minutes	2

Field Type	Display Style	Style Selector
	Degrees-Minutes-Seconds	3
Date-Time	MDY	1
	MDY HMM	2
	DMY	3
	YMD	4
	YMD HMM	5
	D-MMM-Y	6
	D-MMM	7
	MMM-Y	8
	H MM	9
	H MM S	10
	H MM(AM/PM)	11
	H MM S(AM/PM)	12

Dimension Style Selectors

Dimension style selectors control the dimension display options for the VectorScript API functions `LinearDim`, `CircularDim`, and `AngularDim`. Display selectors are additive; multiple options can be combined by adding the selector values and using the result in the appropriate parameter. For example,

```
LinearDim(-2",-2",1",2",-3",0,769,1025,0);
```

will create a constrained horizontal linear dimension with starting and ending witness lines, arrows using the calculated default location, and text which will always display horizontally.

Linear Dimension

Parameter	Style	Selector
dimType	Constrained Horizontal	0
	Constrained Vertical	1
	Horizontal Ordinate	2
	Vertical Ordinate	3
	Unconstrained	4
arroFlag	Use calculated position	1
	Dimension line inside/outside	2
	Start witness on/off	256
	End witness on/off	512
textFlag	Use calculated position	1
	Text inside/outside	2
	Text above dimension line	256
	Text aligned to dimension line	512
	Force text horizontal	1024

Circular Dimension

Parameter	Style	Selector
dimType	Diameter dimension	0
	Other circular dimension	1
arroFlag	Use calculated position	1
	Dimension line inside/outside	2
	Start witness on/off	256
	End witness on/off	512
textFlag	Use calculated position	1
	Text inside/outside	2
	Text above dimension line	256
	Text aligned to dimension line	512
	Force text horizontal	1024

Angular Dimension

Parameter	Style	Selector
arroFlag	Use calculated position	1
	Dimension line inside/outside	2
	Start witness on/off	256
	End witness on/off	512
	Reference Angle	1024
textFlag	Use calculated position	1
	Text inside/outside	2
	Text above dimension line	256
	Text aligned to dimension line	512
	Force text horizontal	1024

Preference Selectors

In this Chapter:

Using Preference Selectors

VectorScript provides a group of API function calls for obtaining and setting document and application preferences. The `GetPref` and `SetPref` suite of API function calls use a selector index value to obtain or set the desired preference value. For example, to determine whether the preference for displaying other objects while in groups is active, the function:

```
showOthers:= GetPref(14);
```

will return the status of this setting. Setting the preference can be performed by using the corresponding SetPref call; for example, the function call:

```
SetPref(14,TRUE);
```

will cause other objects to be displayed while in edit group mode.

Non-boolean setting values can be obtained by using other functions within the group of functions. For example, to determine the 2D conversion resolution, using the function:

```
convRes:= GetPrefInt(55);
```

will return the conversion value. To set the value, use the corresponding function call:

```
SetPrefInt(55,32);
```

Preference Selector Value Tables

The following tables list selector values for various VectorWorks application and document preferences.

General Application/Document Preferences

Preference	Selector	Preference Value	Function
Click-Drag Mode	0	TRUE or FALSE	Pref
Offset Duplicates	1	TRUE or FALSE	Pref
Full Screen Cursor	2	TRUE or FALSE	Pref
Show Screen Hints	3	TRUE or FALSE	Pref
Floating Datum	4	TRUE or FALSE	Pref
Snap To Loci	5	TRUE or FALSE	Pref
Show Rulers	6	TRUE or FALSE	Pref
Show Scroll Bars	7	TRUE or FALSE	Pref
No Fill Behind Text	8	TRUE or FALSE	Pref
Zoom Line Thickness	9	TRUE or FALSE	Pref
Black and White Only	10	TRUE or FALSE	Pref
Use Layer Colors	11	TRUE or FALSE	Pref
Log Time in Program	12	TRUE or FALSE	Pref
Adjust Flipped Text	13	TRUE or FALSE	Pref
Show Other Objects While In Group	14	TRUE or FALSE	Pref
Show 3D Axis Labels	15	TRUE or FALSE	Pref
Use Black Background	16	TRUE or FALSE	Pref
Use Eight Selection Handles	17	TRUE or FALSE	Pref
Use Sound	18	TRUE or FALSE	Pref
Issue Undo Warnings	19	TRUE or FALSE	Pref
Opaque SmartCursor	20	TRUE or FALSE	Pref
Stop VectorScript on Warnings	21	TRUE or FALSE	Pref
Left Palette Margin	22	TRUE or FALSE	Pref
Right Palette Margin	23	TRUE or FALSE	Pref
Use Save Reminder	24	TRUE or FALSE	Pref
Show Parametric Constraints	25	TRUE or FALSE	Pref
Undo View Changes	26	1 (never) 2 (combine all) 3 (combine similar) 4 (combine none)	PrefInt
Display Minor Alerts on Mode Bar	27	TRUE or FALSE	Pref
Associate Dimensions	28	TRUE or FALSE	Pref
Spell Check Capitalized Words	29	TRUE or FALSE	Pref

Preference	Selector	Preference Value	Function
Spell Check Words in ALL CAPS	30	TRUE or FALSE	Pref
Spell Check Mixed Case Words	31	TRUE or FALSE	Pref
Spell Check Words With Numbers	32	TRUE or FALSE	Pref
Auto Join Walls	33	TRUE or FALSE	Pref
Show Page Breaks	34	TRUE or FALSE	Pref
Show Grid	35	TRUE or FALSE	Pref
Print Grid	36	TRUE or FALSE	Pref
Snap To Grid	37	TRUE or FALSE	Pref
Snap To Object	38	TRUE or FALSE	Pref
Save By Time	39	TRUE or FALSE	Pref
Save Confirm	40	TRUE or FALSE	Pref
Save To Backup	41	TRUE or FALSE	Pref
Extended Autoscroll	42	TRUE or FALSE	Pref
Palette Docking	43	TRUE or FALSE	Pref
Dimension Slash Thickness Unit	50	3 (points) 2 (mils) 1 (mm)	PrefInt
3D Rotation Responsiveness	52	1(detailed)..5(responsive)	PrefInt
Custom Constraint Angle	53	REAL (degrees)	PrefReal
Snap Radius	54	INTEGER value	PrefInt
2D Conversion Resolution	55	INTEGER value	PrefInt
3D Conversion Resolution	56	INTEGER value	PrefInt
Current Document Text Size	57	REAL	PrefReal
Current Document Text Style	58	0 (Plain) 1 (Bold) 2 (Italic) 4 (Underline) 8 (Outline - Mac only) 16 (Shadow - Mac only)	PrefInt
Maximum Number of Undos	59	INTEGER	PrefInt
Save Interval	60	no. of minutes	PrefInt
Display Light Objects	61	0(always) 1(wireframe) 2(never)	PrefInt
Retain QuickDraw 3D Model	62	1(never)...5(always)	PrefInt

Preference	Selector	Preference Value	Function
Rotated Text Display	63	0(box) 1(normal) 2(high)	PrefInt
Bitmap Display	64	0(box) 1(low res) 2(hi res)	PrefInt
Dimension Slash Thickness	65	INTEGER value (mils)	PrefInt
Hidden Line Dash Style	66	INTEGER selector	PrefInt
Hidden Line Shading	67	1(dark)...4(light)	PrefInt
Page Origin X	68	REAL	PrefReal
Page Origin Y	69	REAL	PrefReal
Page Scaling Factor	70	REAL	PrefReal
Dimension Standard	71	0 (Arch) 1 (ASME) 2 (BSI) 3 (DIN) 4 (ISO) 5 (JIS) 6 (SIA) 7 (ASME Dual Side-by-Side) 8 (ASME Dual Stacked)	PrefInt
Defacet Angle	72	REAL (0-90 degrees)	PrefReal
Grid Angle	73	REAL	PrefReal
Move Object on Grid Keys	74	1 (arrow) 2 (Cmd+arrow) 3 (Shift+arrow) 4 (Shift+Cmd+arrow)	PrefInt
Nudge Object Keys	75	1 (arrow) 2 (Cmd+arrow) 3 (Shift+arrow) 4 (Shift+Cmd+arrow)	PrefInt
Pan Drawing Keys	76	1 (arrow) 2 (Cmd+arrow) 3 (Shift+arrow) 4 (Shift+Cmd+arrow)	PrefInt
Switch Active Layer/Class Keys	77	1 (arrow) 2 (Cmd+arrow) 3 (Shift+arrow) 4 (Shift+Cmd+arrow)	PrefInt
Text Font Name	100	STRING	PrefString

Preference	Selector	Preference Value	Function
Angular Precision	120	INTERGER	PrefInt
Angular Unit	121	0 (degrees) 1 (radians) 2 (gradians)	PrefInt

Primary Units

Preference	Selector	Preference Value	Function
Unit Fraction	150	REAL value	PrefReal
Units Per Inch	152	REAL value	PrefReal
Unit Style Name	153	64 character STRING	PrefString
Unit Mark	154	STRING value	PrefString
SUnit Mark	155	STRING value	PrefString
SUnit Divider	156	STRING value	PrefString
SMultiplier	157	INTEGER value	PrefInt
Square Unit Mark	158	STRING value	PrefString
Square Unit Divisor	159	LONGINT	PrefLongint
Cube Unit Mark	160	STRING value	PrefString
Cube Unit Divisor	161	LONGINT	PrefLongint
Display Fraction	162	LONGINT	PrefLongint
Show Unit Mark	163	TRUE or FALSE	Pref
Display Leading Zero	164	TRUE or FALSE	Pref
Display Trailing Zeroes	165	TRUE or FALSE	Pref
Use Minimum Units	166	TRUE or FALSE	Pref
Use Custom Units	167	TRUE or FALSE	Pref
Show Decimals as Fractions	168	TRUE or FALSE	Pref
Dimension Precision	169	LONGINT	PrefLongint
Predefined Units Style	170	0 (Custom) 1 (Feet & Inches) 2 (Feet) 3 (Inches) 4 (Millimeters) 5 (Centimeters) 6 (Meters)	PrefInt
Fractional Display Precision	171	LONGINT value	PrefLongInt
Fractional Dimension Precision	172	LONGINT value	PrefLongInt

Preference	Selector	Preference Value	Function
Metric Unit Flag	173	TRUE or FALSE	Pref
Angular Unit	174	0 (degrees) 1 (radians) 2 (gradians)	PrefInt
Round Fraction to Decimal	175	TRUE or FALSE	Pref

Secondary Units

Preference	Selector	Preference Value	Function
Unit Fraction	200	REAL value	PrefReal
Units Per Inch	202	REAL value	PrefReal
Unit Style Name	203	64 character STRING	PrefString
Unit Mark	204	STRING value	PrefString
SUnit Mark	205	STRING value	PrefString
SUnit Divider	206	STRING value	PrefString
SMultiplier	207	INTEGER value	PrefInt
Square Unit Mark	208	STRING value	PrefString
Square Unit Divisor	209	LONGINT	PrefLongint
Cube Unit Mark	210	STRING value	PrefString
Cube Unit Divisor	211	LONGINT	PrefLongint
Display Fraction	212	LONGINT	PrefLongint
Show Unit Mark	213	TRUE or FALSE	Pref
Display Leading Zero	214	TRUE or FALSE	Pref
Display Trailing Zeroes	215	TRUE or FALSE	Pref
Use Minimum Units	216	TRUE or FALSE	Pref
Use Custom Units	217	TRUE or FALSE	Pref
Show Decimals as Fractions	218	TRUE or FALSE	Pref
Dimension Precision	219	LONGINT	PrefLongint
Predefined Units Style	220	0 (Custom) 1 (Feet & Inches) 2 (Feet) 3 (Inches) 4 (Millimeters) 5 (Centimeters) 6 (Meters)	PrefInt
Fractional Display Precision	221	LONGINT value	PrefLongInt

Preference	Selector	Preference Value	Function
Fractional Dimension Precision	222	LONGINT value	PrefLongInt
Metric Unit Flag	223	TRUE or FALSE	Pref
Angular Unit	224	0 (degrees) 1 (radians) 2 (gradians)	PrefInt
Round Fraction to Decimal	225	TRUE or FALSE	Pref

DXF Preference Selectors

Preference	Selector	Preference Data Type	Function
Auto Units	300	TRUE or FALSE	Pref
Units	301	INTEGER	PrefInt
DXF Units Per Inch	302	REAL	PrefReal
Auto Model Space Scale	303	TRUE or FALSE	Pref
Model Space Scale	304	REAL	PrefReal
2D 3D Import Handling	305	INTEGER	PrefInt
Map Layers to Class	306	TRUE or FALSE	Pref
Convert MLines to Walls	307	TRUE or FALSE	Pref
Convert Rays and XLines	308	TRUE or FALSE	Pref
Scale Dash Lengths	309	TRUE or False	Pref
Dash Length Scale	310	REAL	PrefReal
Auto Block Attribute Handling	311	TRUE or FALSE	Pref
Block Attribute Handling	312	INTEGER	PrefInt
Auto Point Handling	313	TRUE or FALSE	Pref
Convert Points to Loci	314	TRUE or FALSE	Pref
Point Symbols are Guides	315	TRUE or FALSE	Pref
Map Colors to Line Weights	316	TRUE or FALSE	Pref
Set Line Colors Black	317	TRUE or FALSE	Pref
Paper Space Units	318	INTEGER	PrefInt
Auto Scale Dash Lengths	319	TRUE or FALSE	Pref
Group Record Fields	320	TRUE or FALSE	Pref
Auto Line Weight Handling	321	TRUE or FALSE	Pref

Gradient and Image Fill Preference Selectors

Preference	Selector	Preference Data Type	Function
Default Gradient Fill	508	LONGINT	PrefLongint
Default Gradient Fill Angle	512	REAL	PrefReal
Default Gradient Fill Repeat	513	TRUE or FALSE	Pref
Default Gradient Fill Geometric Type	515	INTEGER	PrefInt
Default Gradient Fill Mirror	516	TRUE or FALSE	Pref
Default Gradient Fill Maintain Aspect Ratio	517	TRUE or FALSE	Pref
Default Image Fill	518	LONGINT	PrefLongint
Default Image Fill I-Length	521	REAL	PrefReal
Default Image Fill J-Length	522	REAL	PrefReal
Default Image Fill Angle	523	REAL	PrefReal
Default Image Fill Repeat	524	TRUE or FALSE	Pref
Default Image Fill Mirror	526	TRUE or FALSE	Pref
Default Image Fill Flip	527	TRUE or FALSE	Pref
Default Fill Style	528	LONGINT	PrefLongint
Default Fill Type	529	INTEGER	PrefInt
Default Hatch Fill	530	LONGINT	PrefLongint

Miscellaneous Preference Selectors

Preference	Selector	Preference Data Type	Function
RenderWorks Enabled	240	TRUE or FALSE	GetPref
Disable RenderWorks	241	TRUE or FALSE	Pref
Don’t Cache Plug-in Scripts	407	TRUE or FALSE	Pref
Window Zoom Factor	500	REAL	PrefReal

Object Selectors

In this Chapter:

Object Variable Selectors

VectorScript provides a group of API functions for obtaining and modifying selected object settings. The `GetObjectVariable` and `SetObjectVariable` API function calls use a selector index value to obtain or set the desired object setting value.

For example, to determine whether a light casts shadows, the statement:

```
b:= GetObjectVariableBoolean(h,53);
```

will return the shadow casting status of the referenced light object and assign it to the variable `b`. To set the shadow casting status, the statement:

```
SetObjectVariableBoolean(h,FALSE);
```

would turn shadow casting off. To get the type of the same light, the statement:

```
t:= GetObjectVariableInt(h,55);
```

will return the type of the light and assign it to the variable. To set the light type, the statement:

```
SetObjectVariableInt(h,55,2);
```

will set the referenced light object to be a point light source.

Setting Selector Value Tables

The following tables list the setting selector values for various VectorWorks object types.

Dimension

Object Setting	Selector	Setting Value	Function
Dimension Standard	0	0 (Arch) 1 (ASME) 2 (BSI) 3 (DIN) 4 (ISO) 5 (JIS) 6 (SIA) 7 (ASME Dual Side-by-Side) 8 (ASME Dual Stacked)	ObjectVariableInt
Dimension Text Rotation	1	0(aligned) 1(horizontal only) 2(horizontal-vertical)	ObjectVariableInt
Dim Text Offset Above Line	2	REAL value	ObjectVariableReal
Arrows Inside	3	TRUE or FALSE	ObjectVariableBoolean
Dim Text Offset	4	REAL value	ObjectVariableReal
Use Text Box (Primary Value)	5	TRUE or FALSE	ObjectVariableBoolean
Show Primary Dimension Text	6	TRUE or FALSE	ObjectVariableBoolean
Display Starting Witness Line	7	TRUE or FALSE	ObjectVariableBoolean
Display Ending Witness Line	8	TRUE or FALSE	ObjectVariableBoolean
Leader Text (Primary)	9	31 character STRING value	ObjectVariableString
Trailer Text (Primary)	10	31 character STRING value	ObjectVariableString
Dimension Tolerancing	11	0(no tolerance) 1(single tolerance) 2(double tolerance) 3(limit tolerance)	ObjectVariableInt
Dimension Offset	15	REAL value	ObjectVariableReal
Dimension Text Font Size	17	in point size	ObjectVariableInt
Dimension Text Font Style	19	0 (Plain) 1 (Bold) 2 (Italic) 4 (Underline) 8 (Outline [Mac only]) 16 (Shadow [Mac only])	ObjectVariableInt
Dimension Precision (Primary)	20	INTEGER selector value	ObjectVariableInt
Dimension Precision (Secondary)	21	INTEGER selector value	ObjectVariableInt
Use Text Box (Secondary)	22	TRUE or FALSE	ObjectVariableBoolean
Show Secondary Dimension Text	23	TRUE or FALSE	ObjectVariableBoolean

Object Setting	Selector	Setting Value	Function
Leader Text (Secondary)	24	31 character STRING value	ObjectVariableString
Trailer Text (Secondary)	25	31 character STRING value	ObjectVariableString
Dimension Type	26	0 (Constrained) 1 (Unconstrained) 2 (Ordinate) 3 (Radial) 4 (Diameter) 5 (Angular)	ObjectVariableInt
Dimension Standard Name	27	STRING value	ObjectVariableString
Dimension Font ID	28	Font ID	ObjectVariableInt
Calculate Dim Text Position	29	TRUE or FALSE	ObjectVariableBoolean
Force Dim Text Inside	30	TRUE or FALSE	ObjectVariableBoolean
Angle is Reference	31	TRUE or FALSE	ObjectVariableBoolean
Show only Primary	32	TRUE or FALSE	ObjectVariableBoolean
Show only Secondary	33	TRUE or FALSE	ObjectVariableBoolean
Top Tolerance Value	34	REAL value	ObjectVariableReal
Bottom Tolerance Value	35	REAL value	ObjectVariableReal
Top Tolerance String	36	STRING	ObjectVariableString
Bottom Tolerance String	37	STRING	ObjectVariableString
Use Tolerance Strings	38	TRUE or FALSE	ObjectVariableBoolean
Flip Text	39	TRUE or FALSE	ObjectVariableBoolean

Lights

Object Setting	Selector	Setting Value	Function
Light On	50	TRUE or FALSE	ObjectVariableBoolean
Brightness	51	REAL (percentage)	ObjectVariableReal
Shadow Casting On	53	TRUE or FALSE	ObjectVariableBoolean
Light Type	55	1(Directional) 2(Point) 3(Spotlight)	ObjectVariableInt
Light Pan Angle	57	REAL value	ObjectVariableReal
Light Tilt Angle	58	REAL value	ObjectVariableReal
Distance Falloff Type	59	0 (None) 1 (Smooth) 2 (Sharp)	ObjectVariableInt

Object Setting	Selector	Setting Value	Function
Angular Falloff Type	60	0 (none) 1 (Normal) 2 (Smooth) 3 (Sharp)	ObjectVariableInt
Light Spread Angle	61	REAL value	ObjectVariableReal
Beam Angle	62	REAL value	ObjectVariableReal

Symbol/Symbol Definitions

Object Setting	Selector	Setting Value	Function
Symbol Light Multiplier	100	REAL value	ObjectVariableReal
Symbol Insert Mode	125	0 (On center of wall) 1 (On edge of wall)	ObjectVariableInt
Symbol Break Mode	126	1 (Full break) 2 (Full break no caps) 3 (Half break) 4 (no break)	ObjectVariableInt
Insert As Group	127	TRUE or FALSE	ObjectVariableBoolean

Roof/Floors/Columns

Object Setting	Selector	Setting Value	Function
Slab Thickness	170	REAL value	ObjectVariableReal
Slab Height	171	REAL value[1]	ObjectVariableReal
Slab Type	172	1 (Roof) 2 (Floor) 3 (Column)	ObjectVariableInt
Roof Rise	178	REAL value[2]	ObjectVariableReal
Roof Run	179	REAL value[2]	ObjectVariableReal
Roof Edge Miter Type	180	1 (Vertical) 2 (Horizontal) 3 (Compound)	ObjectVariableInt
Double Miter Ratio Value	181	REAL value[3]	ObjectVariableReal

Notes:

[1] *Height is the bottom of the slab for floors and columns, elevation of the roof axis for roofs.*

[2] *Roof only.*

[3] *A value between 0 and 1 indicating the percentage of the miter which is vertical.*

Layers

Object Setting	Selector	Setting Value	Function
Layer Ambient Status	150	TRUE or FALSE	ObjectVariableBoolean
Layer Ambient Brightness	151	REAL value	ObjectVariableReal
Layer Visibility	153	-1 (Invisible) 0 (Normal) 2 (Grayed)	ObjectVariableInt

Layer Link

Object Setting	Selector	Setting Value	Function
Source Layer Name	160	STRING value	ObjectVariableString
Projects 2D Objects	161	TRUE or FALSE	ObjectVariableBoolean

Walls/Wall Cavities

Object Setting	Selector	Setting Value	Function
Number of Cavities	199	INTEGER value[1]	ObjectVariableInt
Cavity Left Offset	200	REAL value[2]	ObjectVariableReal
Cavity Right Offset	220	REAL value[2]	ObjectVariableReal
Cavity is Pair	240	TRUE or FALSE	ObjectVariableBoolean
Cavity Fill Pattern	260	LONGINT index (0-71)[2]	ObjectVariableLongint
Cavity Pen Weight	280	INTEGER value (mils)[2]	ObjectVariableInt
Cavity Pen Style	300	INTEGER index[2]	ObjectVariableInt
Counterclockwise Round Wall	570	TRUE or FALSE	ObjectVariableBoolean
Main Cavity Index	690	INTEGER	ObjectVariableInt

Notes:

[1] *Pass* `NIL` *to access default cavity values.*

[2] *To access different cavities within a wall, add the cavity index to the selector value. For example, to access the right offset of cavity 6, specify 226 (220 + 6).*

Plug-in Objects

Object Setting	Selector	Setting Value	Function
Insertion Mode	123	0 (On center of wall) 1 (On edge of wall)	ObjectVariableInt
Break Mode	124	1 (Full break) 2 (Full break no caps) 3 (Half break) 4 (no break)	ObjectVariableInt
Font Style Enabled	800	TRUE or FALSE	ObjectVariableBoolean

2D/3D Status

Object Setting	Selector	Setting Value	Function
Object Is 3D	650	TRUE or FALSE (read-only)	ObjectVariableBoolean
Object Is 2D	651	TRUE or FALSE (read-only)	ObjectVariableBoolean

Worksheets

Object Setting	Selector	Setting Value	Function
Worksheet Header	80	STRING value	ObjectVariableString
Worksheet Footer	81	STRING value	ObjectVariableString
Show Database Headers	82	TRUE or FALSE	ObjectVariableBoolean
Show Gridlines	83	TRUE or FALSE	ObjectVariableBoolean
Show Tabs	84	TRUE or FALSE	ObjectVariableBoolean
Auto-Recalculate	85	TRUE or FALSE	ObjectVariableBoolean
Default Font Index	86	INTEGER value	ObjectVariableInt
Default Font Size	87	INTEGER value	ObjectVariableInt
Top Print Margin	88	REAL value	ObjectVariableReal
Left Print Margin	89	REAL value	ObjectVariableReal

Object Setting	Selector	Setting Value	Function
Bottom Print Margin	90	REAL value	ObjectVariableReal
Right Print Margin	91	REAL value	ObjectVariableReal

Textures

Object Setting	Selector	Data Type	Function
Texturable Object	500	TRUE or FALSE (read-only)	Get ObjectVariableBoolean
Expanded Material Set	501	TRUE or FALSE[1]	ObjectVariableBoolean
Material Size	510	REAL (inches)	ObjectVariableReal
Texture Bitmap Feature Size	523	REAL value (in inches)	ObjectVariableReal
Texture Bitmap Horizontal Repeat	524	TRUE or FALSE	ObjectVariableBoolean
Texture Bitmap Vertical Repeat	525	TRUE or FALSE	ObjectVariableBoolean
Paint Width	530	LONGINT (pixels)	ObjectVariableLongint
Paint Height	531	LONGINT (pixels)	ObjectVariableLongint
Texture Space Type	540	0 (Plane) 1 (Sphere) 2 (Cylinder) 3 (Perimeter Algorithmic) 4 (Shader)	ObjectVariableInt
Texture Space Scale	543		ObjectVariableInt
Texture Space Rotation	544	REAL value (in radians)	ObjectVariableReal
Texture Space Radius	545	REAL value[2]	ObjectVariableReal
Texture Space Use Start Cap	546	TRUE or FALSE[3]	ObjectVariableBoolean
Texture Space Use End Cap	547	TRUE or FALSE[3]	ObjectVariableBoolean
Texture Space Part ID	548	INTEGER index[4]	ObjectVariableInt

Notes:

[1] *Sets whether multiple textures can be applied to object (two for roof, three for walls).*

[2] *Valid for sphere texture space only.*

[3] *Valid for extrudes and sweeps only.*

[4] *Index of multi-texturable object component.*

Gradient and Image Fills

Object Settings	Selector	Data Type	Function
Fill X Offset	70	REAL	ObjectVariableReal
Fill Y Offset	71	REAL	ObjectVariableReal
Fill I-Axis Length	72	REAL	ObjectVariableReal
Fill J-Axis Length	73	REAL	ObjectVariableReal
Fill Angle	74	REAL	ObjectVariableReal
Fill Repeat	75	TRUE or FALSE	ObjectVariableBoolean
Fill Mirror	76	TRUE or FALSE	ObjectVariableBoolean
Image Flip	77	TRUE or FALSE	ObjectVariableBoolean
Gradient Geometry Type	78	LONGINT	ObjectVariableLongint
Image Aspect Ratio	79	TRUE or FALSE	ObjectVariableBoolean

Hatches

Object Settings	Selector	Data Type	Function
Number of Levels	660	INTEGER	GetObjectVariableInt
Is Transparent	661	TRUE or FALSE	ObjectVariableBoolean
Has Page Units	662	TRUE or FALSE	ObjectVariableBoolean
Rotate In Wall	663	TRUE or FALSE	ObjectVariableBoolean
Rotate In Symbol	664	TRUE or FALSE	ObjectVariableBoolean

Misc.

Object Settings	Selector	Data Type	Function
Sweep Z Offset	400	REAL	ObjectVariableReal
Is 2D Poly Clockwise	652	TRUE or FALSE	ObjectVariableBoolean
Text Is Linked To Record	680	TRUE or FALSE	ObjectVariableBoolean
Text Repeating Tab	681	INTEGER	ObjectVariableInt
Object Fill Style	695	LONGINT	ObjectVariableLongint
Object Fill Type	696	INTEGER	GetObjectBariableInt
Object Is Locked	700	TRUE or FALSE	ObjectVariableBoolean
Format Is Visible	900	TRUE or FALSE	ObjectVariableBoolean

Menu Selectors

In this Chapter:

Menu Items and VectorScript

VectorScript provides the `DoMenuTextByName` function call to allow selection of workspace menu items directly from within a script. The appendix lists the selectors which are used with `DoMenuTextByName` to invoke a specific menu command.

Plug-in Menu Commands

VectorWorks workspaces are a mixture of internally-based menu command and external plug-in based commands. To specify an item which is a plug-in item, use the file name of the item as found in the Plug-ins folder.

Menu Chunks

Certain menu command are actually components of a workspace element known as a **menu chunk**. These elements group related items as a functional unit for ease of editing a VectorWorks workspace.

Menu chunks are called by using the name of the chunk and specifying an index value indicating the position of the desired item within the chunk. For example, to call the **Right Isometric** item (which is a part of the **Standard Views** menu chunk), use the function call:

```
DoMenuTextByName('Standard Views',8);
```

The index value 8 indicates the **Right Isometric** command, which is the eighth item in the menu chunk.

Menu Command Selectors

Menu Command	Selector
Add Surface	Add Surface
Align Objects	Align Objects
Align to Grid	Align to Grid
VectorWorks Preferences...	Application Preferences
Arc Smoothing	Arc Smoothing
Arrow Heads...	Arrow Heads
Bezier Spline Smoothing	Bezier Spline Smoothing
Classes...	Classes
Clear	Clear
Clip Surface	Clip Surface
Close	Close
Color Palette...	Color Palette
Column...	Column
Combine Into Surface	Combine Into Surface
Compose Curve	Compose Curve
Convert Copy to Lines	Convert Copy to Lines
Convert Copy to Polygons	Convert Copy to Polygons
Convert to 3D Polys	Convert to 3D Polys
Convert to Lines	Convert to Lines
Convert to Mesh	Convert to Mesh
Convert to NURBS	Convert To NURBS
Convert to Polygons	Convert to Polygons
Copy	Copy
Create Layer Link...	Create Layer Link
Create Plug-in...	Create Plug-in
Create Report...	Create Report
Create Symbol...	Create Symbol
Cubic Spline Smoothing	Cubic Spline Smoothing
Custom RenderWorks Options...	Custom RW Options Chunk
Custom Selection...	Custom Selection
Custom Tool/Attribute...	Custom Tool/Attribute
Custom Visibility...	Custom Visibility

Menu Command	Selector
Cut	Cut
Cut 2D Section	Cut 2D Section
Cut 3D Section	Cut 3D Section
Dash Styles...	Dash Styles
Decompose Curve	Decompose Curve
Deselect All	Deselect All
Document Preferences	Document Preferences
Duplicate	Duplicate
Duplicate Array...	Duplicate Array
Edit Constraints...	Edit Constraints
Export Database...	Export Database
Export DXF/DWG...	Export DXF/DWG
Export EPSF...	Export EPSF
Export Image File...	Export Image File
Export PICT...	Export PICT
Export RenderMan	Export RenderMan
Export Simple VectorScript (3D only)	Export Simple VectorScript (3D only)
Export VectorScript...	Export Text Format
Export VRML...	Export VRML Chunk
Export Worksheet...	Export Worksheet
Extrude	Extrude
Extrude...	Extrude and Edit
Fit To Window	Fit to Window
Flip Horizontal	Flip Horizontal
Flip Vertical	Flip Vertical
Floor...	Floor
Format Text...	Format Text
Hatch...	Hatch
Import DXF/DWG...	Import DXF/DWG
Import EPSF...	Import EPSF
Import Image File...	Import Image File
Import PICT...	Import PICT
Import PICT as Picture...	Import PICT as Picture

Menu Command	Selector
Import VectorScript...	Import Text Format
Import Worksheet...	Import Worksheet
Intersect Surface	Intersect Surface
Layer Scale...	Layer Scale
Layers...	Layers
Line Thickness...	Line Thickness
Link Text To Record	Link Text to Record
Lock	Lock
lower case	lower case
Move...	Move
Move 3D...	Move 3D
Move Working Plane	Move Working Plane
Multiple Extrude	Multiple Extrude
Multiple Extrude...	Multiple Extrude and Edit
New...	New
Next View	Next View
No Smoothing	No Smoothing
Normal Scale	Normal Scale
Open...	Open
Page Setup...	Page Setup
Paste	Paste
Paste As Picture	Paste As Picture
Paste In Place	Paste In Place
Patterns...	Patterns
Previous Selection	Previous Selection
Previous Views	Previous View
Print...	Print
Engineering Properties...	Properties
Quit	Quit
Redo	Redo
Revert To Saved	Revert To Saved
Roof Face...	Roof Face
Rotate...	Rotate

Menu Command	Selector
Rotate 3D...	Rotate 3D
Rotate 3D VIew...	Rotate 3D View
Rotate Left 90°	Rotate Left 90
Rotate Right 90°	Rotate Right 90
Rotate Working Plane...	Rotate Working Plane
Save	Save
Save As...	Save As
Save As Template...	Save As Template
Save View	Save View
Scale Objects...	Scale Objects
Select All	Select All
Send Backward	Send Backward
Send Forward	Send Forward
Send to Back	Send to Back
Send to Front	Send to Front
Set 3D View...	Set 3D View
Set Grid...	Set Grid
Set Layer Ambient...	Set Layer Ambient
Set Origin...	Set Origin
Set Print Area...	Set Print Area
Set Size...	Set Size
Shallow Symbol to Group	Shallow Symbol to Group
Sweep	Sweep
Sweep...	Sweep and Edit
Symbol to Group	Symbol to Group
Title Caps	Title Caps
Trace Bitmap	Trace Bitmap
Undo	Undo
Units...	Units
Unlock	Unlock
Unrotate 3D Objects	Unrotate 3D Objects
UPPER CASE	UPPER CASE

Menu Command	Selector
Wall Framer...	Wall Framer
Workgroup References...	Workgroup References

Menu Chunk Selectors

Menu Command	Selector
Active Only Gray Others Show/Snap/Modify Others	Class Options
Convert to Group	Convert to Group Chunk
Export As MiniCAD 7 File... Export As VectorWorks 8 File... Export As VectorWorks9 File...	Export Previous File Version
6 9 10 12 18 20 24 28 36 48 72 96 144	Font Size
Plain Bold Italic Underline Outline Shadow	Font Style
Group Ungroup	Group Chunk
Edit Group Exit Group Top Level	Group Navigation Chunk

Menu Command	Selector
Make Guides Select Guides Show Guides Hide Guides Delete All Guides	Guides
Join	Join Chunk
Active Only Gray Others Show Others Show/Snap Others Show/Snap/Modify Others	Layer Options
Hidden Line Dashed Line	Line Render Chunk
OpenGL Options...	OpenGL Options Chunk
OpenGL	OpenGL Render Chunk
Set Perspective... Narrow Perspective Normal Perspective Wide Perspective	Perspective Chunk
Unshaded Polygon Shaded Polygon Shaded Polygon No Lines Final Shaded Polygon	Polygon Render Chunk
2D Plan Orthogonal Perspective Oblique Cavalier 45 Oblique Cavalier 30 Oblique Cabinet 45 Oblique Cabinet 30	Projection
Fast RenderWorks Fast RenderWorks with Shadows Final Quality RenderWorks Custom RenderWorks Custom RenderWorks Options...	RenderWorks Render Chunk
Add Solids Subtract Solids... Intersect Solids	Solid Operations

Menu Command	Selector
Constraints Attributes Object Info Working Planes Resource Browser	Standard Palettes Chunk
Top/Plan Top Front Right Bottom Back Left Right Isometric Left Isometric Right Rear Iso Left Rear Iso Lower Right Iso Lower Left Iso Lower Right Rear Lower Left Rear	Standard Views
Left Center Right	Text Horizontal Alignment
Single Space 1-1/2 Space Double Space Other...	Text Spacing
Top Top Baseline Center Bottom Baseline Bottom	Text Vertical Alignment
Use Full Screen	Use Full Screen Chunk
Wireframe	Wireframe Render Chunk

Script Encryption

In this Chapter:

Encryption Overview

VectorScript provides support for protecting scripts by encryption. Encrypted scripts can then be distributed for sale or other use without making the script source code available for unintended reuse or modification. VectorScript supports encryption of plug-ins, document scripts, and standalone script files.

VectorScript encryption is non-reversible, meaning that once a script is encrypted, it cannot be decrypted for further editing or modification. Scripts should always be saved to a separate file or location prior to encryption to prevent loss of script code.

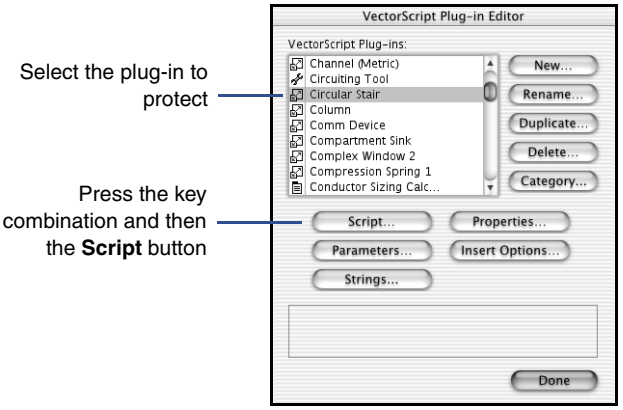
Encrypting Scripts

Plug-ins

1. Select **Organize > Scripts > Create Plug-in**. In the VectorScript Plug-in Editor, select the plug-in to be protected from the editor list.
2. Use the following key combination, pressing the keys simultaneously:

Macintosh	CAPS LOCK+SHIFT+OPTION+COMMAND
Windows	SHIFT+CTRL+ALT

3. Click on the **Script** button in the Editor. Confirm twice that the plug-in should be protected.

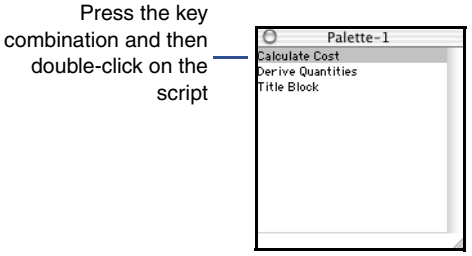


Document Scripts (Script Palette)

1. Open a script palette and select the script to be protected.
2. Use the following key combination, pressing the keys simultaneously:

Macintosh	CAPS LOCK+SHIFT+OPTION+COMMAND
Windows	SHIFT+CTRL+ALT

3. Double-click on the selected script in the palette. Confirm twice that the script should be protected.



File Scripts (Text Files)

1. Select **Organize > Scripts > Encrypt Scripts**.
2. Select the text file containing the script, and then click **Open**.
3. Enter a new name for the encrypted file, and then click **Save**.

   The file is encrypted and saved under the new name.

Include Files and Encryption

Include files used with scripts can be handled in one of two ways during the encryption process. Include files may left as unencrypted source code external to the script by appending a `.vss` extension to the file name. When the script which references the include file is encrypted, the link to the file will be preserved, and the script will use the code from the include file when executed. Scripts encrypted using this method still require the presence of the include file in order to execute correctly.

Note: *Unencrypted include files which will be used with encrypted scripts should not reference subroutines, constants, or variables contained within the script. References to these items in an encrypted script will cause the script to fail.*

Alternatively, include files can be encrypted along with the script which calls them by appending a `.px` extension to the name of the include file. In this case, the contents of the include file are copied into the calling script and then the entire body of source code is encrypted. The source code of the include file remains untouched by the encryption process. Scripts encrypted in this manner do not require the presence of external include files to execute correctly, as they contain all the needed code within the encrypted script.

For example, suppose the following procedure was in the include file `myinclude.vss` and was to be used in another script:

```
PROCEDURE Remote_Sub;
VAR
    j:INTEGER;
BEGIN
    AlrtDialog('This is the include function');
END;
```

The calling script then referenced the include file as shown:

```
PROCEDURE EncryptExample1;
VAR
    i:INTEGER;
    s:STRING;

{$INCLUDE myinclude.vss}
```

```
BEGIN
    Remote_Sub;
END;
Run(EncryptExample1);
```

If the script above were encrypted, the subroutine `Remote_Sub` would remain in the include file. It would be called as needed by the `EncryptExample1` script, and the include file would also need to be present in order for `EncryptExample1` to execute properly. If we were to change the name of the include file and modify the calling script as shown:

```
PROCEDURE EncryptExample1;
VAR
    i:INTEGER;
    s:STRING;

{$INCLUDE myinclude.px}

BEGIN
    Remote_Sub;
END;
Run(EncryptExample1);
```

In this instance the code from `myinclude.px` would be copied into the calling script at the location of the include statement, and the entire script would then be encrypted. The encrypted script would require no additional files to execute properly, and the original code in the file `myinclude.px` would be untouched.

Note: *Include files should **NOT** be encrypted as standalone documents separate from a script. Files encrypted in such a manner cannot be referenced from another script, and cannot be decrypted.*

Color Palette

In this Chapter:

- VectorWorks Standard Color Palette

VectorWorks Standard Color Palette

0	255	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	254	18	19	20	21	247	252	24	253	26	27	28	249	30	31
32	33	251	35	36	246	38	39	40	41	42	250	44	245	46	248
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	21	66	67	68	5	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
31	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
112	113	114	115	116	117	118	119	120	121	122	123	124	16	126	32
128	129	27	131	132	133	134	135	136	137	138	139	140	141	26	143
2	145	146	147	148	6	150	151	152	153	154	155	156	157	45	159
160	161	162	163	164	165	3	167	168	169	170	7	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
192	193	194	195	196	197	198	199	200	44	202	203	204	205	155	207
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
224	225	226	227	228	229	230	231	35	233	234	235	236	237	238	239
240	241	242	243	244	45	4	22	47	29	43	34	23	25	17	1

Index

Symbols

A

B

C

D

E

F

R

S

T

U

V

W