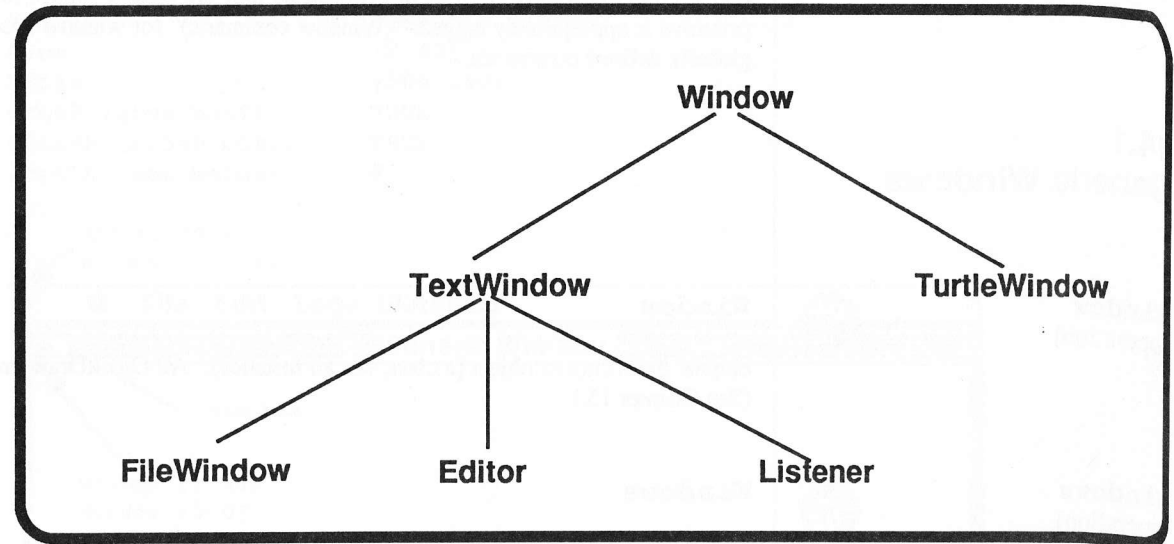


## Overview

Object Logo provides six classes of window objects:



This diagram outlines the Window object hierarchy. The lines represent the inheritance path of the given objects. All primitives defined for the Window object can also be invoked by all objects that inherit from the Window object. Thus, editors can do everything that text windows and windows can do, but cannot invoke primitives that are defined for turtle windows, file windows, or listeners (see Chapter 5).

Note: Windows themselves inherit from the OutputStream object, and the TextWindow object inherits from the InputStream object (see Chapter 12 and Appendix B).

Corresponding to each kind of window object is an Object Logo primitive that outputs that object. Because windows are objects, new kinds of windows are created with `KindOf` and new instances of windows are created with `OneOf`.

```

? Make "MyWindow OneOf Window
? Ask :MyWindow [SetWTitle "MyWindow]
? Make "XtraEditor OneOf Editor
? Ask :XtraEditor [SetWTitle "Editor2]
  
```

Most of the primitives in this chapter are defined for the various kinds of windows. However, several globally defined primitives have been included because they are closely related to windows and their primitives. Each primitive is appropriately tagged: "(Window command)" for window specific commands, or "(command)" for globally defined commands.

## 14.1 Generic Windows

**Window**  
(operation)



**Window**

outputs the Window object (a class, not an instance). All QuickDraw procedures are defined within Window. (See Chapter 15.)

**Windows**  
(operation)



**Windows**

outputs a list of all window instances in the workspace. The list is ordered from front to back; the first window in the list is always the active window.

*Example:* The procedure ActiveW outputs the active window:

```
To ActiveW
OP First Windows
End
```

**Exist**  
(Window operation)



**Exist**

initializes an instance of the Window object.

The following attributes of the window can be set using OneOf (see page 5-14)

ows. However, several globally  
ws and their primitives. Each  
ommands, or "(command)" for

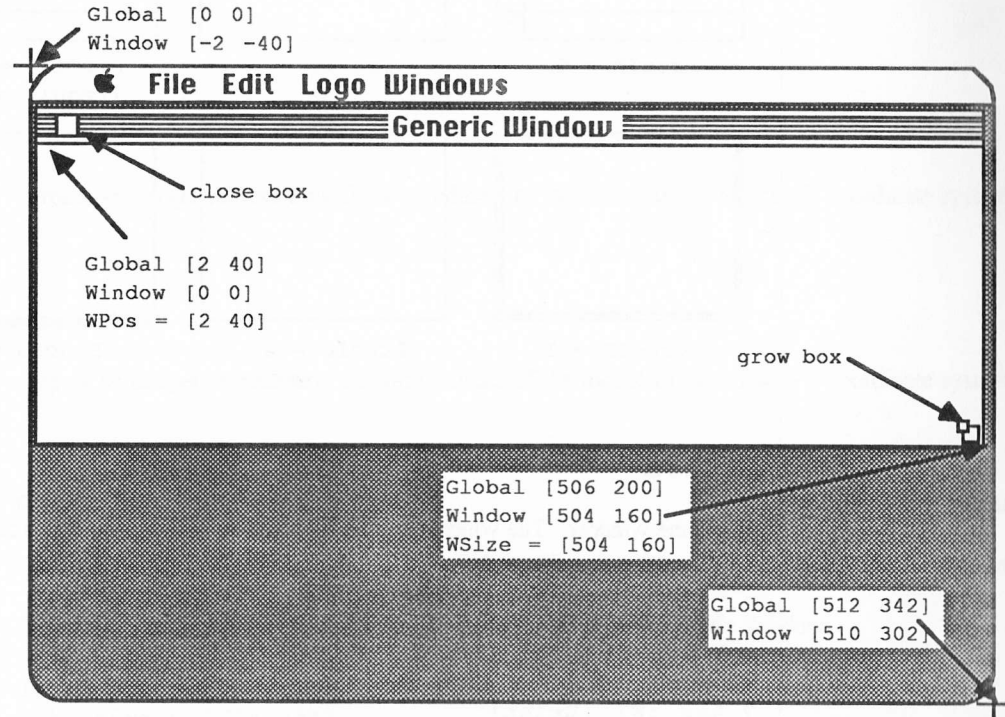
s are defined within Window.

front to back; the first window in

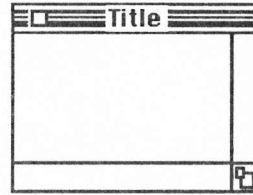
Initialization words:

Default values:

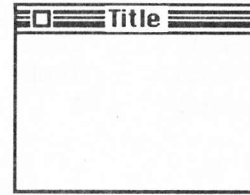
WTitle	Generic Window
WPos	[2 40]
WSize	[504 160]
GrowP (grow box?)	TRUE
CloseP (close box?)	TRUE
ProcID (see below)	8



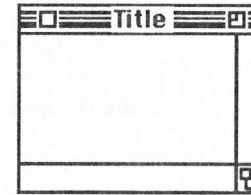
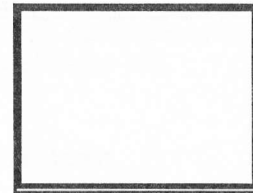
Each of the following kinds of windows can be created by providing one of the following ProcIDs to OneOf when creating a window:



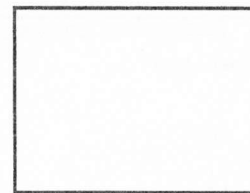
ProcID = 0



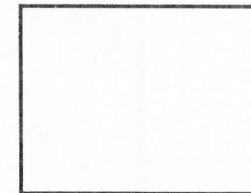
ProcID = 4

ProcID = 8  
(requires Mac+ or 512K E ROM)

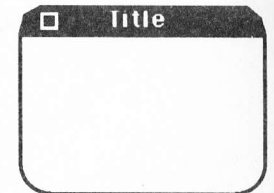
ProcID = 1



ProcID = 2



ProcID = 3



ProcID = 16

**WClose**  
(Window command)



### **WClose**

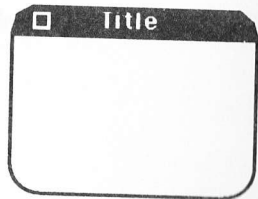
closes the window. The window can be reopened by telling it to Exist.

```
? Make "W1 OneOf Window
? Ask :W1 [WClose]
? Print Ask :W1 [WPos]
{Object WRT030 a Window} is closed.
? Ask :W1 [Exist]
? Print Ask :W1 [WPos]
2 40
```



following ProcIDs to OneOf

or 512K E ROM)



ProcID = 16

**MousePos**  
(Window operation)



**MousePos**

outputs a list of two integers representing the coordinates of the mouse in the window's coordinate system.

*Example:* PrMousePos continuously prints the mouse's window position.

```
To PrMousePos :Window
Show Ask :Window [MousePos]
PrMousePos :Window
End
```

**MouseX**  
(Window operation)



**MouseX**

outputs an integer representing the x-coordinate of the mouse in the window's coordinate system.

**MouseY**  
(Window operation)



**MouseY**

outputs an integer representing the y-coordinate of the mouse in the window's coordinate system.

**WSelect**  
(Window command)



**WSelect**

causes the window to become the active window. The window automatically comes to the front.

*Example:* The procedure SetActiveW makes its input be the active window:

```
To SetActiveW :Window
Ask :Window [WSelect]
End
```

**SetWSize**  
(Window command)



### **SetWSize List**

sets the width and height respectively of the window to the elements of *List*. Unlike the corresponding command in turtle windows, this will not affect the drawing in the window nor the coordinate system of the window. Note also that generic windows will not redraw the portions of their drawings that have been covered up as turtle windows and text windows do.

**WSize**  
(Window operation)



### **WSize**

outputs the size of the window, as a list of two integers representing the window's width and height in pixels. Note that the window's size does not include its title bar.

*Example:* `WidthOf` and `HeightOf` output the width and height of their input, which is presumed to be a window.

```
To WidthOf :Window
OP Ask :Window [First WSize]
End
```

```
To HeightOf :Window
OP Ask :Window [Last WSize]
End
```

**SetWPos**  
(Window command)



### **SetWPos Point**

moves the window so that the upper-left corner (point [0 0]) of the window is *Point* in the global (screen) coordinate system (see diagram above).

**WPos**  
(Window operation)



### **WPos**

outputs the position of the upper-left corner of the window (point [0 0] in the window), in the form of a two-element list containing its global x and y coordinates.

*Example:* `Corners` outputs a rectangle in global coordinates equivalent to the given window.

the corresponding command system of the window. Note when covered up as turtle

width and height in pixels.

which is presumed to be a

on the global (screen)

), in the form of a two-

n window.

**SetWTitle**  
(Window command)



**WTitle**  
(Window operation)



## Drawing Text

```
To Corners :Window
Ask :Window [OP (Se WPos
                (Sum First WPos First WSize)
                (Sum Last WPos Last WSize))]
—
—
End
```

```
? Show Corners Oneof Window
[2 40 504 160]
```

**SetWTitle** *Word*

sets the title of the window to *Word*.

**WTitle**

outputs the title of the window as a word.

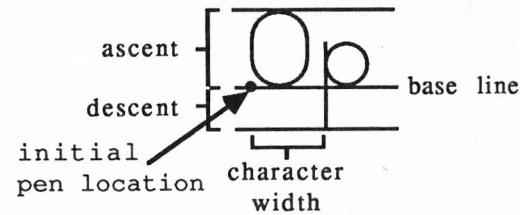
When characters are drawn in a window, they are drawn at the window's pen location, in the window's font, font size, font style, and text mode. See Chapter 15, **QuickDraw**. `WriteAscii` (see below) is the primitive that actually draws characters when a window is asked to `Show`, `Print`, `Type`, or `FType`, all of which are inherited from the `OutputStream` object by windows. The primitives in this section apply to generic windows in which text is mixed with graphics. See Section 14.2, **Text Windows**, to see how text can be printed as characters rather than drawn as dots.

**GetFontInfo**  
(Window operation)



### GetFontInfo

outputs a list of four numbers representing the window's font information, all measured in pixels: ascent, descent, maximum width, and leading (the distance in pixels between the base line on one line of text and the base line on the next line of text). Text is drawn with the beginning of the base line at the pen location.



**SetFont**  
(Window command)



### SetFont *Symbol*

sets the window's font to the font named *Symbol*. Note: Only the names of fonts in your System File can be used as inputs.

```
? Ask :MyWindow [SetFont "chicago]
? Ask :MyWindow [SetFont "|new york|]
```

**WFont**  
(Window operation)



### WFont

outputs the name of the window's font.

**SetFontSize**  
(Window command)



### SetFontSize *Number*

sets the window's font size to *Number*. Note: If the font size is not defined for the window's font, the font will be scaled to the requested size.

**WFontSize**  
(Window operation)



**WFontSize**

outputs the size of the window's font.

**SetFontStyle**  
(Window command)



**SetFontStyle** *Symbol(List)*

sets the window's font style to *Symbol(List)*, which must be one of, or a list of, the following: PLAIN, BOLD, ITALIC, UNDERLINE, OUTLINE, SHADOW, CONDENSE, EXTEND.

? **SetFontStyle** "Underline

? **SetFontStyle** [Bold Underline]

**WFontStyle**  
(Window operation)



**WFontStyle**

outputs a list representing the style of the window's font.

**SetTextMode**  
(Window command)



**SetTextMode** *Symbol*

sets the window's text mode to *Symbol*, which must be one of the following: PAINT, DOWN, REVERSE, ERASE (see page 11-11). When a window is created its text mode is DOWN.

**TextMode**  
(Window operation)



**TextMode**

outputs the window's text mode (see above).

**WriteASCII**  
(Window command)



**WriteASCII** *Number*

draws the character whose ASCII value is *Number* at the window's pen location using the window's font, font style, font size and text mode. The pen is moved to just after the character on the font's baseline (see Appendix A for ASCII values).

**TextWidth**  
(operation)



**TextWidth *Word***

outputs the width of *Word*, in pixels, using the window's font, style and size.

## 14.2 TextWindows

`TextWindow` is a built-in class that inherits from both `Window` and `InputStream`. This means that `TextWindow` and all of its descendants are capable of doing the same things as `Window` (which inherits from `OutputStream`) and `InputStream`. These inherited capabilities are documented elsewhere (Sections 12.1, 12.2 and 14.1) This section describes the primitives that are specific to `TextWindow` and its descendants.

All text windows may contain text that is automatically redrawn and can be scrolled, if the window has scroll bars. All text windows can be edited by using the keyboard and the mouse. There are also a few simple cursor positioning commands that can be used during editing (see Appendix C). Although text windows respond to graphics commands since they inherit from `Window`, text redrawing usually overwrites graphics.

**TextWindow**  
(operation)



**TextWindow**

outputs the `TextWindow` class. Generic text windows (created by `Oneof TextWindow`) are not normally used. But listeners, editors, and file windows, which are specializations of `TextWindow`, are used. `TextWindow` is useful if you want to define your own specializations. Note that text windows can contain a maximum of 32000 characters.

**TextWindows**  
(operation)



**TextWindows**

outputs a list of all text windows ordered from front to back. The list includes listeners, editors, file windows and any other initialized specializations of `TextWindow`.

**Exist**  
(`TextWindow` command)



**Exist**

initializes an instance of the `TextWindow` object.

The following attributes of a text window can be set when initialized by `OneOf` (see page 5-14). Since text windows inherit from `Window`, the window attributes listed on page 14-3 can also be set.

<u>Initialization words:</u>	<u>Default values:</u>
<code>HScrollP</code>	<code>TRUE</code>
<code>VScrollP</code>	<code>TRUE</code>
<code>Text</code>	(the empty word)

`HscrollP` determines whether or not a newly created text window instance has a horizontal scroll bar, `VScrollP` determines whether or not it has a vertical scroll bar, and `Text` is the initial text in the window and may be any Object Logo value. The value is `Printed` into the window before it is displayed.

The following instruction creates a new text window without any scroll bars, with the specified size and text.

```
? Make "GreetWindow (Oneof TextWindow "HScrollP "FALSE
_ "VScrollP "FALSE "WSize [100 20] "Text [Hello There!])
```

**SaveText**  
(TextWindow command)



**SaveText** *FileName*

saves the text of the text window in the file named by *FileName*. If the file already exists, it is overwritten without warning.

**WSave**  
(TextWindow command)



**WSave**

saves the text of the text window in a file that is specified by a `PickPutFile` dialog. Also changes the window title to be the name of the file name (without any pathname information). Text windows have to prompt the user for a file name every time `WSave` is invoked since they have no associated file. `WSave` is equivalent to the following procedure:

```
Ask TextWindow [To NewWSave]
LocalMake "File PickPutFile "Title
SaveText :File
SetWTitle JustFileName "File
End
```

NewWSave uses the procedure JustFileName to remove any pathname information from the file name. JustFileName could be defined as follows:

```
To JustFileName :FileName [:JustName " ]
If Empty :FileName [OP :JustName]
If EqualP Last :FileName ": [OP :JustName]
OP (JustFileName ButLast :FileName FPut Last :FileName :JustName)
End
```

**HardCopy**  
(TextWindow command)



### HardCopy

prints the text of the text window on the currently selected printer according to the current page setup.

? Ask :my.window [HardCopy] ; prints the text of :my.window on the printer

**WClose**  
(TextWindow command)



### WClose

closes the text window and frees up the memory used by the window and its text.

**ClearText**  
(global and  
TextWindow command)



### ClearText CT

clears all the text from the text window. The global version of ClearText Asks the top listener to ClearText.

## Character Locations

The location of a character in a text window can be specified by either a **character position** or by a **character offset**.

A **character position** is a list of two numbers. The first number is the column of a character, or the number of characters preceding it in a line. The second number is the row or line number of a character. Both of these



tion from the file name.

JustName)

current page setup.

on the printer

the top listener to

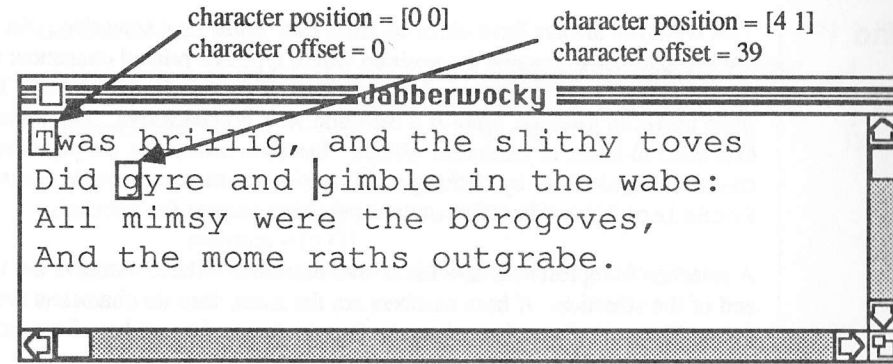
position or by a character


a character, or the number of a character. Both of these

numbers start at zero, so the tenth character on the third line would have character position [9 2]. Fixed-width fonts such as Monaco and Courier are recommended if text is to be printed in columns.


A **character offset** is the number of preceding characters in a text window. Thus, the first character in a window has character offset of zero, the next has character offset of one, and so forth. Note that each line ends with a Return character that is included in the character count.

The primitives CharPos and CharOffset are used to convert between these two different representations of a character's location.



**CharPos**  
(TextWindow operation) 

**CharPos *N***  
outputs the character position of the character whose offset is *N* in the text window.

**CharOffset**  
(TextWindow operation) 

**CharOffset *Position***  
outputs the character offset of the character at *Position* in the text window.

**TextSize**  
(TextWindow operation)



**TextLength**  
(TextWindow operation)



### **Selections, Editing and the Clipboard**

#### **TextSize**

outputs a list that represents the number of columns and rows of characters of the current font, size, and style that could be displayed in the text window at its current size.

#### **TextLength**

outputs the number of characters currently in the text window.

Text windows always have either an **insertion point** or a **selection**. An insertion point is the blinking vertical bar that represents the position where typed or printed characters will appear. A selection is a range of characters that is highlighted in inverse video (if the window is active). These are actually different aspects of the same thing; an insertion point is a selection with no characters. The insertion point and the selection are expressed in terms of **insertion offsets**. Insertion offsets are the positions *between* characters. The selection can be changed either by clicking and dragging the mouse or under program control by using the primitive `SetSelection`. The `Selection` primitive outputs the selection.

A selection is represented as a list of two numbers. These numbers are the insertion offsets of the beginning and end of the selection. If both numbers are the same, then no characters are selected and the selection is an insertion point. The second number minus the first is always the number of characters selected.

current font, size, and style that

point is the blinking  
r. A selection is a range of  
actually different aspects of the  
and the selection are  
characters. The selection  
l by using the primitive

n offsets of the beginning and  
and the selection is an insertion  
ed.

character offsets	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
	T	w	a		s	b	r	i	l	l	i	g	,	a	n	d		
insertion offsets	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

insertion point at insertion offset 3  
selection = [3 3]

character offsets	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
	T	w	a	s	b	r	i	l	l	i	g	,	a	n	d			
insertion offsets	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

selection between insertion offsets 5 and 11  
selection = [5 11]  
selected characters are at character offsets 5 to 10

The selection is implicitly used in most operations on the text of text windows. The editing primitives Cut, Copy, Paste, and Clear all operate on the selection. Typing or printing replaces the selection with the new characters (this is the same as insertion if no characters are selected). Find and FindReverse set the selection if a target is found, and ScrollTo scrolls the window so that the selected text is visible.

The clipboard is also implicitly used in some editing operations. The clipboard is a temporary storage area for text (or graphics). Copy and Cut copy the selection to the clipboard, and Paste replaces the selection with the contents of the clipboard. The contents of the clipboard can be accessed by using the Clipboard primitive and changed by using the SetClipboard primitive.

**SetSelection**  
(TextWindow command)



**SetSelection List**  
**SetSelection N**

if the input is a number, SetSelection sets both the beginning and end of the selection to N. If the input is a

list of two numbers, `SetSelection` sets the beginning and end of the selection to be the insertion offsets of the two numbers in *List*. If any of these numbers is larger than the text length, then the text length is used. Note that even though `SetSelection` accepts a list, it is a list of two insertion offsets, not a character position.

**Example:** The procedure `SetPosSelection` can be used to select the text between two characters, specified by position, including both characters. Notice that the second number in the list that is made as an input to `SetSelection` is one plus the character offset of the second character. This is the insertion offset after the character. The procedure `SelectAll` can be used to select all of the text in a window. It takes advantage of the fact that if a number larger than the length of the text is used as an input to `SetSelection`, then the text length is used.

```
? To SetPosSelection :Pos1 :Pos2
> SetSelection List CharOffset :Pos1 1 + CharOffset :Pos2
> End
SETPOSSELECTION defined.
? To SelectAll
> SetSelection [0 32000]
> End
SELECTALL defined.
```

**Selection**  
(TextWindow operation)



**Selection**

outputs a list of two numbers representing the insertion offsets of the beginning and end of the selection. If no text is selected, then both numbers are the same.

**SetClipboard**  
(command)



**SetClipboard** *Word*

sets the contents of the clipboard to *Word*.

**Clipboard**  
(operation)



**Clipboard**

outputs (in the form of a word) the contents of the clipboard.

**Cut**  
(TextWindow command)



**Cut**

copies the current selection to the clipboard and then deletes it from the window.

**Copy**  
(TextWindow command)



**Copy**

copies the current selection to the clipboard.

**Paste**  
(TextWindow command)



**Paste**

replaces the current selection with the contents of the clipboard.

**Clear**  
(TextWindow command)



**Clear**

deletes the current selection without affecting the clipboard.

**InsertFile**  
(TextWindow command)



**InsertFile** *FileName*

replaces the current selection with the text from the file named by *FileName*. If the file does not exist, an error is signalled. *FileName* may either be a full or partial pathname (see the **Overview** to Chapter 13).

**Find**  
(TextWindow command)



**Find Word**

searches forward through the text window for *Word* starting at the next character after the beginning of the selection. If *Word* is found, it is selected and scrolled to (if the window has scroll bars), otherwise the selection remains unchanged. The search is case-insensitive. (Uppercase characters are considered equivalent to lowercase characters.) The `FoundP` primitive (see below) can be used to determine if the *Word* was found.

**FindReverse**  
(TextWindow command)



### FindReverse Word

searches backwards through the text window for *Word* starting at the character just before the beginning of the selection. If *Word* is found, it is selected and scrolled to (if the window has scroll bars), otherwise the selection remains unchanged. The search is case-insensitive. (Uppercase characters are considered equivalent to lowercase characters.) The FoundP primitive (see below) can be used to determine if the *Word* was found.

**FoundP**  
(TextWindow operation)



### FoundP

outputs TRUE if the most recent Find or FindReverse command found its input in the text, FALSE otherwise. It has no effect on the selection or scroll position. If FoundP is invoked before any invocations of Find or FindReverse, it will output FALSE.

The following procedure can be used to replace all occurrences of a word in a text window with another.

```
Ask TextWindow [To ChangeAll :From :To]
SetSelection 0
DoUntil [Find :From If FoundP [Type :To]] [Not FoundP]
End
```

**ScrollTo**  
(TextWindow command)



### ScrollTo

scrolls the window (if the window has scroll bars) just enough so that the selected text is within the window. If the selection is too big to fit in the window, the top of the selection is scrolled to the first line of the window.

**SetScrollPos**  
(TextWindow command)



**SetScrollPos** *N*  
**SetScrollPos** *Position*

scrolls a text window so that the *N*th line (starting at zero) is the first visible line in the window (does not affect the horizontal position) or scrolls the window so that the character at *Position* is the upper left-hand character in the window. A window can only scroll horizontally if it has horizontal scroll bars and can only scroll vertically if it has vertical scroll bars.

st before the beginning of the  
l bars), otherwise the selection  
nsidered equivalent to lowercase  
Word was found.

`ScrollPos`  
(TextWindow operation)



put in the text, FALSE  
ked before any invocations of

xt window with another.

l text is within the window. If  
o the first line of the window,

in the window (does not affect  
he upper left-hand character in  
s and can only scroll vertically if

`TextWindow`  
Input/Output

The following instruction scrolls a text window to the beginning.

```
? Ask :my.text.window [SetScrollPos [0 0]]
```

### ScrollPos

outputs the character position of the character that is visible in the upper left-hand corner of the window.

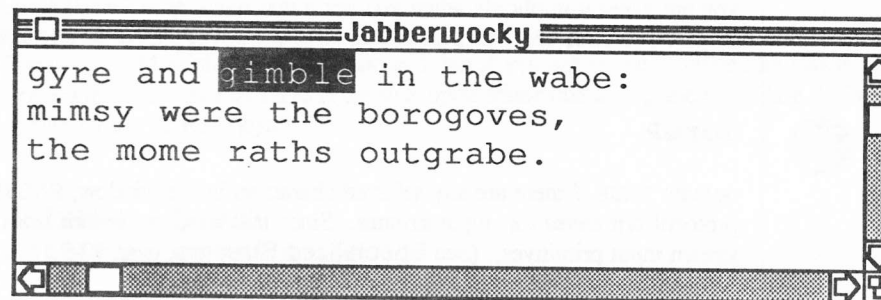
The following procedure can be used to scroll a text window down one window full.

```
? Ask TextWindow [To PageDown]
> SetScrollPos Last ScrollPos + Last TextSize
> End
PAGEDOWN defined.
```

If the window on page 14-13 were given these instructions:

```
? Ask :Jabberwocky [SetSelection [48 54] SetScrollPos [4 1]]
```

it would look like this:



Text windows can behave as both input streams and output streams (see chapter 12). Thus text windows can perform all of the printing and reading primitives (`Print`, `Type`, `Show`, `FType`, `ReadList`, `ReadChar`, and `ReadWord`). All I/O for text windows is affected by the selection. Characters output to text windows replace the selection (if any) and input characters come from the selection.



**WriteASCII**  
(TextWindow command)



**WriteASCII N**

replaces the selected text with the character whose ASCII value is *N*. `WriteASCII` is the required protocol procedure for output streams. Ordinarily, you do not invoke `WriteASCII` explicitly, but you are using it implicitly when you ask a text window to `Print`, `Type`, etc. Since text windows inherit from `OutputStream` (via `Window`), they can perform all stream output primitives. (See **Specialized Streams**, page 12-9.)

**ReadASCII**  
(TextWindow operation)



**ReadASCII**

outputs the ASCII value of the first character of the text window's selection. It increments the selection start so that the character will no longer be included in the selection. If there is no selection, an error is signalled. (`ReadASCII` should only be invoked if `MoreP` outputs `TRUE`). Ordinarily, you do not invoke `ReadASCII` explicitly, but you are using it implicitly when you ask a text window to `ReadList`, `ReadChar`, etc. Since text windows inherit from `InputStream`, they can perform all stream input primitives. (See **Specialized Streams**, page 12-9.)

**UnReadASCII**  
(TextWindow command)



**UnReadASCII N**

decrements the start of the selection (ignoring *N*). Ordinarily, you do not invoke `UnreadASCII` explicitly, but you are using it implicitly when you ask a text window to `ReadList`. Since text windows inherit from `InputStream`, they can perform all stream input primitives. (See **Specialized Streams**, page 12-9.)

**MoreP**  
(TextWindow operation)



**MoreP**

outputs `TRUE` if there are any selected characters in the window, `FALSE` otherwise. `MoreP` is one of the required protocol primitives for input streams. Since text windows inherit from `InputStream`, they can perform all stream input primitives. (See **Specialized Streams**, page 12-9.)



## 14.3 Editors

Editors are specializations of `TextWindow`, so they can do everything that text windows can do (see the previous section). Editors differ from text windows in a few ways. They are normally created by issuing a command, such as `Edit`, `EditNames`, `EDPS`, etc. This brings up a new editor containing the text of the procedures, variables or property lists specified by the command. Editors remember the names of the procedures, variables or property lists they contain. This information can be accessed by using the primitives `EditorProcs`, `EditorNames`, and `EditorPLists`. Editors also ask if you want to run their contents when you close them, and `COMMAND-R` runs the contents (if there is no selection), closes the window and selects the listener in one operation.

Editors are meant to perform the duties of the traditional Logo editor. The main differences are: (1) There can be any number of editors. (2) You don't have to explicitly leave the editor before you can do anything else; other things can be done and Editors can be returned to by simply clicking the mouse. (3) The mouse can be used for editing, and (4) `COMMAND-R` and `COMMAND-PERIOD` are used to complete or abort editing instead of `Control-C` and `Control-G`.

Editors are provided so that users familiar with other versions of Logo will feel at home. But we encourage users to use **File-Based Programming** for large projects (see page iii-32).

**Edit**  
(command)



**Edit *Symbol***  
**Edit *SymbolList***

creates an editor containing the definition of the procedure *Symbol* or the definitions of the procedures in *SymbolList*. Multiple editors are allowed, but if you ask to edit a procedure that is already in an editor, Object Logo will warn you that there is another open editor containing the procedure definition. The procedures must be owned by the current object.

**EditName**  
(command)



**EditName *Symbol***  
**EditName *SymbolList***

creates an editor containing the definition of the variable *Symbol* or the definitions of the variables in *SymbolList*. Multiple editors are allowed, but if you ask to edit a variable that is already in an editor, Object Logo will warn you that there is another open editor containing that variable. The variables must be owned by the current object.

**EditPList**  
(command)



**EditPList** *Symbol*  
**EditPList** *SymbolList*

creates an editor containing the definition of the property list *Symbol* or the definitions of the property lists named by *SymbolList*. Multiple editors are allowed, but if you ask to edit a property list that is already in an editor, Object Logo will warn you that there is another open editor containing that property list.

**EDPS**  
(command)



**EDPS**

creates an editor containing the definitions of all the procedures owned by the current object. Multiple editors are allowed, but if an owned procedure is already in an editor, Object Logo will warn you that there is another open editor containing the same procedure definition.

**EDNS**  
(command)



**EDNS**

creates an editor containing the definitions of all the variables owned by the current object. Multiple editors are allowed, but if an owned variable is already in an editor, Object Logo will warn you that there is another open editor containing the same variable definition.

**EDPLS**  
(command)



**EDPLS**

creates an editor containing the definitions of all property lists. Multiple editors are allowed, but if there are any other Editors containing property lists, Object Logo will warn you.

**Editor**  
(operation)



**Editor**

outputs the *Editor* class.

**Editors**  
(operation)



**Editors**

outputs a list of *Editor* instances in top to bottom order of the windows.

**Exist**  
(Editor command)



**Exist**

initializes an instance of the Editor object.

The following attributes of an Editor can be set when initialized by `OneOf` (see page 8-14). Because Editors inherit from windows, the window attributes listed on page 14-3 can also be set.

Initialization words:

Default values:

Procs

[] (a *SymbolList* may be supplied)

Names

[] (a *SymbolList* may be supplied)

PLists

[] (a *SymbolList* may be supplied)

**WClose**  
(Editor command)



**WClose**

closes the Editor and displays a dialog box asking if you want to run the contents of the Editor.

**EditorProcs**  
(Editor operation)



**EditorProcs**

outputs a list of the names of the procedures that the Editor was opened with. If extensive editing has occurred, this may bear little relation to what is currently in the Editor.

*Example:* The procedure `SmartEdit` searches Editors to see if there is already an editor containing the definition of `:ProcName`. If it finds one, that procedure definition is shown; if not, it opens a new editor.

```
To SmartEdit :ProcName [:EditorList Editors]
If EmptyP :EditorList [Edit :ProcName]
IfElse MemberP :ProcName Ask First :EditorList [EditorProcs]
_ [Ask First :EditorList
_ [WSelect SetSelection 0 Find Word "|To | :ProcName]]
_ [SmartEdit :ProcName BF :EditorList]
End
```

**EditorNames**  
(Editor operation)



### **EditorNames**

outputs a list of the names of the variables that the editor was opened with. If extensive editing has occurred, this may bear little relation to what is currently in the editor.

**EditorPLists**  
(Editor operation)



### **EditorPLists**

outputs a list of the names of the property lists that the editor was opened with. If extensive editing has occurred, this may bear little relation to what is currently in the editor.

## **14.4 FileWindows**

File windows are a specialization of text windows, so they can do everything that text windows can do (see Section 14.2). There are a few differences between file windows and text windows. A file window is always associated with a file. Its title is the same name as the file name without any pathname information. The full pathname of the file can be discovered by using the primitive `EditingFile`. When Asked to `WSave`, a file window saves its text into its file. A file window keeps track of whether or not its contents have changed since the last `WSave`. This can be determined by using the primitive `ChangedP`. When closed, if its contents have changed, a file window puts up a dialog box that asks if you want to save changes. File windows also know how to restore their original contents, or the contents since the last `WSave` (by using the primitive `WRevert`).

File windows are normally created by using the **New File** or **Edit File...** menu items, or by using the `EditFile` primitive. When a file window is created, if there is a file window or file stream already associated with the same file, the file window will be opened read-only and a dialog box will warn you. Any attempt to edit a read-only file window will generate a warning and the editing operation will not be completed. You can find out if a file window is read-only by using the primitive `ReadOnlyP`.

**EditFile**  
(command)



### **EditFile** *FileName*

creates a file window containing the text of the file named *FileName*. The window's title will be *FileName* without any volume or pathname information included. If you try to edit a file that is already being edited, you will get a warning and the new file window will be read-only.

**FileWindow**  
(operation)



**FileWindow**

outputs the FileWindow class.

**FileWindows**  
(operation)



**FileWindows**

outputs a list of all existing file windows ordered from front to back.

**Exist**  
(FileWindow command)



**Exist**

initializes an instance of the FileWindow object.

The file name of a file window can be set when initialized by OneOf (see page 5-14). Because file windows inherit from windows, the window attributes listed on page 14-3 can also be set.

Initialization words:

Default values:

FileName

Untitled

**WSave**  
(FileWindow command)



**WSave**

saves the text of the file window into its file. It does this without user intervention, unlike WSave in FileWindow's parent, TextWindow. WSave is equivalent to the following instruction:

? Ask :My.File.Window [SaveText EditingFile]

**SaveText**  
(FileWindow command)



**SaveText** *FileName*

saves the text of the text window in the file named by *FileName*. This is the same as TextWindow's SaveText procedure except that it also changes the file window's associated file.

**WClose**  
(FileWindow command)



**EditingFile**  
(FileWindow operation)



**WRevert**  
(FileWindow command)



**ChangedP**  
(FileWindow operation)



**ReadOnlyP**  
(FileWindow operation)



## 14.5 Listeners

### **WClose**

if any changes have been made in the text of the window since the last `WSave`, puts up a dialog box asking if you want to save the changes; then closes the window.

### **EditingFile**

outputs the full pathname of the file that the file window is editing.

### **WRevert**

throws away any changes that have been made, and reloads the file window with the current contents of its file on the disk.

### **ChangedP**

outputs `TRUE` if the file window's text has been changed since it was last saved, `FALSE` otherwise. If a file window has just been opened, it outputs `FALSE`.

### **ReadOnlyP**

outputs `TRUE` if the window is read-only, `FALSE` otherwise. The text of a read-only file window cannot be changed. File windows are only read-only if there was another file window, or a file stream, associated with the same file when the file window was created.

Object Logo Listeners are a specialization of text windows, so they can do everything that text windows can do (see Section 14.2). Listeners also have many additional capabilities. Listeners are the windows that are normally used to interact with Object Logo. There is usually only one Listener (although there can be many), and it is the place where commands like `Print` normally print. Listeners use a special font called `Schizo` so that user-typed characters appear **bold** and computer-typed characters appear plain.



When Listeners fill up (at 32000 characters), they automatically delete the first ten thousand characters.

The text in a Listener has two parts: a history of the user's interaction with Object Logo, and characters that have been typed by the user but not yet read by Object Logo. The dividing line between the session history and the unread characters is called a Listener's **read position**. This is the place where Object Logo will look for the next character to be read from a Listener. The read position is also the place where characters appear that are printed to a Listener. The read position is maintained so that Logo's output doesn't interfere with a user's typing.

The unread portion of a Listener is called the **input area**. The input area can be edited in the usual Macintosh style by using the mouse and the command keys (see Appendix C). Typing Return or Enter while the selection is in the input area signals Logo to go ahead and read the input.

There are also four keys that have special meanings while typing in the input area: Tab, Command-Return, Shift-Return, and Command-X (Cut). Tab creates a line continuation, Command-Return ignores the current input and prints a new prompt, Shift-Return types a Return without signalling Logo to read the input, and Command-X does the usual Cut if there is a selection, otherwise it Cuts the entire input area. For example:

```
? show primitives           ; Command-Return typed
? print "Please             ; Shift-Return typed
print "eluci               ; Tab typed
_date                      ; Return typed
Please
elucidate
?
```

The **history area** of a Listener is treated as a source of input and cannot be edited (except by using Cut and Paste). Attempts to type in the history are interpreted as a desire to re-use some of the information contained in the history as input, and characters are copied down to the end of the input area. The rules for copying down the history characters when typing occurs in the history area are:

- If there are selected characters and Return is typed, copy the selected characters to the input area. If the selection contains both bold and plain characters, copy only the bold characters. If the selection contains only plain characters, copy them down then make them bold.
- If there are no selected characters and Return is typed, copy down the bold characters in the line where the insertion point is.
- If characters other than Return are typed, copy down the bold characters surrounding the selection, then continue with the editing operation.
- Command-R in the history area is equivalent to typing two Returns (copy down, then do it).

**Listener**  
(operation)



**Listener**  
outputs the `Listener` class.

**Listeners**  
(operation)



**Listeners**  
outputs a list of current `Listener` instances ordered from top to bottom as the windows appear on the screen.

**Exist**  
(`Listener` command)



**Exist**  
initializes an instance of the `Listener` object.

Because listeners inherit from windows, the window attributes listed on page 14-3 can be set.

**WSave**  
(`Listener` command)



**WSave**  
This is the same as `TextWindow`'s `WSave` except that it also puts up a dialog box that asks if you want to convert the text in the window to normal characters. If you do not convert the text, the Schizo font must be used to read it.

**HardCopy**  
(`Listener` command)



**HardCopy**  
This is the same as `TextWindow`'s `HardCopy` except that it uses normal character codes for the bold characters, but prints them using font style **Bold**.

**WClose**  
(`Listener` command)



**WClose**  
closes the `Listener` if it is not the only `Listener`, or signals an error if it is the only `Listener`. There must always be a `Listener` open so that unspecified output has a place to go.



**ClearText**  
(global and TextWindow  
command)



**ClearText**  
**CT**

clears the text from the text window. When invoked in the Logo object it clears the top listener. This is not actually a Listener command; it is listed here since a global ClearText clears the top listener window.

**WriteASCII**  
(Listener command)



**WriteASCII** *N*

prints the character whose ASCII value is *N* at the current read-position in the listener. WriteASCII is the required protocol procedure for output streams. Ordinarily, you do not invoke WriteASCII explicitly, but you are using it implicitly when you ask a Listener to Print, Type, etc. Since Listeners inherit from OutputStream (via Window) they can perform all stream output primitives. (See **Specialized Streams**, page 12-9.)

**ReadASCII**  
(Listener command)



**ReadASCII**

outputs the ASCII value of the next unread character from the listener. It also deletes the character from the text of the Listener. ReadASCII is one of the required protocol primitives for input streams. Since listeners inherit from InputStream (indirectly through TextWindow) they can perform all stream input primitives. (See **Specialized Streams**, page 12-9.)

**UnReadASCII**  
(Listener command)



**UnReadASCII** *N*

puts the character whose ASCII value is *N* back into the unread input area of the listener. UnreadASCII is one of the required protocol primitives for input streams. Since listeners inherit from InputStream (indirectly through TextWindow) they can perform all stream input primitives. (See **Specialized Streams**, page 12-9.)

**MoreP**  
(Listener operation)



**MoreP**

always outputs TRUE. There are always more potential characters coming from the user. This is a required protocol procedure for input streams. It is used during a ReadList operation from the listener. (See **Specialized Streams**, page 12-9.)

**KeyP**  
(Listener operation)



**KeyP**

outputs TRUE if there are any pending characters in the Listener, FALSE otherwise.

**Cut**  
(Listener command)



**Cut**

copies the current selection to the clipboard and then deletes it from the window. The characters are converted to normal character codes (they lose Schizo boldface information).

**Copy**  
(Listener command)



**Copy**

copies the current selection to the clipboard. The characters are converted to normal character codes (they lose Schizo boldface information).

**Paste**  
(Listener command)



**Paste**

replaces the current selection with the contents of the clipboard. All the pasted characters will appear boldface.

**ClearInput**  
(Listener command)



**ClearInput**

deletes any pending user input.

## Programming Examples

The `TypeOver` procedure types its input as `Type` does, except that `TypeOver` types over existing characters in the text window. It replaces the characters rather than pushing them over (it actually deletes the same number of characters that it types.) `TypeOver` is useful if you need to change text that is already formatted, for example to change a number in a column of numbers.

```
Ask TextWindow [To TypeOver :Input]
LocalMake "SizeBefore TextLength
Type :Input
LocalMake "SizeDiff TextLength - :SizeBefore
LocalMake "SelStart First Selection
SetSelection List :SelStart :SelStart + :SizeDiff
Clear
End
```

`ClearToEOL` deletes all the characters from the current selection to the end of the line.

```
Ask TextWindow [To ClearToEOL]
LocalMake "SelStart First Selection
Find Char 13 ;Return Character
LocalMake "SelEnd IfElse FoundP [First Selection] [TextLength]
SetSelection List :SelStart :SelEnd
Clear
End
```