

APL64

Windows Reference Manual

Copyright

While every attempt has been made to ensure that the information in this document is accurate and complete, some typographical errors or technical inaccuracies may exist. APLNow, LLC does not accept responsibility for any kind of loss resulting from the use of information contained in this document.

This page shows the publication date. The information contained in this document is subject to change without notice. Any improvements or changes to either the product or the document will be documented in subsequent editions.

This software/documentation contains proprietary information of APLNow LLC. All rights are reserved. Reverse engineering of this software is prohibited. No part of this software/documentation may be copied, photocopied, reproduced, stored in a retrieval system, transmitted in any form or by any means, or translated into another language without the prior written consent of APLNow LLC.

U.S. Government Restricted Rights. The software and accompanying materials are provided with Restricted Rights. Use, duplication for disclosure by the Government is subject to the restrictions in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, or subparagraphs (c) (1) and (2) of the Commercial Computer Software - Restricted Rights at 48CFR52.227-19, as applicable. The Contractor is APLNow LLC, 2640 Route 70 Bldg. 4, Manasquan, NJ 08736.

APL64

This edition published 2025.

Copyright © 2005–2025 APLNow LLC.

Portions copyright © Microsoft Corporation, One Microsoft Way, Redmond, Washington 98052-6399 USA. All rights reserved.

APL2000 and APL+Win are registered trademarks of APLNow LLC in the United States and/or other countries. APL★PLUS is a registered trademark of Manugistics, Inc. Microsoft, Windows, Windows 98, Windows 2000, Windows NT, Windows XP, Windows Vista, Windows 7, Windows 8, Windows 10, Windows 11 and Excel are trademarks or registered trademarks of Microsoft Corporation. All other brand names and products are trademarks or registered trademarks of their respective companies.

Table of Contents

Table of Contents	iii
Chapter 1: Windows Interface: Classes	17
Object Class Reference.....	17
Button	18
Check.....	21
Combo	23
CommandBar	26
CommandButton.....	35
DateTime	38
Edit	47
Form	49
Frame	53
Imagelist	55
Label	60
List	62
Listview	64
MDIForm.....	74
Media	76
Menu.....	82
Option	87
Page	89
Picture.....	91
Printer	94
Progress	99
RichEdit	101
Scroll	108
Selector	110
Spinner	114
Status.....	117
Timer.....	120
Toolbox	121
Trackbar	124
Tree	127
#	138
Chapter 2: Windows Interface: Properties.....	141
activeform	142
apldata	142
autocomplete	143
bars (read-only)	143
barwrap.....	144
bin	144
bins (read-only).....	144
bitmap	145
bitmapsize (read-only).....	145
border	145
buddy	147

canpaste (read-only)	147
canredo (read-only)	147
canundo	148
caption	148
captiondock	149
captionfloat	149
channelwhere (read-only)	149
children (read-only)	150
class (read-only)	150
classes (read-only)	151
clienthWnd (read-only)	151
clientwhere	151
clsid	152
collate	152
color	152
colordepth	155
columnDisplay	156
contact	157
content	157
copies	158
count (read-only)	158
data	158
crashList	159
crashLogEvent	159
crashLogMode	160
crashLogPath	160
ddeApplication	161
ddelItem	161
ddeMode	162
ddeTimeout	162
ddeTopic	162
debugLogEvent	163
debugLogPath	163
def	164
default	165
defaultFont	165
deferexit	168
description (read-only)	168
dirty	169
dock	169
dockable	169
dockbreak	170
dockheader	170
dockideal	170
docklength	170
dockmargin (read-only)	171
dockmax	171
dockmin	171
dockmindepth	172
dockside	172
dockvert (read-only)	172
dpi (read-only)	172
dragimage	173
dragSource (read-only)	173

dropcolor	174
dropfont	174
droptarget (read-only)	174
edge	174
empty (read-only)	175
enabled	175
error	176
errorcode (read-only)	176
errorLogEvent	176
errorLogFilter	176
errorLogPath	177
errormessage (read-only)	177
eventLogName	177
events (read-only)	178
extent	178
file	179
firstday	179
fixedtabsize	180
floatlayout	180
floatsize	180
floatwhere	181
focus	181
font	181
format	183
hdc	184
help	185
helpcontext	185
highlightbold	185
highlightcut	185
highlightdrop	186
highlightfocus	186
highlightselect	186
himage (read-only)	186
hwnd (read-only)	187
hwndddropdown (read-only)	187
hwndmain (deprecated)	187
hwndsplash	187
icon	188
image	188
imagealloc	189
imagecount (read-only)	189
imageindex	189
imagelist	190
imagelistdisabled	190
imagelisthot	191
imagelistlarge	191
imagelistuser	191
imagenames	192
imagescale	192
imagesize	193
imagespace	194
increment	194
indent	195
infotipmask	195

instance	195
interface	196
interval.....	196
keys	196
labeledithwnd (read-only).....	197
limit	197
limitwhere	198
links	198
list	199
margin	202
maskcolor.....	202
mediahwnd (read-only)	202
mediatype.....	203
methods (read-only).....	203
minsize (read-only)	203
missing	204
mode	204
modified.....	206
modifystop.....	207
monthdelta	207
msversion (read-only)	207
name	208
newclasses.....	208
noredraw	208
normsize (read-only)	209
null	209
obj	210
opaque	210
opened (read-only).....	211
order	211
orientation	212
origin	212
overlays.....	212
padding	213
page (read-only).....	213
pages (read-only).....	213
palette	213
paper.....	214
papers (read-only).....	214
pending	214
place.....	215
pointer	215
position.....	216
poslargeimage.....	217
possmallimage	217
printername	218
progid	218
prompt.....	218
properties (read-only).....	218
range	219
recursions.....	220
redoname (read-only)	220
restartArg	221
restartCmd	221

restartDir	221
restartMsg	222
restartRate	222
roworigin (read-only)	222
rtf	223
savecontent	223
scale	223
scrollaccel	225
scrollmargin	225
searchstring (read-only)	225
selalign	226
selborder	226
selbullet	226
seleffects	227
seleffectsall (read-only)	227
selcolor	228
selection	228
self (read-only)	229
selfont	229
selfontall (read-only)	230
selindents	230
sellinespacing	231
selnumstart	231
selnumstyle	231
selnumtab	232
selfpargall (read-only)	232
selftf	233
selshading	233
selfspaceafter	233
selfspacebefore	233
selstyle	233
selstyleall (read-only)	234
selftabs	234
selftext	234
separator	235
server (read-only)	235
sessions (read-only)	235
shortcut	236
showtext	237
siblings (read-only)	237
size	238
sliderlen	238
sliderwhere (read-only)	238
sortorder	239
sourceformats	239
state (read-only)	239
status	240
style	241
suppress	250
tabgroup	251
tabparent	251
tabrows (read-only)	251
tabstop	252
targetformats	252

text	253
theme	253
themeactive (read-only)	253
themed (read-only)	253
thememode	254
tickinterval	254
tickpos (read-only)	255
ticks	255
today	255
tooltip	256
tooltipenabled	256
tooltipstyle	257
tooltiptime	257
tooltipwidth	258
translate	258
undolimit	260
undoname (read-only)	260
unicodebstr	260
units (read-only)	261
value	261
version (read-only)	264
viewalign	264
viewmode	264
visible	264
where	265
width	266
workarea (read-only)	267
wrap	267
xclasses	268
zoom	268
Δudp (user-defined data properties)	268
ActiveX Array Properties	269
 Chapter 3: Windows Interface: Event Handlers	271
Introduction	271
onAction	273
onActivate	274
onBeginLabelEdit	274
onChange	275
onClick	276
onClose	277
onCloseUp	278
onColClick	278
onColResize	278
onComAction	279
onContextMenu	279
onCrypto	279
onDblClick	279
onDdeConnect	280
onDdeDisconnect	280
onDdeExecute	280
onDelete	281
onDestroy	281
onDockChange	282

onDragContinue	282
onDragEnd	283
onDragEnter	284
onDragFeedback	284
onDragLeave	285
onDragOver	285
onDragStart	286
onDrop	287
onDropDown	288
onEditEnd	289
onEditNL	289
onEditLoad	289
onEditSaved	290
onEditStart	290
onEndLabelEdit	291
onError	291
onExit	292
onExitError	292
onExpanded	293
onExpanding	293
onFile	293
onFocus	294
onFormatQuery	294
onFormatString	294
onGetDayState	295
onGetInfoTip	295
onGetVirtualData	296
onHelp	297
onHide	299
onHotChange	299
onKeyDown	299
onKeyPress	301
onKeyUp	302
onLimit	302
onMemory	303
onModified	303
onMouse	303
onMouseDouble	304
onMouseDown	305
onMouseDrag	306
onMouseEnter	306
onMouseLeave	307
onMouseMove	307
onMouseUp	308
onMove	308
onNew	308
onOpen	309
onPage	310
onPaint	310
onPrompt	311
onProtected	311
onReopen	312
onReqResize	312
onResize	312

onResource	313
onScroll	313
onSelChange	314
onSelect	314
onSend	314
onShow	315
onSort	315
onSpin	315
onStart	316
onStop	316
onSystemChange	316
onTimer	317
onTooltip	317
onUnfocus	318
onUpdate	318
onUserKeyDown	319
onValidate	319
onWait	320
 Chapter 4: Windows Interface: Methods	321
Introduction	321
Abort	322
AddImages	322
AddRows	322
Arrange	322
AutoFit	323
BeginLabelEdit	324
CancelLabelEdit	324
CharToLine	324
Click	325
Close	325
CloseDoc	326
Command	326
ContextHelpMode	327
Copy	327
Create	327
Cut	328
DdeExecute	328
DdePoke	329
DdeRequest	329
DecryptFile	329
Defer	330
Delete	331
DeleteNodes	332
DeleteRows	332
DevMode	332
DevNames	333
Draw	333
Eject	349
EncryptFile	350
EncryptLog	352
EnsureVisible	353
dEnumEnd	353
EnumNext	353

EnumStart.....	354
Event.....	354
Exec.....	355
Expand	355
FindNode	356
Focus.....	356
FormatMessage.....	357
Get	357
GetCells	358
GetImages	358
GetInfo	359
GetPos	360
GetRows	360
GetSession	360
Help	360
Hide	362
HitTest	363
HwndToName.....	363
Info	364
InsertNodes	365
InsertRows.....	365
LineToChar.....	366
Modify	366
New.....	366
Notify.....	368
Open	368
OpenDoc	369
PageSetup.....	369
Paint.....	370
PaneWhere.....	370
Paste.....	370
Play.....	371
Pop	371
Popup	372
Print	374
PrintDlg	375
Printers	376
Push.....	377
Put..	378
Redo	378
Ref..	378
ReleaseObjects	379
ReqResize	379
Reset	379
Resize.....	380
ScrollCaret.....	380
Seek.....	380
Send	380
Set..	381
SetCells	382
SetImages	382
SetInfo	382
SetLinks.....	383
SetPos	383

SetRows	384
SetStatus	384
Setup	384
Show	385
ShowNode	385
SortChildren	385
Stepit	386
Stop	386
Undo	386
UseObject	387
Validate	387
VT	387
VV	389
Wait	390
XInfo	391
 Chapter 5: Programming the Windows Interface: Introduction	393
Overview	393
Understanding the Fundamental Elements	394
Naming Objects	395
Windows Interface and System Variables Summary	396
Building a Graphical User Interface in APL	397
Using Controls to Get User Input	398
Understanding the Basics	400
Creating Objects	400
Using Precompiled Object Definitions	401
Using Relative Object Naming	402
Taking the First Steps	405
A Simple Form to Customize Widgets	405
Purposes of the Controls in the APL+Win Interface	408
DPI-Awareness	411
 Chapter 6: Programming the Windows Interface: Basic	415
Working with Objects	415
System Handling of Windows Objects	415
Determining the Contents of an Object	417
Positioning and Sizing Controls	417
Positioning and Sizing Forms	418
Fundamental Aspects of a User Interface	419
Menus and Shortcuts	419
Alternatives to Menus	422
Moving through the Interface	422
Accommodating Keyboard and Mouse Input	424
Keyboard Handling	424
Mouse Handling	426
Using the Wait Method on Forms	427
Building an APL+Win Windows Interface Program	428
How Callbacks Work	428
Common Dialogs and Message Boxes	430
Using Printer Objects	431
Extending the Widget Customizer	433
Revising the Main Form	433

Chapter 7: Programming the Windows Interface: Intermediate	439
Naming with Links	439
Parsing Rules for Using Links	440
Drag-and-Drop of Objects	441
Using Graphics with the Draw Method	443
Draw Operations	445
Pens and Brushes	446
Scaling in Graphics	447
Modes of Drawing	448
Bitmaps and Icons	449
Freeing Bitmap, Icon, Pen, and Brush Handles	450
Using Dynamic Data Exchange	451
Understanding DDE Terminology	451
Using APL for DDE Communication	452
Simple Examples	455
More Complex Examples	457
DDE Workspaces	462
APL Windows Utilities Summary	462
Windows Demonstration Workspaces	462
The UCMDSW User Command File	463
The WINAPI Workspace	463
The WINDOWS Workspace	464
Unicode Character Support	465
Advanced-to-Expert Programming Techniques	465
Chapter 8: Using ActiveX in APL64	467
Creating an ActiveX Control	467
Properties of an ActiveControl	468
ActiveControl Persistence	470
ActiveControl Methods and Events	470
Creating an ActiveX Server Object	471
ActiveObject Properties, Methods, and Events	471
Using ActiveObjects and ActiveControls	472
APL Facilities that Apply to ActiveX Objects	472
Error Properties on ActiveObjects and ActiveObjects	473
ActiveX Typed Values	473
Creating a Hierarchy of Objects and the Redirection Syntax	475
An Example Using the Excel Spreadsheet	475
Special APL Syntax Rules for ActiveX Objects	477
Specifying the Difference in Setting and Referencing for ActiveX Objects	477
Enumerating Collections in ActiveX Objects	478
Chaining Actions and Specifying Parameters	480
Passing ActiveX Objects as Arguments to Other ActiveX Methods	482
Chapter 9: Creating User-Defined Classes	485
Creating a User-Defined Class	485
Creating an Event Handler for onNew	485
Creating an Event Handler for onAction	487
Creating Events for the User-Defined Class	488
Programming Style for User-Defined Classes	491
Modifying a Built-in Class	491
Specifying the Class to be Modified	491
Using the Event Handler for onNew	491
Modifying Built-in Classes without using onNew	492

Controlling Default versus Special Behavior	493
Caution: Use with Care	494
Remember Your Escape Hatch	494
Chapter 10: Using the APL Registry Object.....	495
Introduction	495
Getting Started	496
Properties Reference	500
Methods Reference	506
Event Reference	511
Chapter 11: Calling Dynamic Link Libraries	512
Using the Name Association Function	513
Creating and Using Associated Functions.....	513
Using the Windows Call Function	519
Using <code>▢wcall</code>	519
Startup Files for <code>▢wcall</code> (.ADF and .INI).....	521
Argument Conversion for the Windows Call Function	533
Filters and Callbacks	535
Related Workspaces.....	541
Chapter 12: Zip and Unzip in APL+WIN	542
Background.....	542
Zip Class in APL+Win	542
Zip Code Examples	542
The <code>lftocr</code> Function	548
Reference Section	548
<code>▢WI</code> Zip Class	548
Zip Methods	549
Create	549
Delete	549
Zip	550
Unzip.....	550
Zip Event Handler Properties.....	552
<code>onComment</code>	552
<code>onDisplay</code>	552
<code>onListing</code>	553
<code>onPassword</code>	553
<code>onProgress</code>	553
<code>onReplace</code>	554
<code>onSound</code>	554
Zip Class Properties.....	555
<code>comment</code>	555
<code>level</code>	555
<code>omitdir</code>	555
<code>onlynewer</code>	555
<code>overwrite</code>	556
<code>privilege</code>	556
<code>recreatedirs</code>	556
<code>recursedir</code>	557
<code>totals</code>	557
<code>uzoptions</code>	557
<code>verbose</code>	558
Info-ZIP copyright and license	559

Appendix WA: Virtual Key Codes 560
 Virtual Key Codes..... 560

Index 437

Chapter 1: Windows Interface: Classes

This chapter, along with the chapters on Properties, Event Handlers, and Methods, describes the fundamental elements of the APL64 Windows user interface. It is written for the programmer creating a graphical user interface or interacting with other applications or objects through Windows. The use of "you" refers to the programmer and to the actions of the program you write to control the use of the Windows interface (WI). The person who generates actions in an interface you construct is a "user."

- **Object**

An object in the APL64 WI context has a different meaning than an object that refers to the APL language in general. In the APL64 WI context, an object is any of the fundamental units you use in building the interface. A form is an object. A control that you place on the form -- for example, a push button -- is also an object. If you use a combo box, which is a drop-down list with an associated entry field, the entire combo box is an object. An individual object is an instance of the class to which it belongs.

- **Class**

A class is a category of objects, all of which have certain common characteristics. If you have a pair of radio buttons on a form that allow the user to choose either high resolution or low resolution, each of those buttons (with its associated caption that describes the choice) is an instance of the Option class.

- **Property**

Properties of an object are the characteristics or attributes you can set that determine how the object looks, where it is located, and how it reacts. A push button, for example, has a caption that labels it, a style that determines whether it is the default button for its form, and a location on the form (where), among other characteristics. All of these are its properties.

There are also a set of special properties, called event-handler properties, that specify actions for the system to take when certain events occur, such as the user clicking the mouse on the object. Program actions can also trigger event-handler properties; for example, when something in the system changes the text of an Edit object, that change can trigger an `onChange` event handler. The `onChange` event-handler property defines the action for the system to take when the text changes. Informally, this is known as a Change event.

You can set most properties for an object, which means you assign a value to that characteristic, but you do not have to set all the properties for an object. Some properties are optional; some have defaults. Properties can change indirectly, as the result of a user or program action. The properties that an individual object possesses depend on the class to which it belongs. Some properties are read-only; they can be referenced but not set.

Object Class Reference

APL64 uses the object classes that this chapter describes. The description for each class mentions the general uses for its objects, lists the properties and event-handler properties associated with the class, describes the key properties for the class, and lists the associated methods. Your program can reference all the properties of the objects when your application runs; you can also set or reset most properties.

The descriptions indicate which classes can contain (be a parent of) this class and which classes can be contained by (be children of) this class. The descriptions also include a standard prefix, which you can use, but are not required to use, as part of a systematic naming style. The classes are listed alphabetically. They include the following:

Button	Form	Media	RichEdit	Trackbar
Check	Frame	Menu	Scroll	Tree
Combo	Imagelist	Option	Selector	# (the system object)
CommandBar	Label	Page	Spinner	
CommandButton	List	Picture	Status	ActiveControl (see note)
DateTime	Listview	Printer	Timer	ActiveObject (see note)
Edit	MDIForm	Progress	Toolbox	

Note: ActiveControl and ActiveObject are two designations that each represent many classes of ActiveX objects, which the APL system treats in a fashion similar to the built-in classes. See the Using ActiveX in APL64 chapter in this manual for information on including instances of these classes in your interface.

Button



Prefix: bn

Summary:

A rectangle with a three-dimensional appearance. The user selects a button to take an action.

Uses:

You use a button when the form design calls for a command or procedure to be executed immediately. When the user places the cursor over the button and presses the primary mouse button, the appearance of the button on the form changes so it looks pushed in. The user can slide the cursor off a pushed-in button without selecting it and without generating a Click event.

You might provide a button on a form for the user to indicate that input is complete. When the user clicks the button, or presses the space bar when the button has the focus, the program begins to process the data the user just entered. To provide this effect, the system executes the expression defined by the onClick event-handler property.

Regular Properties:

border, caption, children, class, color, data, def, deferexit, edge, enabled, events, extent, font, help, helpcontext, hwnd, imageindex, imagelist, imagescale, imagespace, instance, keys, links, methods, mode, modified, modifystop, name, noredraw, opened, order, place, pointer, prompt, properties, scale, scrollaccel, scrollmargin, self, size, state, style, suppress, tabgroup, tabparent, tabstop, targetformats, theme, tooltip, tooltipenabled, tooltipstyle, tooltiptime, tooltipwidth, translate, value, visible, where, and $\Delta ud p$.

caption

The caption text for this object appears on the button itself. The initial value of caption is the name you assign the object when you create it. Once you create the object, however, the name property and caption property are independent. If you set caption or change name, the value of the other does not change. You can place multiple rows of text in the caption of a button by specifying the caption property as a vector that includes `\n` or `\t` characters where you want the lines to break. If a caption contains an ampersand (&), the letter following it is an access key that the user can press with the Alt key to click the button.

style

The style property for a Button object is an integer scalar or vector; a scalar can be the sum of selected

values. The initial value of `style` for any newly created control object is 0. You can set `style` to contain either or both of the primary values for ordinary buttons.

The larger values of `style` are for enhanced buttons that allow you to place an image on the button as well as a text caption, align and justify the image, place multiple rows of text in the caption, justify the text, create a two-state toggle button, and prevent the button from receiving the focus. This last feature allows you to have a Button that acts like one cell of a toolbar; it can receive mouse input without having the input focus. See the notes below the table for additional information.

Values for the Button Class `style` Property

Value	Description
Primary Value: Choose only one value	
0	Draw button in style of operating system. Automatically justify text, and align image to left of text.
1	Default Button for entire form.
2	Cancel button (closes form if no <code>onClick</code> handler exists for this button).
Add-on Values: Choose none, any or all of these values	
4	Button does not get the focus. (Cannot be used with 1 or 2)
8	Button toggles between normal and selected. (Cannot be used with 1 or 2)
16	Do not grey image when button is disabled.
32	Use foreground color of color property for text; ignore background color.
65536	Wrap caption. See Note 6.
Drawing Style: Choose one of these values	
64	Force Windows 3.1-style buttons. (Obsolete)
128	Force Windows 95 style button. (Obsolete)
Justification Values: Choose one of these values	
256	Left justify text and image.
512	Right justify text and image.
768	Center justify text and image.
Alignment Values: Choose one of these values	
4096	Align image to right of text.
8192	Align image above text.
12288	Align image below text.

Note 1 (Default): The default button for a form is indicated by a bold frame around the button when the user first sees the form. If the user tabs to another button, the bold frame moves to the button with the focus. If the user tabs to a control other than a button, the bold frame returns to the default button. Usually, when a user presses Enter, that action is equivalent to clicking the button with the bold frame. (One exception is when the focus is in a text entry field that accepts the Enter character, that is, one with 8192 in its `style` property.) Only one button on a form should have style 1.

Note 2 (Cancel): If the `style` property value for a button includes 2 in its sum, the button becomes a cancel button, which by convention has “Cancel” as its caption. If the user presses Esc while in a form, the system, in effect, clicks the cancel button if one exists. Cancel buttons have distinctive behavior in Windows. If there is no `onClick` event handler for a cancel button, the default behavior triggers the `onClose` event-handler property for the form (and, in turn, the form closes unless the event handler prevents that default behavior). The cancel button is intended to allow the user to back out of a form without exercising any options. The button's `onClick` event handler or the form's `onClose` event handler interprets and implements this. Only one button on a form should have this style.

You can define your own event-handler properties for this button style. If you do so, the system does not close the form when a user clicks this button. The action the system takes depends entirely on your handler.

Note 3 (Justification): The default is to allow the system to automatically justify the image and text. If you have only one or the other specified, automatic justification centers the text or the image. If you have both, the system left justifies if you aligned the image to the left, right justifies if you align the image to the right, and centers if you align the image above or below the text.

Note 4 (No focus): You can add the value 4 into the style property for a button to specify that it cannot receive the focus. You cannot combine this with either style 1, Default button, or style 2, Cancel button. Setting this property also implicitly sets the tabstop property to zero. Note that if you set style 4 and then change to another style that does not include 4, the system does not reset the tabstop property state. You must do that yourself if you want the user to be able to tab to the button again.

Note 5 (Toggle button): You can add the value 8 into the style property for a button to specify that it toggles between the normal and the selected state; you cannot use it in conjunction with either style 1 or style 2. Style 8 is almost always used in combination with style 4.

Note 6 (Wrap caption): You can add the value 65536 to allow the caption to wrap when the width of the caption is wider than the width of the button.

imagelist

The imagelist property is a character vector containing the name of the Imagelist object from which to draw the image to place on the face of the button. You specify the index to the image you want to display in the imageindex property. This is analogous to placing images on the pages of a selector.

imageindex

The origin one index of the image you want to appear on the button from the imagelist named in the imagelist property of that Button object. If you do not define the imagelist property, specifying imageindex has no meaning or effect. If you do have an imagelist defined for a button, specifying zero means no image.

Warning: Although imageindex is now defined as an integer singleton, additional elements may be added in a future release; your application should be prepared to gracefully accept a result of more than one element if you reference this property.

imagespace

The value of this property defines the number of pixels (not scaling units) between the image and the caption. The default value is 3 pixels. This property is useful only when you set both an image and a text caption for the button.

theme

The theme property of a Button class controls the visual styling theme used for drawing the control. The default value is an empty character vector (' '), which uses the new visual styling for the control. To suppress the new visual styling for the control, set thememode to "~".

value

The value property of a Button object indicates whether the button is down (1) or up (0) when queried. When the button's value is set, the button changes state to down (1), up (0), or down/up flash (-1). When the button's value is set to -1, it causes a Click event.

Event-handler Properties:

onAction, onClick, onClose, onDelete, onDestroy, onDragEnter, onDragLeave, onDragover, onDrop, onExit, onExitError, onFocus, onHelp, onHide, onKeyDown, onKeyUp, onModified, onMouseDouble,

onMouseDown, onMouseDrag, onMouseEnter, onMouseLeave, onMouseMove, onMouseUp, onMove, onOpen, onPaint, onReopen, onResize, onSend, onShow, and onUnfocus.

Methods:

Click, Close, Create, Defer, Delete, Draw, Event, Exec, Focus, Help, Hide, Modify, New, Open, Paint, Popup, Ref, Resize, Send, Set, SetLinks, and Show.

Parentage:

Buttons are most often children of a Form, Frame, Page, or CommandBar. They may also be children of some control objects: Button, Check, Combo, Edit, Label, List, Option, Picture, Scroll, Spinner, Status, and Toolbox.

Buttons can usefully contain a Timer, Imagelist, or invisible Menu. The system allows most classes as children of a Button; it specifically prohibits Form, MDIForm, CommandBar, CommandButton, Page, and Printer.

Examples:

These example statements create buttons labeled OK and Cancel on the existing form `fmForm`; they also set the style of Cancel so it closes the form when clicked and specify an `onClick` event-handler, named `Go`, for the OK button.

```
'fmForm.bnOK' owi 'New' 'Button'
'fmForm.bnOK' owi 'caption' 'OK'
'fmForm.bnOK' owi 'where' 5 15 1.5 10
'fmForm.bnOK' owi 'onClick' 'Go'
'fmForm.bnCancel'owi 'New' 'Button' ('caption' 'Cancel')
  ('where' 8 15 1.5 10) ('style' 2)
```

Check



Prefix: ck

Summary:

A square box with associated text, usually to the right of the square. When users click on the box or its text, or press the space bar while the object has the focus, they toggle a mark to appear or disappear, giving a visual indication of an on/off setting for the condition the text specifies. The same action triggers the `onClick` event-handler.

You can also define a three-state check box, which can be unchecked, checked, or indeterminate (greyed out, which usually indicates the choice does not apply given the circumstances).

Uses:

Check controls can be single items or they can exist as part of a group. Each check box can be selected independently of the others; you can select any number of check boxes in a group. The Option control is similar; however, options exist only in groups, and if the user selects any one option, it precludes the other options in the same group.

Regular Properties:

border, caption, children, class, color, data, def, deferexit, edge, enabled, events, extent, font, help, helpcontext, hwnd, imagescale, instance, keys, links, methods, mode, modified, modifystop, name, noredraw, opened, order, place, pointer, prompt, properties, scale, scrollaccel, scrollmargin, self, size, state, style,

suppress, tabgroup, tabparent, tabstop, targetformats, theme, tooltip, tooltipenabled, tooltipstyle, tooltiptime, tooltipwidth, translate, value, visible, where, and $\Delta u d p$.

caption

The text beside a Check control. If a caption contains an ampersand (&), the letter following it is underlined and designates an access key that the user can press with the Alt key to check or uncheck the control.

theme

The theme property of a Check class controls the visual styling theme used for drawing the control. The default value is an empty character vector (' '), which uses the new visual styling for the control. To suppress the new visual styling for the control, set thememode to "~".

value

To set or reference whether a Check control is checked (1) or unchecked (0). For a three-state check box, the value 2 specifies the indeterminate state. Each time the value property is set to any valid value, it triggers the onClick event-handler.

order

The sequential position of this control among all the controls (other than menu) that are children of the object that contains it. order values are a sequence of positive integers starting with 1. The order property defines where the focus moves when a user presses the Tab key.

For objects that are grouped, the tab moves to the first object in the group and the arrow keys move among the members of the group. Pressing the Tab key when the focus is on any member of the group places the focus on the next ungrouped object or the first member of the next group, whichever immediately follows the order value of the last member of this group.

As you use the New or Create method to create control objects on a form, the system assigns each control the next available order number. If the last object you created was a Check object (or an Option), this new Check object is in the same group. This means you must create all instances of a Check group sequentially or you must assign the order numbers so they group properly.

Typically, there is a nongrouped object that separates a group, such as a Label above or a Frame that surrounds the group. If you do not have a natural separator, you can create a hidden Label to separate distinct groups of Check objects, or you can use the tabgroup property. See the descriptions of tabstop, tabgroup, and tabparent in the Windows Interface: Properties chapter for more information.

style

The style property for a Check object is an integer vector or scalar. The initial value of style for any newly created control object is 0. The table below shows the possible values.

Values for the Check Class style Property

Value	Description
Primary Value: Choose only one value	
0	Check box on the left of text
1	Check box on the far right; text left-justified
Add-on Value: Choose with either primary value	
2	Three-state check box; unchecked, checked, or indeterminate

Event-handler Properties:

onAction, onClick, onClose, onDelete, onDestroy, onDragEnter, onDragLeave, onDragover, onDrop, onExit, onExitError, onFocus, onHelp, onHide, onKeyDown, onKeyUp, onModified, onMouseDouble, onMouseDown, onMouseDrag, onMouseEnter, onMouseLeave, onMouseMove, onMouseUp, onMove, onOpen, onPaint, onReopen, onResize, onSend, onShow, and onUnfocus.

Methods:

Close, Create, Defer, Delete, Draw, Event, Exec, Focus, Help, Hide, Modify, New, Open, Paint, Popup, Ref, Resize, Send, Set, SetLinks, and Show.

Parentage:

Check boxes are most often children of a Form, Frame, Page, or CommandBar. They may also be children of some control objects: Button, Check, Combo, Edit, Label, List, Option, Picture, Scroll, Spinner, Status, or Toolbox. The system allows most classes as children; it specifically prohibits Form, MDIForm, CommandBar, CommandButton, Page, and Printer.

Example:

This example shows how you can create two Check objects with access keys tagged by the ampersand.

```
w ← 1 1 1.5 14
'fmMain.chkBold' □wi 'New' 'Check' ('where' w) ('caption' '&Bold')
'fmMain.chkItalic' □wi 'New' 'Check' ('where' (w+3 0 0 0))
↳ ('caption' '&Italic')
```

Combo

Prefix: cb

Summary:

A combination text box and list where the user can either type a value in the edit box or select a value from the associated list.

Uses:

Typically, the edit box is one line high and wide enough to accommodate the longest choice. The list can be of variable length; it can be displayed routinely, or the user can pull down the list like a menu. You can change the choices dynamically by setting the list property. If there are more choices than can be displayed in the space allowed, the Combo lets a user scroll vertically through the choices. When the focus is on a Combo box and a user presses keys, the characters appears in the edit box. If the list is longer than the system can display, the system also moves the first entry in the list that begins with the typed characters to the top of the list display area. If no item in the list begins with the typed characters, the system displays the top of the list.

Regular Properties:

autocomplete, border, children, class, color, data, def, deferexit, edge, enabled, events, extent, font, help, helpcontext, hwnd, imagelist, imagescale, instance, keys, limit, links, list, methods, mode, modified, modifystop, name, noredraw, opened, order, place, pointer, prompt, properties, range, scale, scrollaccel, scrollmargin, selection, self, size, state, style, suppress, tabgroup, tabparent, tabstop, targetformats, text, theme, tooltip, tooltipenabled, tooltipstyle, tooltiptime, tooltipwidth, translate, value, visible, where, and $\Delta u d p$.

style

An integer vector that contains selected values from the following list or an integer scalar that is the sum of the selected values. The initial value of `style` for any newly created control object is 0. The table below shows the possible values; use one of the primary values plus any of the add-on values.

Values for the Combo Class style Property

Value	Description
Primary Value: Choose only one value	
0	Drop-down list.
1	Simple (always down).
2	Drop-down list with display field.
Add-on Values: Choose none, any, or all of the values	
8	Inhibit automatic scrolling in the edit field.
16	Vertical scroll bar in list.
32	Allow the display of a vertical fraction of a line.
128	Hide scroll bar if not necessary (short list).
256	Update text in edit window before running <code>onClick</code> .
2048	APL keyboard entry in edit portion of combo.
4096	Extended Combo control (images and indents). See note.

Note: The extended Combo control has a different underlying structure; it allows you to add images to the list items and to indent list items one or more steps. The list property is allowed to be a five-column matrix, with the last four columns being meaningful only for this style. In addition, this style Combo control acts as though you also included 256 in the style property; the text in the edit window is updated before the `onClick` handler runs.

Caution: Microsoft warns that the extended Combo style may be incompatible with other style values. In particular, the simple style where the list is always down may have severe Paint problems when used in conjunction with the extended Combo control.

autocomplete

If you set this property to a non-default value, as the user types, the Windows system will pop up a small window like a tooltip with choices that match the letters typed so far. If the user presses the Enter key, the entire string is inserted into the edit box as the text.

The default value is zero. Non-zero values allow you to specify various sources from which the candidate strings are taken. As with a style property, you can specify multiple sources by setting a value that is the sum of the individual values.

The values for this property define the following items as sources for the suggested strings:

- 1 The file system and the rest of the shell
- 2 URLs in the User's History
- 4 URLs in the User's Recently Used List (this includes entries in Start/Run, for example)
- 16 The file system only
- 32 File system directories, UNC servers, and UNC server shares (but not files).

The values 1, 16, and 32 are variations on the same set, so they should not be used together. Thus, any odd value 17 or greater is improper, as is any value greater than 46.

list

For all styles, a vector of enclosed character vectors, a character vector with `\t`-delimited strings, or a

character matrix of choices in the list. When you reference it, it is returned as you specified it. For the extended Combo control, this property may also be a five-column matrix with the first column holding the character text that is displayed in the list. The remaining columns hold single integers: the second column is the number of indent steps (each one 10 pixels); the third column is the origin 1 index of the image in the imagelist specified in the imagelist property that you want to display when the item is not selected (if zero, no image); the fourth column is the index in the same imagelist for the image you want to display when the item is the one selected (you may specify -1, the default, to have the same image as unselected); and the fifth column is the index into the overlays property of the same imagelist for an overlay image.

You may also specify this extended list property as a nested vector of vectors, with some or all of the vectors having the same structure as the matrix row. If you do not specify style 4096, the extra columns are ignored.

text

A character vector that specifies the text that appears in the edit window of the ComboBox. Setting this property for a drop-down (style 0) or simple (style 1) Combo object causes the value property to be reset to 0 (that is, it nullifies the current selection). When you set the text of a drop-down Combo, this triggers the onChange event-handler, and the system highlights a matching item in the list. However, the value property is zero; the user must still drop down the list and let it close to cause the value to change. If you want to have a default value, you can set the value property, which will change the text in the edit box and the selection to match.

If you attempt to set the text in a style 2 Combo, you are successful only if the text matches an item in the list. In that case, the text and the selection change and the onClick event handler fires.

theme

The theme property of a Combo class controls the visual styling theme used for drawing the control. The default value is an empty character vector (' '), which uses the new visual styling for the control. To suppress the new visual styling for the control, set thememode to "~".

value

An integer scalar that indicates the origin 1 index of the item selected. If value is zero, no item is selected. Setting the value property automatically sets the text property equal to the corresponding item in the list. Setting this property also triggers the onClick event-handler.

limit

A non-negative integer that specifies the maximum number of characters that the user can enter in the edit window; zero (or empty) specifies no limit.

where

When you set the where property, you can specify two elements or four. The fourth element defines the width of the list. By design, the width of the list matches the width of the edit box plus the arrow button that allows the user to access the list. If you want the entire selection to display in the edit box, you must make the value of where[4] wide enough so the longest selection does not extend past the end of the edit box. This leaves white space to the right of all the items in the list. (You can also specify the width using the size or extent properties. See the Windows Interface: Properties chapter in this manual for more information.)

When you set the where property, the height of the list, where[3], defines the space for both the edit box and the drop-down list. However, when you reference where, the third element returns only the height of the edit box. You can also define height using size or extent; the difference between setting and referencing the first element of either of these two properties is the same as for the third element of where.

imagelist

The name of the Imagelist object that contains the images you want to display to the left of items in the list. You use the origin one index to the images in this object to associate an image with a list item.

Event-handler Properties:

onAction, onChange, onClick, onClose, oncloseup, onDbClick, onDelete, onDestroy, onDragEnter, onDragLeave, onDragover, onDrop, onDropDown, onExit, onExitError, onFocus, onHelp, onHide, onKeyDown, onKeyPress, onKeyUp, onLimit, onMemory, onModified, onMouseDown, onMouseDrag, onMouseEnter, onMouseLeave, onMouseMove, onMouseUp, onMove, onOpen, onPaint, onReopen, onResize, onSend, onShow, and onUnfocus.

The onDbClick event-handler property is valid only for a simple Combo (style 1). The onChange event handler fires for changes made to the edit box portion of a Combo. It does not fire when the text property changes as a result of a user clicking an item in the list portion of the control. If you want to take an action based on a user selecting an item from the list, use the onClick event handler.

Methods:

Close, Copy, Create, Cut, Defer, Delete, Draw, Event, Exec, Focus, Help, Hide, Modify, New, Open, Paint, Paste, Popup, Ref, Resize, Send, Set, SetLinks, and Show.

Parentage:

Combo boxes are most often children of a Form, Frame, Page, or CommandBar. They may also be children of some control objects: Button, Check, Combo, Edit, Label, List, Option, Picture, Scroll, Spinner, Status, and Toolbox. The system allows most classes as children; it specifically prohibits Form, MDIForm, CommandBar, CommandButton, Page, and Printer.

Example:

This example shows how you create a Combo object and define an OnClick event handler for this object. In the example, the onClick event handler is an expression that the system executes directly. It saves the existing text from the edit window (before the new text value is displayed) to the data property.

```
c ← 'fmSample.cbTitle' □wi 'New' 'Combo'
c □wi 'where' 1 5 5 10
c □wi 'list' (>'Mr.' 'Mrs.' 'Ms.' 'Dr.')
c □wi 'style' 16 A Vertical scroll allowed
c □wi 'onClick' 'oldValue ← c □wi ''text'' ♦ c □wi ''data'' oldValue '
```

CommandBar

Prefix: bar

Summary:

The CommandBar is a rectangular area that serves as a container for controls and CommandButtons. These two classes work together to provide a dynamic tool bar/work area that the user can drag around the screen. When you create a command bar as a child of a form, by default it is docked at (attached to) the top of the form and extends the entire width of the form. However, it can dock on any of the four sides of the form, or it can float.



The picture shows a form with five command bars, one of which is a fixed-position corner display bar (Hello World); bar DDD is floating. CommandButton Two on bar AAA is a toggle button; the fourth button on bar BBB is drop-down. The mouse cursor is over the fourth button on bar CCC; the raised appearance of the button and the tooltip result.

In other applications, you may see these controls referred to as cool bars or Rebars. They generally have the appearance of toolbars, with entry fields mixed in among flat buttons that give a raised appearance when the user passes the cursor over them. These buttons, called command buttons, are a special class that can be children only of command bars. You can place images on command buttons, such that a different image appears at the same time the button is raised when the cursor moves over it. A third image can be assigned for showing when the button is disabled. If you don't specify a disabled image, the system "grays-out" the normal image. In addition to command buttons, a command bar can have as children any objects other than Form, MDIForm, Printer, or another CommandBar.

A CommandBar has a vertical bar, known as the gripper, at the far left. If you assign a caption, the caption appears beside the gripper. The bar can appear horizontally along the bottom of the form or vertically along either side. (If you have a vertical bar, the gripper is horizontal with the caption below it.) When you construct multiple bars, they can be configured in one band or in multiple bands, and they can be docked on one or more sides of the container. In addition, a bar can float; you can position it, or the user can move it to any spot on the screen, even outside the boundaries of the form itself.

The user can rearrange the relationships of command bars by double clicking on the label or the "gripper." If there are children on each bar, when the user double-clicks a bar that is minimized at the far right, it extends leftward until all its children are visible (its "ideal" length). If the user double-clicks again, the bar maximizes its length, covering the children of the bar to its left, leaving only the gripper and caption of that bar visible. If a bar at the left has its children covered, double-clicking on that bar expands it to the right far enough so that all its children are visible. Note that you can limit a bar to its ideal length, in which case double-clicking on that bar will "push" adjacent bars, but not necessarily minimize them. In the picture above, bar BBB is at its ideal length.

A user can also move a bar by placing the mouse over the gripper or the caption and dragging the bar. When the cursor is over the gripper, the cursor turns into a double arrow pointer. When dragging, the user can move the bar to a specific position in the band, to another band so each command bar extends the entire width of the form, or to another docking position.

When bars move around the form, the virtual client area of the form (as defined by the size or extent property) adjusts to exclude the space used by the bars, similar to the adjustment for the menu area along the top of a form or a Status bar along the bottom of a form. You can make command bars children of an MDI Form; they remain on the parent form along with whatever child forms are visible. You can also place command bars on a page of a selector or within the confines of a frame. If you use a relatively small frame as the container, and you have multiple bars which you allow the user to dock, you must size the frame carefully, or bars may be only partially visible.

Uses:

A command bar functions both as a toolbar and a working area. You can place almost any control that has a visible presence on the screen on a command bar. However, large objects, such as a Picture or a Listview, make a movable bar awkward. Edit fields and Combo boxes work well for horizontal bars. If they are long, they make a vertical bar undesirable.

Note that you can restrict the ability of the user to reconfigure the command bar locations. So, in general, command bars should provide all the functionality previously available with the Toolbox control and with floating toolbars (Form with border 5 1 2), with additional capabilities that are not present in either.

Regular Properties:

bitmap, border, caption, captiondock, captionfloat, children, class, color, data, def, deferexit, dock, dockable, dockbreak, dockheader, dockideal, docklength, dockmargin, dockmax, dockmaxdepth, dockmin, dockmindepth, dockside, dockvert, edge, enabled, events, extent, floatlayout, floatsize, floatwhere, font, help, helpcontext, hwnd, imagelist, imagescale, imagelistdisabled, imagelisthot, instance, keys, links, methods, mode, modified, modifystop, name, noredraw, normsize, opened, order, place, pointer, prompt, properties, scale, scrollaccel, scrollmargin, self, showtext, siblings, size, state, style, suppress, tabgroup, tabparent, tabstop, targetformats, theme, tooltipenabled, tooltipstyle, tooltiptime, tooltipwidth, translate, visible, where, and $\Delta ud p$.

Since the size, location and relationships of bars are primarily under control of the user, there are numerous properties that allow you to track the size and position of a bar. Using these properties, you can save the placement of the bars set by the user and restore them when restarting your application. You can also use these properties to readjust the bars or limit how the user can change their positions.

The first set of properties listed below describe the docking of bars. The next group of properties describe their size, both actual and potential. A third set of properties control floating command bars. Other properties allow you to place images on buttons and otherwise determine the appearance of the bar.

CommandBar Properties that affect Docking:**dockable**

An integer value that tells where a user can move the bar. Note that if you limit the user's options by setting the dockable property, the bar can still be docked at another location by setting the dock property. This property limits the location of the bar only through the user interface.

You can set the dockable property to a combination of any of the following values:

- 0 User may not change the docking location of the bar
- 1 Bar may be docked along the top edge of the container
- 2 Bar may be docked along the bottom edge of the container
- 4 Bar may be docked along the left edge of the container
- 8 Bar may be docked along the right edge of the container
- 16 Bar may be floated to any position on the screen

The default value is 3 1, unrestricted docking and floating. Although dockable always returns a scalar sum of the values, you can set it as a vector of any combination of the above values.

dock

An integer scalar or one-element vector showing the side of the container where the bar is currently docked or if it is floating. The possible values are:

0 = floating; 1 = top; 2 = bottom; 3 = left side; 4 = right side; or -1 to designate the bar is a client

Note that the sides are specified in the same order as for the `dockable` property, although the values are different. The negative value means you are treating this bar similar to a normal control, and that you have or are planning to specify its location explicitly by setting the `where` property. Such a bar does not have a gripper, does not dock, and is not controlled by the user. If you change the `dock` property to client mode from one of the other values, the system hides the bar, and you must explicitly Show it.

dockside

An integer scalar or one-element vector showing the side of the container where a floating bar will dock, if the user double-clicks on the label. If the bar is floating, you can set it to:

1 = top; 2 = bottom; 3 = left side; or 4 = right side

If the bar is docked, `dockside` has the same value as `dock`, and you cannot set the value to a different side; trying to do so gives a `NONCE ERROR`. If you do not change this property, it retains the value for the side where the bar was most recently docked. If you create a floating bar, it defaults to 1.

dockvert

A read-only Boolean flag that says whether the docked orientation is vertical (1) or horizontal (0).

dockbreak

A Boolean flag that says whether the bar starts on a new band (1) or follows another bar (0). You can set this property except that any setting is ignored for the first bar on any side of the container (the bar has an order property of 1) or for a `style 2` bar, which is always positioned in the upper right corner if docked at the top or the lower left if docked on a side. You can use this property when creating multiple bars to determine their initial arrangements.

CommandBar Properties that affect Size:

Many of the properties that describe or determine the size of command bars allow negative values; these values generally allow or prescribe default system behavior for determining size. Also note that under many circumstances, you can control only one dimension of a bar; the other dimension is determined by the system to allow the children of the bar to be visible. The `dockideal`, `dockmax` and `dockmin` properties control the behavior of the bar when it is docked with other bars. These three positions can occur, for example, when a user double clicks the captions on various bars on the same band. These properties do not necessarily return usable values. Use the `normsize` property to determine the "ideal" length.

docklength

The `docklength` property is useful primarily for saving the state of a bar, so you can restore it when the application is restarted. If you want to set the `docklength` and `dockbreak` properties, you should set them for all bars before any bars are opened. Setting `docklength` to an explicit value after a bar is open can give unexpected results.

This property describes the length of the bar in the bar's scale units when docked top or bottom or the height when docked on a side. It does not include the area that contains the gripper and caption. You can specify one of the special values:

-1 = set to ideal size; -2 = set to maximum size; or -3 = set to minimum size

This property does not affect the size of the bar when it is floating. (It retains the length it had most recently when docked if you have not specified one of the special values.) When a bar is moved to a new location, the system resets `docklength`.

dockideal

The `dockideal` property describes the length of the bar in the bar's scale units when it is expanded from zero but does not fill an entire band. It does not include the area that contains the gripper and caption. The default value is `-1`, which specifies the minimum length required to show all the children of the bar. If you set it to another value, the bar expands to the specified length, for example, when the user double-clicks on it when its length is zero (only the caption visible). This property retains its value when the bar is floated, but it resets to its default if the orientation of the bar changes from horizontal to vertical or reverse (that is, the `dockvert` property changes).

dockmax

The `dockmax` property describes the maximum length of the bar in the bar's scale units when docked top or bottom or the height when docked on a side. It does not include the area that contains the gripper and caption. The default value is `-1`, which means unrestricted. If you set it to another value, the bar does not expand beyond that length unless it is the only bar on a band, in which case it fills the entire width or height of the container. This property retains its value when the bar is floated, but it resets to its default if the orientation of the bar changes.

dockmin

The `dockmin` property describes the minimum length of the bar in the bar's scale units when docked top or bottom or the height when docked on a side. It does not include the area that contains the gripper and caption. The default value is `-1`, which means the bar can collapse to zero (leaving only the caption visible). If you set it to another value, the bar does not collapse shorter than the specified length. This property retains its value when the bar is floated, but it resets to its default if the orientation of the bar changes.

normsize

The `normsize` property is a two-element, read-only numeric vector measured in the bar's scale units; it describes the calculated height and width of the bar required to show all the children of the bar when docked. It does not include the area that contains the gripper and caption. The values change when the vertical/horizontal orientation changes. If the bar is not open, the system returns `0 0`. One element (depending on orientation) of this property determines the default value for `dockideal`. When the bar is floating, it retains the values for the docked state.

dockmindepth

The `dockmindepth` property is measured in the bar's scale units; it describes the minimum size of the bar perpendicular to its length when docked. The default value is `-1`, which allows the bar to collapse to the minimum size of the largest child in the appropriate dimension. If you set it smaller than the default, the system ignores the setting.

dockheader

The `dockheader` property is measured in the bar's scale units; it describes the length of the area that holds the gripper and caption in the same direction as `docklength`. The default value is `-1`, which specifies the minimum length required to hold the entire caption.

dockmargin

The `dockmargin` property is measured in the bar's scale units; it is a read-only four-element vector that describes the lengths on the four sides of a command bar that are included in the total screen space required to hold the bar but are not included in the `normsize` values. The margins include the area of the dockheader, plus the size of the pixels required for the edges.

where, extent, and size

A command bar has the normal `where`, `extent` and `size` properties, which you can reference. However, you do not want to set these properties, particularly `where`, unless you set the bar to be a client of the container, in which case it acts like other controls and unlike a command bar. If you want to position a floating bar, use the `floatwhere` property.

If you want to position a docked bar, use the `dock` and `order` properties, augmented by the previously set `docklength`, if necessary. With normal command bars, these properties do not include the portion of the bar represented by the `dockmargin` property, which includes the area for the gripper and caption.

Also note that, because the client area of a form excludes the space taken by bars, one of the first two elements of the `where` property is negative for bars docked at the top or left side of the container. For floating bars, `where` is relative to the screen and not the form or other container.

CommandBar Properties that affect Floating Bars:

A command bar changes appearance when it is undocked to a floating state. It resembles a small independent form, such as a tool palette (form with `border 5 12 + 18`); the caption appears in a short title bar, along with a close button, and you can right-click on the title bar to access a system menu (unless you suppress these with style settings). Children of the command bar appear in one or more rows beneath the title bar. The user can move the floating command bar with the mouse and can also resize it by dragging the resizable border. As the user changes the size, the contents of each row on the bar may change.

Note that the command bar remains part of the form that is its parent. Even if the user moves the bar into an area that is completely separate from the form and covers it with a window from another application, when he clicks on the form, the command bar becomes visible along with the form. You can use the following properties to control the position and layout of a floating command bar.

floatwhere

A four-element vector showing the location (row, column) relative to the screen origin, not the container of the bar, and size (height, width) of the outside edge of a floating bar in screen units (characters). If the bar is closed or docked, but the position has been defined, this returns a two-element vector showing location. If the bar has never been floated and this property has not been set, it returns an empty vector.

The third and fourth elements of `floatwhere` change along with the two elements of the `floatsize` property, although they may be in different units. They differ slightly because `floatwhere` defines the outside edge of the floating bar, while `floatsize` defines the inside edge.

If you set `floatwhere` to an empty vector, it returns to an undefined state. If you set it to a two-element vector, it defines a new position; setting it to a four-element vector defines both position and size, except that the size dimensions are treated as constraints and not fixed values.

You can constrain the length of either dimension, height or width; the system adjusts the other to make all children of the bar visible. If you assign positive values to both dimensions, the system treats one as a priority. You can set special values to predetermine which dimension the system favors; however, you can also set these constraints and special values in `floatsize`. It may be better to do so, because the settings are retained in `floatsize`, but not in `floatwhere`, when the bar is docked. The settings are described under `floatsize` below.

floatsize

A two-element vector showing size (height, width) of the inside edge of a floating bar in command bar units. These values exclude the floating border and title bar. If the command bar has never been floated and this property has not been set, it returns an empty vector. If the bar is closed or docked, it shows the previous settings. If you set this property, the system treats the values as constraints. You can constrain either dimension, height or width; the system adjusts the other to make all children of the bar visible. If you assign positive values to both dimensions, the system treats one as a priority.

You can set the following values to specify how the system arranges the command bar when it floats.

Settings for floatsize (or floatwhere)

Height	Width	Result
0	0	fixed layout mode; system creates new rows after command button wrap.
-1	0	vertical layout mode; system places one child on each row.
0	-1	horizontal layout mode; system places all children on one row.
m	0	constrain vertical dimension; system limits height to m units.
0	n	constrain horizontal dimension; system limits width to n units.
m	n	constrain priority dimension; use floatlayout or previous mode. (See note.)

Note: When you float a command bar, the system determines the layout by constraining one dimension and varying the other to make all the children fit. If you specify constraints for both dimensions, the system chooses one as the priority. If the system recently performed a layout, for example, because you set `floatsize`, it continues to use the same dimension as priority. If you set the `floatlayout` property, the system uses that setting to determine the layout mode; however, it will use the dimension-constraining mode rather than the one row or one column mode if `floatsize` is specified.

floatlayout

A scalar integer showing how a bar should be sized when it is made to float. The possible values are:

- 0 No automatic wrapping (use wraps specified on command buttons only).
- 1 Horizontal layout mode; use one row.
- 2 Vertical layout mode; use one column.
- 3 Constrain the layout width by the value specified in the second element of `floatsize`.
- 4 Constrain the layout height by the value specified in the first element of `floatsize`.

Other CommandBar Properties:

The following properties define the bar in ways other than docking and position.

order

Integer scalar or one-element vector that defines the sequential position of the bar among its contemporary siblings. Among the set of siblings, order values are a sequence of positive integers starting with 1. A command bar can belong to one of six different sets of controls that each have their own order values. Bars docked at the top, bottom, left side and right side of the container are each a separate set of controls for purposes of order. Floating bars have their own set of order values. If you make a bar a client of the container (by setting the first element of `dock` to -1), it becomes part of the set of all "normal" controls that are children of the container. Usually, you will not have members of all the possible sets of controls that have order properties.

As bars are moved within a dock location, for example at the top of the form, the value of the order property can change, but the set remains the same. If the bar is moved to a different side of the container, it becomes part of a different set of siblings, and the values of the order properties for some or all of those controls may change. If you want to control how command bars are positioned when the user moves one, you can reference the siblings property and set the order property. (See the description of the siblings property in the Windows Interface: Properties chapter in this manual.)

imagelist, imagelisthot, and imagelistdisabled

The name of the `Imagelist` object that contains the images you want to display on command buttons. You use the origin one index to the images in these objects to associate an image with a command button. The same index is used for all three imagelists. The image from `imagelist` displays on a command button when it

is enabled. The corresponding image from `imagelistdisabled` displays when the button is disabled. The corresponding image from `imagelisthot` displays when the user moves the cursor over the button using the mouse. These images also interact with the non-default settings of the `value` property of a command button. If you set only the `imagelist` property, only one image appears for each button; it is modified by the appearance of raised edges when the cursor is over the button and by system greying when it is disabled.

Your images appear best if they are all the same height. The size of your images determines the height of the button and hence the size of the command bar, unless you set the size specifically or add controls that make it larger.

bitmap

A character vector that specifies the bitmap you want to appear as the background on the bar. If you have multiple bars docked on the same edge of the container, and they use the same bitmap, the image is drawn over the bars as a unit, not on each individual bar. See the picture above.

theme

The `theme` property of a `CommandBar` class controls the visual styling theme used for drawing the control. The default value is an empty character vector (`' '`), which uses the new visual styling for the control. To suppress the new visual styling for the control, set `thememode` to `"~"`.

style

An integer vector that contains selected values or an integer scalar that is the sum of the selected values. The initial value for `style` is zero. The table below shows the possible values.

Values for the CommandBar Class style Property

Value	Description
Primary Value: Choose only one value	
0	Default command bar.
1	Place button captions at side of image instead of below.
2	Fixed-position corner display bar. (See note 1.)
Add-on Values: Choose these values with appropriate restrictions	
4	Reduce space surrounding image or control.
8	Remove Gripper from bar. (See note 2.)
16	Fixed wrap positions. (See note 3.)
32	No caption. (See note 4.)
64	Fixed height bar. (See note 5.)
128	Floating bar cannot be closed by menu or close button. (See note 4.)
256	Do not show menu or close button on floating bar. (See note 4.)
512	Command buttons look like normal buttons; raised, no hot images.

Note 1. By default, this style bar locates itself in the upper right corner when docked at the top edge (lower left if docked on a side). It has no gripper and cannot be dragged or double-clicked. It automatically sizes itself to accommodate any child object you place on it. If all other bars are removed from the same edge of the container, this bar expands to fill the entire available width or height, but it will not otherwise change length. This style allows you to place a logo or an animated display at a fixed position in your application.

Note 2. When you remove the gripper from the left (or top) of the command bar, the user cannot position the bar with the mouse, cannot change its relationship to other bars in the same band by double-clicking on the label, nor move the bar to another docking position. The bar still responds to changes in other bars docked at the same edge. For example, when the bar is pushed all the way to the right edge of the container

so none of its children are showing, if the user moves the other bars in the same band to new docking rows, the bar with no gripper becomes visible and fills the entire band.

Note 3. When you float a command bar, the system may arrange the children on multiple rows, or it may allow the bar to float as one band. If you assign this style to the bar, the system always breaks the bar after a command button to which you assigned a style property so that it wraps. This enables you to make tool palettes with multiple specific rows.

Note 4. You may choose only one of these three styles: No caption, No Close, No menu. Choosing style 32 deletes the entire title bar area from the floating command bar; style 128 leaves the system menu enabled, but the Close item on it disabled, as is the Close button; style 256 deletes the Close button and does not allow the system menu to pop up. When you specify style 32, the user can drag the command bar to a floating location. But once it is dropped the user cannot move it because there is no title bar with which to reactivate it.

Note 5. When you have multiple bars with different size images docked on the same edge of the container, this style causes all the bars on all bands on that edge to display with the tallest height of any bar docked on the same edge, regardless of whether it is on the same band or not. Without this style, the bars on each band display at the maximum height of any bar on that band. You may want to assign this style value to each bar you create if you assign it to any.

caption

A character vector that specifies the text you want to appear as a label for this bar. The default value is empty. If one of the other two caption properties, `captiondock` or `captionfloat`, is set, the value of this property shows for the opposite state. That is, if you set `captionfloat`, this caption shows when the bar is docked. If both of the other properties are set, this caption does not appear.

Note that you should be careful with long captions for a docked bar; when a bar is docked top or bottom, the header size expands to show the caption. If the bar is moved to a side, the caption may be truncated if the image width or the largest child control is narrower than the caption.

captiondock

A character vector that specifies the text you want to appear as a label for this bar when the bar is docked only. The default value is empty.

captionfloat

A character vector that specifies the text you want to appear as a label for this bar when the bar is floating only. The default value is empty. If style includes 32, this caption does not appear.

showtext

A Boolean flag that you can use to show (1) or hide (0) captions on command buttons.

edge

Setting a non-zero `edge` property for a command bar does not interact well with the size of command buttons. Furthermore, the command bar does not interact well with non-zero `edge` property settings of its container. If you want the appearance of a 3-D edge, you can use a style 1 frame that you size to fit the client area of the form; that is, the area excluding the command bar.

Event-handler Properties:

`onAction`, `onClose`, `onDelete`, `onDestroy`, `onDockChange`, `onDragEnter`, `onDragLeave`, `onDragover`, `onDrop`, `onExit`, `onExitError`, `onFocus`, `onHelp`, `onHide`, `onHotChange`, `onKeyDown`, `onKeyPress`, `onKeyUp`, `onModified`, `onMouseDouble`, `onMouseDown`, `onMouseDrag`, `onMouseEnter`, `onMouseLeave`,

onMouseMove, onMouseUp, onMove, onOpen, onPaint, onReopen, onResize, onSend, onShow, and onUnfocus.

onHotChange

The HotChange event fires for the bar when the user moves the mouse cursor onto or off of a control or command button that is a child of the bar. `EventArgs` is a two element nested vector that shows the name of the child that is currently under the cursor and the name of the child that the cursor just left. Either the first item or the second item may be an empty character vector.

onDockChange

The DockChange event fires for the bar whenever the dock state (dock property) changes. During the event handler, the bar is hidden while it transitions. `EventArgs` is a two-element integer vector indicating the new and old values of the dock property.

Methods:

Close, Create, Defer, Delete, Draw, Event, Exec, Focus, Help, Hide, HitTest, Modify, New, Open, Paint, Popup, Ref, Resize, Send, Set, SetLinks, and Show.

Parentage:

CommandBars may be children of a Form, MDIForm, Frame, or Page. The system allows most classes as children; it specifically prohibits CommandBar, Form, MDIForm, Menu, Page, and Printer.

CommandButton

Prefix: b

Summary:

A CommandButton is exclusively the child of a CommandBar. Initially, a flat surface with no visible border or edge, when the cursor is moved over it, it gives a raised appearance. You can place an image and a caption on a command button. It serves as a multi-purpose placeholder, control, and device to trigger an action.

Uses:

A command button can act like a normal button or a toggle button; it has an `onClick` event handler. As a toggle button, it can also function in a set of buttons like Option controls, where only one of the set can be selected at a time. It can also be a two-part control like a combo box, with an arrow that causes a drop-down display, as well as the normal button behavior. You can use buttons as separators on the command bar, like menu separators, or as spacers between controls. The command button allows you to configure a command bar into a dynamic work area with a specific appearance and layout.

Regular Properties:

caption, children, class, data, def, enabled, events, help, helpcontext, image, imagescale, instance, links, methods, modified, modifystop, name, opened, order, properties, self, state, style, suppress, tooltip, translate, value, visible, where, width, wrap, and `Δudp`.

style

An integer vector that contains selected values or an integer scalar that is the sum of the selected values. The default value for style is zero. The table below shows the possible values.

Values for the CommandButton Class style Property

Value	Description
0	Normal button
1	Separator (See note 1.)
2	Toggle button
4	Part of a group of radio buttons; must be used with 2. (See note 2.)
8	Drop-down button (See note 3.)
16	Automatically resize button to fit text. (See note 4.)
32	Don't underline character following ampersand (&) in caption.

Note 1. The separator has a visible appearance on the bar, similar to the separator style for a menu. You can use this to provide a visual clue as to a related group of buttons or other controls. The separator defaults to 8 pixels width. You can also use a normal, but disabled, button with no image on a command bar as a spacer with no visual presence, for example between adjacent edit boxes.

Note 2. You can make a group of toggle buttons act like radio buttons, that is, only one of the group can be selected at a time. If the user clicks on one, it unselects the one that was previously selected. If you want multiple groups, you must separate them with a button that has a different style value. **Warning:** Do not set a style 2 toggle button adjacent to a group of style 6 buttons.

Note 3. The drop-down button has two parts; the normal button that triggers an `onClick` event handler when clicked and an arrow to the right that triggers the `onDropDown` event handler, but not `onClick`. You can use this arrow, for example, to expose a popup menu anywhere on the screen.

Note 4. The automatic resize style is for use in conjunction with the command bar style 1, which has the captions beside the image.

image

An integer that is the origin one index into the `Imagelist` objects you specify for the `CommandBar` that contains this `CommandButton`. The same index is used for all three imagelists, but the image may vary depending on whether the button is enabled and whether the cursor is over the button.

caption

A character vector that contains the caption you want to appear on the button. If you set the caption property for any button on a command bar, you should set it for all the buttons on that bar. If you do not set captions for all the command buttons, you can put yourself into the position where none of them appear. You can use the `showtext` property of the command bar to toggle the display of captions on and off without changing the captions themselves.

width

An integer value that represents the width of the button in the parent command bar's scale units (since a `CommandButton` does not have a `scale` property). The default value is zero, which causes the button to take the width of the images from the imagelist named in the bar's `imagelist` property. (Note that the height of the button is the largest of the three image sizes from the bar's imagelists, if all three are defined.) A `CommandButton` does not have a `size` property; its `where` property is read only. Captions on a command button can affect height or width.

value

The value property normally has meaning only for a toggle button. If the button is selected, the value is 1; the default value is zero. You can also set the value property. Windows provides some built-in effects for various values, including showing the button pressed down and different background colors or overlays. In addition, the image that appears may change, depending on which imagelists are set on the parent command bar. You can set combinations of the values through 15 with varying effects. In particular, values 8 through 15 may depend on system settings as well as what images are set. The values are:

Values for the CommandButton Class value Property

Value	Description
0	Normal – flat button; normal image
1	Selected – sunken appearance; image from imagelisthot
2	Pressed – sunken appearance; normal image (See note.)
4	Indeterminate – flat appearance but image may be masked, greyed, or invisible; image from imagelistdisabled, if set.
8	Marked – flat button, but image may be overlaid or otherwise accented.

Note: Although this is designated as pressed, it is not the appearance the button has when the user holds it down with the mouse. When the mouse is over the button, it has the hot image.

wrap

A Boolean flag that specifies you want the bar to wrap to a new line after the command button that has this property set to 1. This takes effect only when the style of the command bar includes 16, fixed wrap positions. Note that when you float any bar with multiple children, the underlying Windows system may display it on multiple layers. Using style 16 on the bar forces the wrap at the location you specify with this property. When the bar is floating, and its style does not include 16, you can reference this property to determine where the system has caused the bar to wrap, but setting this property to 1 for any button on the bar is ignored.

Do not confuse the effect of setting the wrap property on a command button with the effect of setting the dockbreak property on a command bar. The former causes the next child of the bar to wrap to a new line within the same bar; the latter causes the entire bar to move to a new band on the side where it is docked.

Event-handler Properties:

onAction, onClick, onClose, onDelete, onDropDown, onHelp, onModified, onOpen, onReopen, and onSend.

onDropDown

This event handler fires only for the style 8 button that has an associated drop-down arrow. It is triggered when the user clicks the drop-down arrow on the right side of the button. This is independent and mutually exclusive from the Click event that is triggered when the user clicks the left side of the button.

Methods:

Close, Create, Defer, Delete, Event, Exec, Help, Hide, Modify, New, Open, Popup, Ref, Send, Set, SetLinks, and Show.

Popup

The Popup method has options that allow you to position the menu; for example, you can specify a character vector that is the name of the control as the first optional argument, after the name of the menu, to display the menu immediately below that control. This is particularly useful for the style 8 drop-down command button, so the menu does not obscure the button when you display a pop up menu as the result of the user clicking the drop-down arrow. See the description of the Popup method for more information. Note that

you can use relative naming to refer to the object on which the `Popup` method is being invoked or its children; for example:

```
□wi 'onDropDown' "□wi 'Popup' 'menu' '.' "
```

Parentage:

`CommandButtons` may be children only of a `CommandBar`. A `CommandButton` can contain a `Timer` or a hidden `Menu` for the purpose of assigning a shortcut key .

DateTime

Prefix: dt

Summary:

A `DateTime` selector is a facility that allows a user to select a date, or alternatively a time, by using the mouse or the keyboard. By customizing the display fields, you can allow both date and time in the same control.

Uses:

The date portion of the `DateTime` control comes in two forms: by default, it has a visible calendar that shows one or more months (up to 12); the user scrolls the display, if necessary, to make the desired month visible, and selects a date by clicking on it. The chosen date, or date range, is highlighted. The calendar has a maximum range from September 1752 (when the last major calendar revision occurred) up to December 65535. You can modify the range within those limits by setting the `range` property. The user can change months either by clicking on buttons (provided by the control) or by using the `PageUp` and `PageDown` keys. When using the calendar, the user can also change the date by using the arrow keys. The selection moves automatically from month to month as the user arrows.

If the user clicks on the year in the title of a calendar display, the control provides spinner buttons for changing the year; the display does not change until the user clicks elsewhere on the control. Similarly, if the user clicks on the month in the title of a calendar display, the control provides a drop-down list of months from which to select. You cannot modify these features of the control from APL; they are behavior inherent to the control.

The second form of date selector has a field that resembles a combo box, where the edit field displays the date and an arrow allows the user to drop down a one-month calendar. The user selects a date as with the default style, and the calendar closes up, leaving the selected date displayed in the edit field.

Alternatively, you can specify spinner buttons for the date instead of having a drop-down calendar. The user can change the year, month, day, or day of week individually, using the spinner buttons or typing directly into the edit field. You cannot have a date range with other than the default style.

When used as a time selector, the control resembles an edit control that displays hours, minutes and seconds with spinner buttons that can change any one of the three values. The user can select the value to change by clicking on one or moving among them with the arrow keys; the selected value is highlighted. When choosing a time, there is an implied date, either the current date or one that was set under program control.

You can customize the display of the date or time in the edit field. With any of the non-default styles, drop-down calendar, date with spinner buttons, or time with spinner buttons, you can choose which fields to include so that a user could select both date and time in the same control. See the "Customizing the Display of the Date or Time" section below for more information on customizing your display.

The programmatic representation of the date (and time, for styles other than the visible calendar) is a vector contained in the `value` property. Note that the value changes when a user changes the month on a calendar,

for example by clicking on the buttons provided by the control, even if the user does not click on a date to select it.

Regular Properties:

border, children, class, color, data, def, deferexit, dropcolor, dropfont, edge, enabled, events, extent, firstday, font, format, help, helpcontext, hwnd, hwnddropdown, instance, keys, limit, links, methods, minsize, mode, modified, modifystop, monthdelta, name, noredraw, opened, order, place, pointer, prompt, properties, range, scale, scrollaccel, scrollmargin, self, size, state, style, suppress, tabgroup, tabparent, tabstop, targetformats, text, theme, today, tooltip, tooltipenabled, tooltipstyle, tooltipwidth, translate, value, visible, where, and Δudp .

style

An integer vector that contains selected values or an integer scalar that is the sum of the selected values. The initial value of **style** for any newly-created control object is 0. Use one of the primary values plus any of the add-on values appropriate for the primary value. Styles 1 and 2 provides a drop-down calendar unless you suppress it with style 16.

Values for the DateTime Class style Property

Value	Description
Primary Value: Choose only one value	
0	Calendar control
1	Date Selector with short date format.
2	Date Selector with long date format.
3	Time Selector.
Add-On Values: Choose any of the values with the appropriate Primary value	
4	Allow a range of consecutive dates to be picked for calendar control.
8	Display week (of the year) number to the left of the calendar.
16	Provide up/down buttons instead of the drop-down calendar. (See note 2.)
32	Allow blank for value. (See note 3.)
64	Do not circle today's date on calendar.
128	Do not display today's date below calendar.
256	Right align the drop down calendar.
512	Clicking calendar with mouse does not move focus to DateTime control.

Note 1. There are three different types of control; the visible calendar; the date selector, usually with a drop-down calendar, and the time selector. Some properties and behaviors are common to the calendar and date selector styles. Other properties and behaviors are common to the date and time selector styles but differ from the visible calendar. When you customize the display to include more than the default fields, styles 1, 2, and 3 are essentially variations on each other.

Note 2. Style 16 is automatic with the Time selector (3); there is no drop-down calendar. You can add 16 to the date selector, styles 1 or 2, to suppress the drop-down calendar and provide spinners.

Note 3. You can use style 32 with any of the three non-zero primary values. This style provides a check box in the control's edit field. If the box is checked, the value property reflects the display. If the check box is not checked, the value is empty, regardless of the display. In addition, when the check box is unchecked, the control is disabled, in the sense that the user cannot enter a value from the keyboard, use the arrow keys, or use spinner buttons, if they exist, until the box is rechecked.

This behavior is subtle, and it can be confusing if the user of the application is unaware of its consequences. The control becomes active again when the user clicks on the checkbox with the mouse, presses the spacebar

when the focus is on the control, clicks on the drop-down calendar button, if that exists, or if the value is reset by the program.

theme

The `theme` property of a `DateTime` class controls the visual styling theme used for drawing the control. The default value is an empty character vector (' '), which uses the new visual styling for the control. To suppress the new visual styling for the control, set `thememode` to " ~ ".

value

When you do not have `style 4`, allowing a range of dates, this property is a simple numeric vector.

When you reference the `value` property, it is either a three-element or a seven-element numeric vector. If you are using the `calendar style`, it is a three-element vector showing the year, month, and date. If you are using either of the drop-down date styles or the `time style`, `value` is a seven-element vector, such as that provided by `⌈ts`: year, month, date, hours, minutes, seconds, milliseconds.

When you set the `value` property for any `style`, you can use a seven-element vector. You can also use a six-element vector, ignoring milliseconds. In this case, the milliseconds value is set to zero.

For any of the date styles, you can specify a three-element vector, designating year, month, and date. When you do this, the time elements are left unchanged.

For the `time style` only, you can specify a three- or four-element time, designating hour, minute, second and, optionally, millisecond. The date portion of the `value` property remains unchanged, either the last value provided by the program or the default current date.

When you have `4` included in the `style`, which is valid only with the `style 0` calendar control, `value` is a two-element nested vector showing the first and last dates of the range selected. If only one date is selected, the two vectors are the same. If you have the multiple date style selected, you can set either one date or the first and last dates of a span. You cannot have a range of times for a `style 3` control or a range of dates for styles `1` or `2`.

limit

An integer singleton that specifies the maximum number of days allowed in the calendar control with `multiselect (style 4)` selected. This property defaults to `1` if the primary style is not zero, or if primary style is zero and `4` is not selected. Note that it is meaningless with styles `1`, `2`, or `3`. The default value is `7` for primary style zero with `style 4` selected.

today

For the visible calendar this property is a `value` structured like the non-nested `value` property showing the day marked as "today" on the calendar. For the other styles, this property is initially empty; you can set it to a three-element numeric date vector.

range

A value structured like the nested `value` property showing the minimum and maximum dates or times the control allows the user to select. You may set either or both elements; if an element is an empty vector, there is no limit for that end of the span other than the built-in limits of the control.

firstday

A singleton integer or two-element integer-valued vector identifying which day of the week is displayed in the first column of the calendar for either the visible or drop-down calendar styles. Values `1` through `7` represent Monday through Sunday respectively; the value zero means the locale-specific default.

When referencing `firstday`, if the property has not been set, or if it has been set to zero, `firstday` is a two-element vector, with zero as the first element and the current locale-specific first day as the second element. If the property has been set to a nonzero value, it returns a scalar with that value. When the drop-down calendar is not dropped down, and the value is not set to a non-default value, this will be a scalar zero.

When you specify `firstday`, you specify only the first element. If you specify two elements, the system ignores the second.

monthdelta

A scalar or one-element integer giving the number of months that the control scrolls when the user clicks the scroll button. The value defaults to the number of months that are visible at one time; for the visible calendar style, you can display more months by making the control larger.

minsize (read-only)

A two-element numeric vector showing the minimum value for the `size` property that allows a single month to be displayed completely. You cannot reference this property while the object is closed. If the primary style is non-zero, and the calendar is not dropped down, this property returns an empty vector.

color

A vector of up to six color values. For the calendar the six elements of color are:

- Background within the month
- Text within the month
- Background for the title; names of days; week numbers; circle around selected day
- Text for the title; selected days
- Background between months
- Text for leading and trailing days around the month

Each color may be specified as you specify other colors, either as an enclosed three-element vector (RGB) or a singleton composite number. A negative one for either of the first two elements specifies that the color is inherited from the parent control; negative numbers less than -1 specify system colors designated on the host machine. Negative numbers for other elements specify system colors. The default values are all -1 .

Unlike other color properties, a three-element vector that is not enclosed specifies three colors rather than a three-element single color. Similarly, a six-element integer vector specifies six colors.

The `color` property does not currently affect the edit box for styles other than the visible calendar. However, the `dropcolor` property inherits its values by default. Thus, if you set `color`, the values affect the drop-down calendar. If you set `dropcolor`, the values for `color` are irrelevant. **Note:** Microsoft may change the underlying behavior to allow colors in the edit box in the future.

dropcolor

The `dropcolor` property has the same form as the `color` property. It applies to the drop-down calendar for those styles. The `dropcolor` property inherits values from the `color` property of this control rather than the parent.

dropfont

This property has the same format as the `font` property. It specifies the font used by a drop-down calendar.

hwndddropdown

The window handle of the drop-down calendar when it is dropped down; otherwise, zero.

Event-handler Properties:

onAction, onChange, onClose, onCloseUp, onDelete, onDestroy, onDragEnter, onDragLeave, onDragover, onDrop, onDropDown, onExit, onExitError, onFocus, onFormatQuery, onFormatString, onGetDayState, onHelp, onHide, onKeyDown, onKeyPress, onKeyUp, onModified, onMouseDown, onMouseDrag, onMouseEnter, onMouseLeave, onMouseMove, onMouseUp, onMove, onOpen, onPaint, onReopen, onResize, onSelect, onSend, onShow, onUnfocus, and onUserKeyDown.

onChange and onSelect

The onChange event handler fires for any style whenever the user changes the value, even if the new value is not selected. Note that you may have to handle multiple occurrences of this event for each change. In particular, for a visible calendar, you may get two Change events each time the user changes the date without selecting it, for example, by scrolling the display of months. In addition, you may get periodic spurious Change events for a visible calendar-style control.

The onSelect event handler applies only to the visible calendar control. It fires when the user clicks on a date. The onSelect event handler fires after the onChange event handler.

onGetDayState

The GetDayState event occurs for the visible calendar only; it fires when a program action changes the display of the month or months or when a user causes the display of the months to scroll.

You can use the onGetDayState event handler to display dates in bold. Specify the dates you want to have in bold in a three-column matrix in `□wres`. The system initializes `□wres` to a three-column, empty, numeric matrix so you can concatenate (⌘) the dates you want in bold. Alternatively, you can simply reset `□wres` to a three-column matrix of dates.

To use this event-handler, you need to recognize that the calendar display shows parts of two more months than those whose names appear in the title bar. The control displays up to a week of the end of the month preceding the first titled month and up to two weeks of the beginning of the month following the last titled month in the display. You can set some or all of these dates in bold, as well as any of those in the months that are displayed in full.

When the event-handler is triggered, `□warg` contains a two-element, nested numeric vector. The first element is a three-element vector that contains the date of the first of the month for which the user can see only the last few days. That is, if the first (or only) full month on the calendar display is January 2003, then the first element of `□warg` is 2002 12 1. The second element is a singleton representing the number of months or parts of months that appear in the display. That is, if there are three full months displayed, the second element is 5.

onDropDown and onCloseUp

These handlers are fired for the drop-down calendar only.

Methods:

Close, Create, Defer, Delete, Draw, Event, Exec, Focus, Help, Hide, HitTest, Modify, New, Open, Paint, Popup, Ref, Resize, Send, Set, SetLinks, and Show.

HitTest

You can use the HitTest method for the visible calendar only; it defines what portion of the calendar is located at a location specified in a row and column coordinate position. The argument to this method is a two-element numeric vector that specifies the vertical and horizontal location within the calendar in its current scaling units. You might get these values from, for example, the first two elements of `□warg` from an onMouseDown event handler.

The method returns a two-element nested vector showing the part of the calendar at that spot. The first element is a date if the location is on a date; otherwise, it is empty.

The second element is a character vector showing one of the applicable location values provided by Windows. The possible values for the second element are: `calendarbkgnd`, `calendardate`, `calendardatenext`, `calendardateprev`, `calendarday`, `calendarweeknum`, `titlebkgnd`, `titlebuttonnext`, `titlebuttonprev`, `titlemonth`, `titleyear`, `todaylink`, `nowhere`, or empty (if the control reports an unknown value). The values that include "prev" and "next" refer to the partial months preceding and following the months displayed in full. The "todaylink" entry refers to the display of today's date below the calendar.

Parentage:

DateTime controls are most often children of a Form, Frame, Page, or CommandBar. They may also be children of some control objects: Button, Check, Combo, Edit, Label, List, Option, Picture, Scroll, Spinner, Status, and Toolbox. DateTime objects can contain Timers, Imagelists, or hidden Menus.

Example:

The following example shows a visible calendar style DateTime control with two full months displayed. The `onGetDayState` event handler puts the 12th and 27th of every month (payments due), including the partial months displayed before and after the two full months, in bold. Additionally, Christmas of every year is placed in bold. The last entry shows that you can use a wild card value for the year in this event handler to make dates bold regardless of the year, for example, for birthdays, etc. (Note that this simplified example would not work for calendars that display more than ten months.)

```

sink<'fmCal' □wi 'New' 'Form' ('where' 1 1 14 50)
sink<'f.cal' □wi 'New' 'DateTime' ('where' 1 1 12 46)
'f.cal' □wi 'onGetDayState' 'embolden'

▽ embolden
[1] (prev span)←□warg
[2] □wres←□wres, (prev+0 0 26)
[3] :for i :in 1span-1
[4]   :if (prev[2]+i)>12
[5]     yr←prev[1]+1
[6]   :else
[7]     yr←prev[1]
[8]   :end
[9]   mo←1+12|(prev[2]+i-1)
[10]  □wres←□wres, yr mo 12
[11]  □wres←□wres, yr mo 27
[12] :endfor
[13] □wres←□wres, -1 12 25

```

Customizing the Display of the Date or Time

If you are using one of the drop-down calendar styles or the time picker, you can customize the display of the edit field. There are three ways to customize the display. First, you can add literal strings to the field, either permanently, such as "Today is:" or conditionally, such as "Merry Xmas" when the user selects December 25. Second, you can choose the types and formats of the standard date and time fields. Using the `format` property, you can specify the existence, order, and format of any or all of the eight fields: day of the week, date, month, year, hours, minutes, seconds, and AM/PM designator. Third, you can specify the existence of "callback" fields that hold information you specify.

text (read-only)

The `text` property is a read-only character vector with the contents of the edit box for styles other than zero. If the style is the visible calendar, this property is an empty vector. If the object is closed, the system signals an error.

format

The format property is a character vector that specifies how the contents of the edit box are displayed for styles other than the visible calendar. You specify the fields using the codes listed below. They appear in the control in the order you specify them with the formatting characteristics associated with the codes.

literal strings

You can place literal strings in the format property by enclosing the string with quotes. Since the property is a character vector, you must use double quotes and single quotes or double your quotes. You can insert punctuation marks and spaces without using quotes; however, Microsoft warns that this behavior may give unpredictable results for anything other than the comma.

date and time fields

You can specify the formats of these fields:

Day of the week	Enter ddd for the three-character weekday abbreviation or dddd for the full weekday name.
Date	Enter d for the one- or two-digit day or dd for the two-digit, zero-filled day.
Month	Enter M for the one- or two-digit month number, MM for the two-digit, zero-filled month number, MMM for the three-character month abbreviation, or MMMM for the full month name.
Year	Enter yy for the last two digits, zero-filled, of the year, or yyyy for the full four-digit year.
Hour	Enter h for the one- or two-digit hour in the 12 hour format, hh for the two-digit, zero-filled hour in the 12 hour format, H for the one- or two-digit hour in the 24 hour format, or HH for the two-digit, zero-filled hour in the 24 hour format.
Minute	Enter m for a one- or two-digit minute; mm for a two-digit, zero-filled minute.
Second	Enter s for a one- or two-digit second or ss for a two-digit, zero-filled second.
Morning - Afternoon designator	Enter t for a one-letter abbreviation (A or P) or tt for a two-letter abbreviation (AM or PM).

Note: The appearance of the display in formatted date and time fields depends on the Regional settings on your computer. If, for example, you want to change the morning/afternoon designator to lowercase on your own system, you can change the Regional settings. If you want to force such a display in an application that runs on many systems, you must use callback fields and specify the format yourself.

callback fields

You can specify one or more callback fields by using uppercase Xs. You must use a different number of Xs for each field, but that number does not modify the appearance or format of the field; it merely designates which callback field it is. You control callback fields using event handlers.

You specify the size of the field in the onFormatQuery event handler and the contents of the field in the onFormatString event handler. If you want the user to be able to modify the field directly using the keyboard, you use the onUserKeyDown event handler.

onFormatQuery

You use the onFormatQuery event handler to set the (maximum) size of any callback fields. The FormatQuery event occurs for each callback field defined in the format property; `pwarg` is a character vector that contains the identifier of the field, that is 'X' or 'XX', etc. (Note that a single X is still a character vector, so if you use `:case` statements in your event handler, you must match to `(1ρ'X')` or `(, 'X')`. You set the size as a two-element vector in `pwres`, showing height and width in the current scaling units of the control. Practically, however, only the length of the second element is useful. The display should be no higher than your control.

When there is at least one callback field defined, this event is triggered whenever the format property is set and when the control is opened. (You may see this event twice for each callback field that is defined when the control is opened.) You should be sure to define the onFormatQuery event handler before you set the

format property with any callback fields. If no callback fields are defined in the format property, the event does not fire.

onFormatString

You use the onFormatString event handler to specify the contents of any callback fields. The FormatString event occurs once for each callback field defined in the format property; it occurs just before every Paint event for the control. Thus, it occurs when the value property changes, either by user action or under program control; when the user arrows from field to field within the control; and when the control gets or loses the focus, is moved, is resized, or is covered or uncovered. It also occurs when you query the text property of the control, although this does not trigger a Paint event. Your event handler should be prepared to specify the contents of every callback field under any set of circumstances.

When the event is triggered, `□warg` is a two-element, nested vector; the first element is a character vector holding the field identifier, for example 'XXX'. The second element is a seven element vector holding the current value property for the control. You set the contents of the callback field as a character vector in `□wres`. Any date and time fields that are defined in the format property get their values from the value property. Note that you cannot set the value in this event handler, because of recursions.

onUserKeyDown

You use the onUserKeyDown event handler to capture keystrokes in a callback field. This event handler is similar to the onKeyDown handler for any control that has it, including a DateTime control, but the UserKeyDown event fires only in a callback field. The contents of `□warg` and the behavior of `□wres` are therefore somewhat different.

When the focus is in a callback field and the user presses a key, `□warg` is a two-element nested vector. The first element is a character vector that identifies the field, for example 'XXXX'. The second element is a three-element numeric vector; the first element contains the virtual key code for the key that was pressed, the second element contains the keyboard shift state, and the third element contains the toggle state of the CapsLock key when the event occurred.

The keyboard shift state is the sum of three elements, 1=Shift Key down, 2=Control Key down, and 4=Alt Key down. However, note that you will not get values 4 and 5. The underlying DateTime control processes Alt+ any key, and Alt+Shift+any key in a different manner than the other combinations, so these do not trigger this event. You can see Alt+Ctrl+any key or all three shift keys down+any key. The toggle state of the CapsLock key is on=1, off=0.

The initial value of `□wres` is the value property of the control. You can reset `□wres` or change any of the elements of `□wres` in your event handler. When the event handler returns, the system updates the value of the control; this in turn triggers the onFormatString event handler, which you can use to update all the callback fields.

Note: Due to an anomaly in the processing of the underlying control, if you change only the seventh element of `□wres`, the system may not update the value; in this case you must reset the value property to equal the changed `□wres` in your onUserKeyDown handler.

Examples:

The following examples show two formatted DateTime controls. The first one displays date and time fields formatted like a file system timestamp and two callback fields that indicate the relationship of the value to the current date. The second example shows how you might capture keystrokes in a callback field to use in conjunction with a DateTime value for an automatic timecard system.

```
'fmDT' □wi 'New' 'Form' ('where' 1 1 9 60)
'fmDT' □wi 'caption' 'DateTime Examples with Callback Fields'
dd←'fmDT.dtTS' □wi 'New' 'DateTime' ('where' 1 1 1.25 36) ('style' 2)
dd □wi 'onFormatQuery' '□wres+1 4' A Works for both fields
```

```
dd [wi 'format' " 'Set timestamp to:' yyyy '/' M '/' d h:mm tt X XX"
dd [wi 'onFormatString' 'ShowX'

'fmDT.bnNow' [wi 'New' 'Button' ('where' 1 38 1.25 6)
'fmDT.bnNow' [wi 'caption' 'Now' A Clicking this sets current timestamp
'fmDT.bnNow' [wi 'onClick' " 'fmDT.dtTS' [wi 'value' [ts"
```

This function specifies the display of the two callback fields. Note using ravel with a single X field identifier. Both these fields derive their values from the relationship of the date in the control to the current timestamp. The user can change any of the displayed date and time fields or drop down the calendar to select a date.

```
▽ ShowX
[1] when←(3↑[wi 'value'])≠(3↑[ts])
[2] :select ←[warg[1]
[3]   :case (, 'X')
[4]     :if (when≡0 0 0)
[5]       [wres←'today'
[6]     :elseif ([wi 'value'])[when1] > [ts[when1]
[7]       [wres←'future'
[8]     :else
[9]       [wres←'past'
[10]    :endif
[11]  :case 'XX'
[12]    :if (when≡0 0 0)
[13]      [wres←' '
[14]    :else
[15]      [wres←←1>(when/'year' 'month' 'day')
[16]    :endif
[17]  :end
```

This DateTime control displays the current time and provides a field for entry of an id number. When the id is entered correctly, the OK button allows the system to record the time of entry and the user.

```
'fmDT.lbl' [wi 'New' 'Label' ('where' 3 1 1 7)('caption' 'Check in/out')

[wsself←'fmDT.dtCkin' [wi 'New' 'DateTime' ('where' 4 1 1 36) ('style' 3)
[wi 'onFormatQuery' '[wres←1 4' A Size of callback field never changes
[wi 'format' 'MMM d, yyyy H:mm 'Enter your 4-digit ID slowly' XXXX'
[wi 'onFormatString' 'DisplayID'
[wi 'onUserKeyDown' 'ReadID'

'fmDT.bnOK' [wi 'New' 'Button' ('where' 4 38 1.25 6) ('caption' 'OK')
'fmDT.bnOK' [wi 'onClick' 'ProcessEntry'
```

The DisplayID function takes the numeric value of the user's ID and displays it as characters in the callback field. If the user attempts to enter too many characters, a visible rejection is displayed.

```
▽ DisplayID
[1] :if ctr=0
[2]   [wres←'----'
[3] :elseif ctr<5
[4]   [wres←⌈id
[5] :else
[6]   ctr←0 ⋄ id←0
[7]   [wres←'%%%'
[8] :end
```

The ReadID function converts the value of the virtual key code to the numeric value of its character for both keyboard and keypad numeric keys. If any other key is pressed during entry of the id number, the function ignores it by resetting the counter. Upon completion, [wres is set to the current timestamp, which resets the value of the DateTime control with the current time and the id number entered.

```

▽ ReadID
[1]  ctr←ctr+1
[2]  :if ctr<5
[3]      key←2 1>□warg
[4]      :select key
[5]          :caselist (47+110)
[6]          value←key-48
[7]          id←id+value×10*(4-ctr)
[8]          :caselist (95+110)
[9]          value←key-96
[10]         id←id+value×10*(4-ctr)
[11]         :else
[12]             ctr←ctr-1
[13]         :endselect
[14]     :endif
[15]     □wres←□ts

```

The `ProcessEntry` function resets the current time so the user cannot backdate his check-in time, records the user's ID and current time, and initializes the callback field for the next user.

```

▽ ProcessEntry
[1]  'fmDT.dtCkin' □wi 'value' (time←□ts)
[2]  idno←-4↑'fmDT.dtCkin' □wi 'text'
[3]  A 'Record person and time'
[4]  ctr←5
[5]  DisplayID

```

Edit



Prefix: ed

Summary:

A field that lets the user enter text from the keyboard.

Uses:

You can use an Edit control when it is necessary for the user to type input. Edit controls can hold a single field of data or multiple lines of text. You may want to validate the input from an Edit control (for example, in the form's `onClose` or in an `onExit` event handler), since the application provides no restrictions, except length, on the input.

Regular Properties:

autocomplete, border, children, class, color, data, ddeApplication, ddeItem, ddeMode, ddeTimeout, ddeTopic, def, deferexit, edge, enabled, events, extent, font, help, helpcontext, hwnd, imagescale, instance, keys, limit, links, methods, mode, modified, modifystop, name, noredraw, opened, order, place, pointer, prompt, properties, range, scale, scrollaccel, scrollmargin, selection, self, seltext, size, state, style, suppress, tabgroup, tabparent, tabstop, targetformats, text, theme, tooltip, tooltipenabled, tooltipstyle, tooltiptime, tooltipwidth, translate, visible, where, and *Δudp*.

autocomplete

If you set this property to a non-default value, as the user types, the Windows system will pop up a small window like a tooltip with choices that match the letters typed so far. If the user presses the Enter key, the entire string is inserted into the edit box as the text.

The default value is zero. Non-zero values allow you to specify various sources from which the candidate strings are taken. As with a style property, you can specify multiple sources by setting a value that is the sum of the individual values.

Note that this feature is not compatible with a multiline edit control that accepts the Enter key as a character. The autocomplete property intercepts the Enter key and inserts the suggested string, even if the string is empty.

The values for this property define the following items as sources for the suggested strings:

- 1 The file system and the rest of the shell
- 2 URLs in the User's History
- 4 URLs in the User's Recently Used List (this includes entries in Start/Run, for example)
- 16 The file system only
- 32 File system directories, UNC servers, and UNC server shares (but not files).

The values 1, 16, and 32 are variations on the same set, so they should not be used together. Thus, any odd value 17 or greater is improper, as is any value greater than 46.

range

A two-element integer vector that describes what lines of a vertically scrolling, multiline Edit control are visible in the edit window. The first element is the origin—one line number of the top row; the second element is the number of visible lines (which can vary with font size). You can set only the first element; if you change the second, you change the size of the control to accommodate the setting.

selection

A two-element integer vector that contains the number of characters before the first selected character and the number of characters of highlighted text. You can specify -1 for the first element to place the cursor at the end of the field. You can specify -1 for the second element; that selects everything after the insertion point to the end of the field

seltext

A character vector of the selected text. You can query or replace text using this property as well as `text`.

style

An integer vector that contains selected values from the following list or an integer scalar that is the sum of the selected values. The initial value of `style` for any newly created control object is 0; if you want multiple lines in the control, you must select 4. You can select any meaningful combination of other values for `style`; many of them require that you have a multiline control.

Values for the Edit Class style Property

Value	Description
0	One-line; left-justified text (the default).
1	Centered text (only if 4 is also coded on some systems).
2	Right justified text (only if 4 is also coded on some systems).
4	Multiline.
8	Inhibit automatic vertical scrolling (only useful if 4 also coded).
16	Vertical scroll bar (only useful if 4 also coded).
32	Inhibit automatic horizontal scrolling (causes word wrapping with 4).
64	Horizontal scroll bar.
128	Password (displays chars as ★).
256	Lowercase (cannot be used with 512).

512	Uppercase (cannot be used with 256).
1024	Keep selection highlighted when focus moves away from field.
2048	APL entry field (you must also set the font property).
4096	Read only.
8192	Allow Enter character (only if 4 is also coded).
16384	Do not select text when tabbing back to control. (Often used with 4 and 1024.)
524288	Read only with the Paste menu in the context menu disabled (only with 4096).

text

A character vector that contains the contents of the edit box.

Event-handler Properties:

onAction, onChange, onClose, onDdeConnect, onDdeDisconnect, onDelete, onDestroy, onDragEnter, onDragLeave, onDragover, onDrop, onExit, onExitError, onFocus, onHelp, onHide, onKeyDown, onKeyPress, onKeyUp, onLimit, onMemory, onModified, onMouseDouble, onMouseDown, onMouseDrag, onMouseEnter, onMouseLeave, onMouseMove, onMouseUp, onMove, onOpen, onPaint, onReopen, onResize, onScroll, onSend, onShow, and onUnfocus.

Methods:

CharToLine, Close, Copy, Create, Cut, DdeExecute, DdePoke, DdeRequest, Defer, Delete, Draw, Event, Exec, Focus, Help, Hide, LineToChar, Modify, New, Open, Paint, Paste, Popup, Ref, Resize, Send, Set, SetLinks, and Show.

Parentage:

Edit boxes are most often children of a Form, Frame, Page, or CommandBar. They may also be children of some control objects: Button, Check, Combo, Edit, Label, List, Option, Picture, Scroll, Spinner, Status, and Toolbox. The system allows most classes as children; it specifically prohibits Form, MDIForm, CommandBar, CommandButton, Page, and Printer.

Examples:

This is an example of creating Edit control objects.

```
'fmSample.edName' □wi 'New' 'Edit' ('where' 5 10 1.5 20)
'fmSample.edText' □wi 'New' 'Edit'
'fmSample.edText' □wi 'where' 8 10 8 20
'fmSample.edText' □wi 'style' 4 16 8192
'fmSample.edText' □wi 'text' 'sampleText'
```

theme

The theme property of a Edit class controls the visual styling theme used for drawing the control. The default value is an empty character vector (' '), which uses the new visual styling for the control. To suppress the new visual styling for the control, set thememode to "~".

Form



Prefix: fm

Summary:

A rectangular framed area upon which you can define controls. A form is a window that is part of the

Windows desktop. It does not need to fill the entire screen; you can allow it to be resized, minimized, and maximized. You can use it as a dialog box, editing window, message box, graphics window, etc.

Uses:

An APL Form is an object that carries with it:

- properties and event handlers for the Form object
- control objects (such as Button, List, Menu, or Edit objects) that are children of the Form
- properties and event-handler properties for these controls

Once you have created them, the Form and all of its associated objects are treated as a unit for :

- creating, and editing objects in a Make function
- displaying the form and executing the processes of your application
- storing and loading definitions from template variables in the workspace.

For each of these purposes, you need only to name the Form, and the Form and all its associated objects go with it.

Regular Properties:

bars, border, caption, children, class, color, data, ddeTopic, def, edge, enabled, events, extent, font, help, helpcontext, hwnd, icon, imagescale, instance, keys, limitwhere, links, methods, mode, modified, modifystop, name, noredraw, opened, place, pointer, prompt, properties, scale, scrollaccel, scrollmargin, self, size, state, style, suppress, targetformats, theme, tooltipenabled, tooltiptime, tooltipwidth, translate, value, visible, where, and $\Delta u d p$.

border

An integer vector that contains selected values or an integer scalar that is the sum of the selected values. The table below shows the possible values; use one of the primary values plus any of the add-on values for the menu bar icons. The default value of border is 1 1 4, a sizable border with all three icons.

Values for a Form Class border Property

Value	Description
Primary Value: Choose only one value	
0	No border
1	Thin border
2	Thick (sizable) border
3	Modal border (obsolete; now same as 1.)
Add-on Values: Choose none, any, or all of the values	
16	System menu and Close button
32	Minimize button (only with 16)
64	Maximize button (only with 16)
128	Title bar without buttons (and possibly captionless)
256	Context help "?" button (with 16, but not with 32, 64 or 512)
512	Windows tool palette (not with 32, 64, or 256)

If you include 16, the system places and allows items in a form's title bar that enable the user to manipulate it. You must have the system menu and the Close button to get the minimize and maximize buttons; if you choose one of the buttons, both appear but the other is disabled. You must have the system menu and the Close button, but not the minimize/maximize buttons nor the palette to get the context help button.

If the form's `border` property does not include 16 or 128 in its sum and you set the `caption` to an empty vector, then the form does not have a title bar, in which case the user cannot drag it, although it still can be resizable. If you want a form with no border, `caption` must be empty; if you want a form with a blank title bar and a thin border, use border value 128 and an empty caption. You make a form modal by using the `Wait` method, but you can no longer have the same visual effect.

If you want to provide mouse-accessible context sensitive help for the children of a form, you can replace the minimize and maximize buttons with the context help button, which has a question mark as its caption. If a user clicks on this button, the cursor changes to a pointer augmented by a large question mark. When the user next clicks on a control on the form, it can trigger an `onHelp` event handler, either on that object or an object in its parentage chain. You do not have to use the context help button for this facility; pressing the F1 key triggers the same event. See the description of the `onHelp` event handler for more information.

You can create a tool palette by adding the value 512 to the `border` property of a form. You cannot combine this with minimize/maximize buttons. If you include 16 in the border property, you get a Close button but no system menu. A tool palette does not appear in the taskbar or in the Alt+Tab task list, unless you include 32 in the `style` property. The tool palette's appearance also differs from a standard form in that the title bar and its caption are smaller in height than normal. Note that this functionality came before command bars. You can create a richer toolbar using a floating command bar and associated command buttons.

caption

The caption text for a form appears in the title bar; it also appears with the icon when the form is minimized. The initial value of `caption` is the name you assign the form when you create it. Once you create the form, however, the `name` property and `caption` property are independent. If you set `caption` or change `name`, the value of the other does not change.

def

The `def` property for a form returns a definition of the form and its properties, and all of its children and their properties, in a precompiled format. You can re-create the form from this definition at a later time or use it as a template to create an identical copy. See the “Using Precompiled Object Definitions” section in the Programming the Windows Interface: Introduction chapter in this manual for more information.

visible

The `visible` property is an integer; it controls not only whether the user can see the form, but how much of the screen the form occupies (or will occupy when made visible).

Values for the Form Class `visible` Property

Value	Description
1	Shown normally
2	Minimized (reduced to an icon)
3	Maximized (full screen)
-1	Hidden
-2	Minimized and hidden
-3	Maximized and hidden

value

The `value` property of a Form object normally has no special significance. However, when you exit from the `Wait` method, the system returns whatever you specify for the `value` property of a Form as the explicit result of `Wait`. You can define `value`, for example, as a nested array that contains the state of the objects on the Form when you close it.

style

The **style** property for a Form object is an integer scalar; its initial value is 0. The style property allows different ways for a form to interact with the Wait method. Note that these styles are intended for forms as top-level objects. For forms that are children of an MDI form, values other than the default are pointless.

Values for the Form Class style Property

Value	Description
0	Default form.
1	Signal Wait event before Showing form.
2	Make form independent of any other form that is Waiting.
4	Hide form before Waiting.
8	Do not disable for Wait on a secondary form
16	Always on top.
32	Force icon in taskbar even if form is a tool palette (border 5 1 2).

Ordinarily, when you invoke Wait, the system executes the onOpen and onResize event handlers before it executes the onWait event handler. If you set **style** to 1, the system executes the onWait event handler before it opens the form.

Similarly, when you Hide a form on which you invoked the Wait method, that ordinarily ends the wait loop. If you set **style** to 4, the system hides the form before waiting. This allows you to have a Wait on a hidden form.

When you have a form open and you open another form on which you invoke Wait, the default behavior is that the secondary form blocks input to the primary form until its wait loop ends. If you set **style** (on the primary form) to 8, the primary form is not disabled by a Wait on a secondary form.

If you set **style** to 2, the form is independent of any other form that is waiting.

If you have 16 in the **style** property, the form is always on top. If you place 32 in the **style** property, the system forces an icon in the task bar, even if the form is a tool palette (border 5 1 2). This value is meaningful only when you have border set to 5 1 2, since an icon automatically appears in the task bar otherwise.

place

You can use the where, extent, and size properties for forms just as you do for controls. In addition, you can use the **place** property to control its location. When a form is minimized or maximized, you can set the **place** property to specify the location and size you want the form to have when it is restored. This is primarily useful for forms that are children of an MDI form.

In addition to numeric values when you set this property, you can use the equal character ('=') to indicate the element should not change, or you can use the question mark ('?') to indicate the system should use the default setting.

When a form is closed, or when it is in a restored state, **place** is identical to **where**; setting one changes the other. The interaction with the size and extent properties for a closed form is the same as for closed controls. When you reference the **place** property, it always returns a four-element vector. It may show the equal character or question mark.

Warning: In future systems, the **place** property may have more than four elements. If you reference it with the intention of using the values in your program, you should use a statement like

```
(r c h w) ← 4 ↑ 'fmMain' □wi 'place' rather than the simple statement
(r c h w) ← 'fmMain' □wi 'place'
```

Additional values would be catenated to the existing vector, so it is strongly suggested that you avoid statements like `(h w) ← -2 ↑ 'fmMain' □wi 'place'`

theme

The theme property of a Form class controls the visual styling theme used for drawing the control. The default value is an empty character vector (' '), which uses the new visual styling for the control. To suppress the new visual styling for the control, set thememode to "~".

Event-handler Properties:

onAction, onClose, onDdeConnect, onDdeDisconnect, onDdeExecute, onDelete, onDestroy, onDragEnter, onDragLeave, onDragover, onDrop, onDropDown, onExitError, onFocus, onHelp, onHide, onKeyDown, onKeyPress, onKeyUp, onModified, onMouseDouble, onMouseDown, onMouseDrag, onMouseEnter, onMouseLeave, onMouseMove, onMouseUp, onMove, onOpen, onPaint, onReopen, onResize, onSend, onShow, onUnfocus, and onWait.

Methods:

Close, ContextHelpMode, Create, Defer, Delete, Draw, Event, Exec, Focus, Help, Hide, Modify, New, Open, Paint, Popup, Ref, Resize, Send, Set, SetLinks, Show, and Wait.

Parentage:

Forms can be top-level objects or they can be children of an MDIForm. They can contain Button, Check, Combo, CommandBar, DateTime, Edit, Frame, Imagelist, Label, List, Listview, Media, Menu, Option, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, and Tree objects.

Example:

This example creates a blank form in the upper left corner of the screen:
'fmSample' □wi 'New' 'Form' ('where' 2 2 16 40)

Frame



Prefix: f

Summary:

A border that encloses a group of controls.

Uses:

A Frame is a graphical element you use to enclose related controls. Users have little interaction with frames because a frame cannot have the input focus. If a user types the access character (denoted by an & in the Frame caption), the input focus moves to the next control that can receive the input focus. Normally this is an object within the frame; for example, an option button that is selected.

You can define control objects as children of a frame by using the name of the frame as part of the name of the object. For example, if you name a frame `fmMain.fSize` and you name an option `fmMain.fSize.opLarge`, the frame contains the option. The coordinates of the where property for the option button are relative to the upper left-corner of the frame's client area and in scale units of the frame.

However, if you name the option `fmMain.opLarge`, leaving the frame name the same, the frame does not contain the option. The coordinates of the `where` property for the option button are relative to the upper left-corner of the form's client area and in scale units of the form. You can still locate the option button physically on the frame, by setting its `where` property to be within the area of the frame.

Regular Properties:

bars, border, caption, children, class, color, data, def, deferexit, edge, enabled, events, extent, font, help, helpcontext, hwnd, imagescale, instance, keys, links, methods, mode, modified, modifystop, name, noredraw, opened, order, place, pointer, prompt, properties, scale, scrollaccel, scrollmargin, self, size, state, style, suppress, tabgroup, tabparent, tabstop, targetformats, theme, tooltip, tooltipenabled, tooltipstyle, tooltiptime, tooltipwidth, translate, visible, where, and `Δudp`.

style

The effects that the `style` property has depend on whether the three-dimensional control setting (`CTL3D`) in the `[Config]` section of the application's `.INI` file is set to 1 (the default) or zero (no 3D). A frame with style 0 is the standard frame with a caption. Other styles do not display a caption.

Values for the Frame Class style Property

style	Visual Effect without 3D	Visual Effect with 3D
0	Frame appears as black line	Frame appears etched into surface.
1	Black line rectangular frame	Inset 3D rectangle
2	Gray line rectangular frame	Flush 3D (etched) rectangle
3	White line rectangular frame	Outset 3D rectangle
4	Solid black rectangle	Inset 3D rectangle
5	Solid gray rectangle	Flush 3D (etched) rectangle
6	Solid white rectangle	Outset 3D rectangle
7	Transparent	Transparent

theme

The `theme` property of a `Frame` class controls the visual styling theme used for drawing the control. The default value is an empty character vector (`' '`), which uses the new visual styling for the control. To suppress the new visual styling for the control, set `thememode` to `"~"`.

Event-handler Properties:

`onAction`, `onClose`, `onDelete`, `onDestroy`, `onDragEnter`, `onDragLeave`, `onDragover`, `onDrop`, `onExit`, `onExitError`, `onHelp`, `onHide`, `onModified`, `onMouseDown`, `onMouseDown`, `onMouseDown`, `onMouseEnter`, `onMouseLeave`, `onMouseMove`, `onMouseUp`, `onMove`, `onOpen`, `onPaint`, `onReopen`, `onResize`, `onSend`, and `onShow`.

Note: Although a `Frame` cannot receive the focus, it can receive mouse events. For a `Frame` to do so, you must set the `pointer` property to a nonzero value or set one of its `onMouseXx` event handlers. If none are set, the `Frame` is transparent to all mouse events. This can affect your application if you have controls that are siblings of the frame overlapping or within the area of a frame.

Methods:

`Close`, `Create`, `Defer`, `Delete`, `Draw`, `Event`, `Exec`, `Help`, `Hide`, `Modify`, `New`, `Open`, `Paint`, `Popup`, `Ref`, `Resize`, `Send`, `Set`, `SetLinks`, and `Show`.

Parentage:

Frames are most often children of a `Form`, `MDIForm`, another `Frame`, `Page`, or `CommandBar`. They may also be children of some control objects: `Button`, `Check`, `Combo`, `Edit`, `Label`, `List`, `Option`, `Picture`, `Scroll`, `Spinner`, `Status`, and `Toolbox`. They can contain `Button`, `Check`, `Combo`, `CommandBar`, `DateTime`, `Edit`, other `Frame`, `Imagelist`, `Label`, `List`, `Listview`, `Media`, `Option`, `Picture`, `Progress`, `RichEdit`, `Scroll`, `Selector`, `Spinner`, `Status`,

Timer, Toolbox, Trackbar, and Tree objects. A Frame can also contain a Menu object, not as a visible menu bar, but for the purpose of assigning a shortcut key to the frame itself rather than to individual controls within the frame.

Example:

This example shows how you create a frame labeled Style. The ampersand (&) in the value specified for the caption property causes the S character to be the access character for frame's caption.

```
'fmSample.fSize' □wi 'New' 'Frame' ('where' .5 .5 8 16) ('caption' '&Style')
```

Imagelist

Prefix: i

Summary:

An Imagelist is a container holding a collection of same-sized (usually small) images. You do not display an imagelist directly; you display the images as part of Tree, Listview, Combo, Status, and Page objects and on Buttons, CommandButtons, and Menus. You can use bitmaps, icons, or cursors as your images.

Uses:

An imagelist is a container of images. All the images you place in an imagelist must be the same height. This is particularly important for bitmaps; the system can scale icons and cursors to conform to a specified size.

If you are using a large number of bitmaps in your application, you can improve performance by tiling them into one bitmap. If you have N bitmaps that are H pixels high by W pixels wide, create a tiled bitmap that is H pixels high and (W×N) pixels wide. Then, define the `imagesize` property to be H by W before you load the bitmap into the Imagelist. Using this technique, the system loads only one bitmap, but treats it as N individual images, which you can index separately. You can have multiple bitmaps in one imagelist that are wider than your specified width and hence have more than one image each.

You can specify an imagelist as being masked or not masked. If you define a masked imagelist, the images can have a transparent background so that whatever is underneath the image shows through those pixels that are the background color. For bitmaps, you define the "color" that the system makes transparent. The default value is Windows background gray (192 192 192).

Within an imagelist, you can also designate up to fifteen images as being overlays. Overlay images, which must be masked to be effective, are drawn by the system onto the same area as a regular image to create a composite image. You can use overlay images in a Listview or Tree object.

Regular Properties:

children, class, colordepth, data, def, events, himage, imagealloc, imagecount, imagenames, imagescale, imagesize, instance, links, maskcolor, methods, modified, modifystop, name, opened, overlays, properties, self, state, style, suppress, and *Δudp*.

imagesize

A two-element numeric vector that defines the height and width of each image you want the imagelist to hold for display. The height of each bitmap you add to the imagelist must match the first element of this property; if it does not, the system triggers the `onResource` event handler of the system object, which may occur when you specify the bitmap, when you open the imagelist, or when you reset the `imagecount` or `imagesize` properties. You can set `□wres←1` to have the system signal a `□WI RESOURCE ERROR`. (See the description of the `onResource` event handler in the Windows Interface: Event-Handlers chapter in this manual.)

When you specify `imagesize`, the system removes all images that are currently in the imagelist. The system reinitializes the object, and then checks the `imagenames` property. If this property is not empty, the system repopulates the imagelist with the images specified in `imagenames`. If you do not set `imagesize`, the first image you add to an imagelist establishes the size. If you want to use vertical slices of one bitmap for multiple images, you must set `imagesize` before you add the first bitmap.

When you add a bitmap to the imagelist, the system calculates the number of images it can take horizontally and sets the index values for those images. If you subsequently add another bitmap, its first index is one higher than the index for the last image of the previous bitmap. Using a bitmap whose width is not an even multiple of the horizontal value of `imagesize` is not recommended; if you do, the system may or may not try to use the remainder as the first part of the next image.

If you add an icon or a cursor to an imagelist when `imagesize` is not specified, the system uses its default icon or cursor size to specify `imagesize`. If the (possibly multi-sized) icon or cursor has an image of this size, the system uses it. If the icon or cursor is not the default size, the system takes an image near the desired size and stretches or shrinks it to fit. If you add an icon or a cursor to an imagelist after `imagesize` is specified, the system stretches or shrinks the image to fit the specified size.

imagealloc

A two-element numeric vector that gives the number of spaces for images in the object. The first element is the number of images for which you want to have memory allocated; the second is the incremental number of images for which to allocate memory if you add an image when there is no room for it. The default values are 1 5. If you specify `imagealloc` after the imagelist is populated, the system removes the existing images and reinitializes the imagelist. The system then looks at the `imagenames` property to populate the list.

imagenames

A vector of character vectors specifying image sources. When you specify this property and the object is open, or when the object is opened and `imagenames` is not empty, the system removes any existing images and loads the images specified in this property into the imagelist. The `imagenames` property remains intact and is used whenever necessary to populate the imagelist until the first time the `AddImages` method is used.

Once you use `AddImages`, the `imagenames` property is set to an empty vector. When `imagenames` is empty, existing images remain and the system adds additional images at the end of the imagelist. You can clear the imagelist by resetting the `imagealloc` or `imagesize` properties.

There are a number of ways to specify the image sources in `imagenames`; the simplest is to use file names with the appropriate `.bmp`, `.ico`, `.cur`, or `.ani` suffix. See the technical note at the end of this description for the alternatives in specifying image sources. If you want to specify images in the `def` property of an `Imagelist` or in a `Make` function, you must define the `imagenames` property. The images themselves are not included.

imagecount (read-only)

An integer that gives the number of images in the object. If the object is closed, this is an empty vector.

style

If you set `style` to 1, you define a nonmasked imagelist, which does not allow transparent pixels in an image. The default style of 0 defines a masked imagelist. A masked list allows you to have one transparent color, so that the background shows through the image when the system places it on the screen. There are several ways to specify the transparent color.

For a bitmap, you can specify the transparent color as a second element of the argument to the `AddImages` method; for example,

```
'iBig' □wi 'AddImages' (c'c:\windows\256color.bmp' ('color' 0 255 0))
```


You can specify a source for an image as a pair of bitmaps. The system interprets the second bitmap as a Boolean mask array of the same dimensions as the first bitmap, with 1s where you want the first bitmap to be transparent.

You can specify one color in the `maskcolor` property; the system draws bitmaps with that color transparent, if you have not specified color in either of the ways explained above. Icons and cursors can be created with transparent pixels.

maskcolor

This property defines the color that is drawn, by default, as transparent for bitmaps you add to the imagelist. It does not apply to images that already exist in the imagelist when you specify `maskcolor`. This value also does not apply when you add a bitmap that has a mask associated with it or when you explicitly specify a mask color for a bitmap.

You can specify this value as a composite color using three values of 0 to 255 for each of Red, Green, and Blue, or you can use a scalar formed by combining the three numbers using the Base Value primitive function: `256 * blue, green, red`. (Note the reversed order of the colors.)

The default value is `192 192 192`, which is Windows default button-face gray. You can also use negative integers to specify the colors defined for certain Windows elements by the user. These, of course, will vary from user to user. See the description of the color property in the Windows Interface: Properties chapter in this manual for a list of these values and elements.

overlays

An integer vector with a maximum length of fifteen elements. You specify the indices to those images in the imagelist you want to use as overlays. When you want to use an overlaid image in a Tree or Listview object, you specify the index value between 1 and 15 in the `overlays` vector rather than the direct index into the imagelist.

colordepth

An integer scalar whose valid values are: `-1 0 4 8 16 24 32`. The `colordepth` property controls the number of colors and the palette of colors used to represent images in the imagelist. The default value is zero; this specifies that the system use the default for the underlying Imagelist control, which is usually a 4-bit-per-pixel, device-independent bitmap. However, it can sometimes be a device-dependent bitmap. You can use the value `-1` to force a device-dependent bitmap. This value is recommended when you are using the imagelist only for controls on the screen; that is Listview, Tree, Button. Using this value can result in a noticeable improvement in performance over using a device-independent depth.

If you are going to draw images on a Printer object, you should use the number that represents the highest color depth of any image in the imagelist. These values (4 through 32) designate the color depth in bits per pixel for a device-independent bitmap.

Setting the `colordepth` property reinitializes the Imagelist; so if you have used the `AddImages` method, you must reconstruct the imagelist. If the images are listed in the `imagenames` property, the imagelist is automatically reconstructed with the changed depth of color.

Note: The ability to display graphics, bitmaps for example, with "bright" colors depends on the combination of the color depth of the video drivers and the support software. Color depth is basically the number of bits per pixel that describe the color of the image.

The Imagelist object was originally designed for graphics such as icons and low-color, small bitmaps. As resolution and color depth on systems have grown, bitmaps tend to be larger and are generated in multiple levels and with greater color depths. When systems with low color depth desktops display such bitmaps, the images appear to be washed out or shadows of the intended image.

You can perhaps use the `colordepth` property to alleviate this problem somewhat. If you do not know what value to use, setting it to `-1` produces the best results; zero is the default, however, to maintain existing behavior for those who do not wish to change. Setting this property to a positive value is most effective when the desktop is set for 16 million colors (2×24) or greater. Setting it is not very effective at all when the desktop is set to 256 colors or below.

When you develop applications that may run on different systems, some of which may be low resolution, you can use two sets of bitmaps for your images. You can query the resolution of the desktop by doing the following:

```
hdc = hwndcall 'GetDC' 'HWND_Desktop'  
hwndcall 'GetDeviceCaps' hdc 'bitspixel'  
16      A indicates 16 bits per pixel  
hwndcall 'ReleaseDC' 'HWND_Desktop' hdc
```

You can detect when a user changes the desktop resolution by using the `onSystemChange` event handler on the system object.

himage

The handle of the underlying Windows control; you use this for low-level programming.

Event-handler Properties:

`onAction`, `onClose`, `onDelete`, `onModified`, `onOpen`, `onReopen`, and `onSend`.

Methods:

`AddImages`, `Close`, `Create`, `Defer`, `Delete`, `Event`, `Exec`, `Modify`, `New`, `Open`, `Ref`, `Send`, `Set`, and `SetLinks`.

AddImages

You can use this method to populate the imagelist or add images to it after it is initialized. The imagelist must be open when you use `AddImages`. The argument can be any of the forms that are valid for the `imagenames` property. In addition, you can supply a singleton integer bitmap handle that you obtain through low-level Windows calls or through the `Draw` method using the `?Bitmap` operation. Note that using `AddImages` copies the bitmap, so you can free the bitmap, using the `wfree` function from the WINDOWS workspace immediately. You do not need to wait until you have finished using the `Imagelist` object.

The result of using `AddImages` is the number of images previously in the `Imagelist`. Note that if you want to specify images in the `def` property of an `Imagelist` or in a `Make` function, you must define the `imagenames` property. The images themselves are not included.

Parentage:

`Imagelists` can be children of any object except a `Timer`, `CommandButton`, or another `Imagelist`. They can contain only `Timers`.

Technical Note: You can specify the sources or images you want to use to populate an imagelist in any of several ways. All of these specifications are character vectors, which can be nested for specifying multiple sources. These syntax specifications apply to both the `imagenames` property of an imagelist and arguments to the `AddImages` method.

You can use the full path to a file name with the extension that corresponds to its image type, bitmap, icon, cursor or animated cursor. (The `.ani` file uses only the first image.) For example,

```
'c:\windows\argyle.bmp'  
'c:\aplwin\aplw.ico'  
'd:\data\mycursor.cur'  
'c:\windows\cursors.ani'
```

You can also use just the file name and extension, for example, 'argyle.bmp'; the system looks in each directory specified in the `RESOURCES=` file entry in the `[CONFIG]` section of the `APLW.INI` file. This entry is a list of directory names in the same format as the `DOS PATH` command. You can have multiple entries separated by semicolons (;). If the system does not find the module in any directory listed, it looks in the current directory. You can also specify a path that is relative to the current directory, such as 'images\developers.bmp'.

You can specify an image resource; in this context, "resource" means an image file compiled with a resource compiler and bound into a Windows executable file (.EXE) or a 32-bit Dynamic Link Library (.DLL). You specify the type of image by using the greater than sign (>) followed by the three letters denoting the image type and a space, for example, '>BMP '. You then must specify the resource by a resource name or a resource number, optionally followed by a space and a module name. The following are examples of valid resource specifications.

```
'>BMP dagger'  
'>ICO 3'  
'>CUR 4 d:\zilde.exe'  
'>BMP foomap shell32.dll'
```

If you do not specify a module name, the system looks first in the `Resource DLL=` file entry in the `[CONFIG]` section of the `APLW.INI` file to identify the module containing the resource. You can specify only one module in this entry. If the system does not find the resource there, it looks in the application's executable file (`APLW.EXE` or whatever you have renamed the runtime system).

If you specify a module name, you can specify a full path or just a module name. If you specify a full path, the system looks only in that directory. If you specify just a module name, the system looks in the directory from which the application was loaded, the current directory, the Windows system directory, and finally in the directories specified in the `PATH` environment variable. Once the system finds the specified module, it does not look any further, whether or not the resource specified is present.

In the first two examples, the resource designation, the name or numeral, identifies a specific resource in an implied file; a bitmap named dagger or the third icon. In the last two examples, the resource designation, either a name or a numeral, identifies a specific resource in the file whose name follows; the fourth cursor in the executable file named zilde, or a bitmap named foomap in the dynamic link library named shell32.dll.

For masked images, you can specify bitmaps in pairs, where the second bitmap masks the first.

```
'iForest' □wi 'AddImages' ('treeimgs.bmp' 'treemsk.bmp')  
↳ ('buttons.bmp' 'btnmask.bmp')
```

Note that if you specify only one pair, you must enclose it to distinguish the pair from two individual resources; for example,

```
'iForest' □wi 'AddImages' (< 'treeimgs.bmp' 'treemsk.bmp')
```

You can also specify masked bitmaps with a specific color value the system draws as transparent. The following designations allow the background to show through white pixels:

```
'iB' □wi 'imagenames' (<'buttons.bmp' ('color' 255 255 255))  
'iB' □wi 'imagenames' (<'buttons.bmp' ('color' 16777215))
```

Label

Sample

Prefix: l

Summary:

An output-only text field that defines or separates other controls or displays information for any purpose.

Uses:

You can use a label on a form to describe, categorize, or explain another control or a section of the form.

You can place a label anywhere on a form. If it relates to other controls, it provides its information by virtue of its placement relative to those controls. In this regard, a `Label` differs from a `Frame` caption, which imparts its information relative to the visible border of the `Frame`.

You can also create a hidden `Label` on a `Form`, for example, as an item in the tab order sequence to separate two groups of `Option` or `Check` controls. Without a nongrouped object between them - that is, an object of a class other than `Option` or `Check` - they would be members of the same group. You may find this useful as an alternative to using the `tabgroup` property.

You can also use a `Label` on a form to hold text for a Dynamic Data Exchange (DDE) with another application. See the Using Dynamic Data Exchange chapter and the descriptions of the DDE-related properties, event handlers, and methods in this manual.

`Label` controls cannot receive the focus. If a user types the access character (denoted by the `&` in the `Label`), the input focus moves to the next control that can receive the input focus after the `Label` in tab order.

However, a `Label` can receive mouse events. You must set the `pointer` property to a nonzero value or set one of the `onMouseXx` event handlers for a `Label` to recognize the mouse events. If none are set, the `Label` is transparent to all mouse events. This can affect your application if you have sibling controls overlapping or within the area of a `Label`.

Regular Properties:

`border`, `caption`, `children`, `class`, `color`, `data`, `ddeApplication`, `ddeltem`, `ddeMode`, `ddeTimeout`, `ddeTopic`, `def`, `deferexit`, `edge`, `enabled`, `events`, `extent`, `font`, `help`, `helpcontext`, `hwnd`, `imagescale`, `instance`, `keys`, `links`, `methods`, `mode`, `modified`, `modifystop`, `name`, `noredraw`, `opened`, `order`, `place`, `pointer`, `prompt`, `properties`, `scale`, `self`, `size`, `state`, `style`, `suppress`, `tabgroup`, `tabparent`, `tabstop`, `theme`, `tooltip`, `tooltipenabled`, `tooltipstyle`, `tooltiptime`, `tooltipwidth`, `translate`, `visible`, `where`, and `Δudp`.

caption

The caption text for a label is what appears on the form. The initial value of `caption` is the name you assign the object when you create it. Once you create the object, however, the `name` and `caption` properties are independent. If you set `caption` or change `name`, the value of the other does not change.

The text you place in the `caption` property can appear on multiple lines. You must set the `size` or `where` property to the width that allows the amount of text you want on each line. Text that is longer than the width allows automatically wraps to the next line (except that an individual word does not break). You must set the `height` to the desired number of lines.

style

An integer vector that contains selected values or an integer scalar that is the sum of the selected values. The table below shows the possible values; use one of the primary values plus either or both of the add-on values. The initial value of `style` for any newly created control object is 0.

Values for the Label Class style Property

Value	Description
Primary Value: Choose only one value	
0	Left justified.
1	Centered.
2	Right justified.
Add-on Values: Choose none, any, or all of the values	
4	Inhibit the effect of an access key (do not underline character following &). (See note 1.)
8	Inhibit word wrap (only with 0).
32	Display ellipsis at the end of the visible caption; (not with 32). (See note 2.)
64	Display ellipsis in the middle of the visible caption that has the form of a path; (not with 64). (See note 2.)

Note 1: Normally, an ampersand in the caption property causes the next character of the caption to be the access character for the label. When the user types the access key, it moves the focus to the control whose order property value immediately follows the label's order property value. If you set style to inhibit the effect of the access key, and you include an ampersand in the caption, the ampersand appears on the screen as a character of the caption.

Note 2: There are two new mutually exclusive styles for the Label object that provide the Microsoft style ellipsis for captions that exceed the length of the label. If you set the style property to include 32, the ellipsis is placed at the end of the visible caption. If you set the style property to include 64, and the caption has the form of a path, the ellipsis is placed in the middle of the caption so the final (file) name is readable.

theme

The theme property of a Label class controls the visual styling theme used for drawing the control. The default value is an empty character vector (' '), which uses the new visual styling for the control. To suppress the new visual styling for the control, set thememode to "~".

Event-handler Properties:

onAction, onChange, onClose, onDdeConnect, onDdeDisconnect, onDelete, onDestroy, onExit, onExitError, onHelp, onHide, onModified, onMouseDown, onMouseDrag, onMouseEnter, onMouseLeave, onMouseMove, onMouseUp, onMove, onOpen, onPaint, onReopen, onResize, onSend, and onShow.

Methods:

Close, Create, DdeExecute, DdePoke, DdeRequest, Defer, Delete, Draw, Event, Exec, Help, Hide, Modify, New, Open, Paint, Popup, Ref, Resize, Send, Set, SetLinks, and Show.

Parentage:

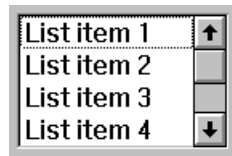
Labels are most often children of a Form, MDIForm, Frame, Page, or CommandBar. They may also be children of some control objects: Button, Check, Combo, Edit, Label, List, Option, Picture, Scroll, Spinner, Status, and Toolbox. The system allows most classes as children; it specifically prohibits Form, MDIForm, CommandBar, CommandButton, Page, and Printer.

Example:

This example shows how you create a label:

```
'fmSample.lTop' □wi 'New' 'Label'  
'fmSample.lTop' □wi 'where' .5 13 2 20  
'fmSample.lTop' □wi 'caption' 'Sample Form'  
'fmSample.lTop' □wi ('font' 'Arial' 3)
```

List



Prefix: ls

Summary:

A set of explicit choices from which a user can select; an input field restricted to program-defined choices.

Uses:

The List object allows the user to select one or more choices that the program offers. If there are more choices than the system can display in the space allowed, the List object provides for vertical scrolling through the choices. If a user presses a key when the focus is on a List, the system highlights the next item in the list that starts with that letter, numeral, or symbol. If the highlight is currently below or on the last item in the list that starts with the character, the system starts at the top of the list. If no item in the list begins with the typed character, the system does not move the highlight bar.

A List object is similar to a Combo box. In a List control, however, the user can select only from the choices provided. In a Combo box, the user can either select from a list or type text into the edit box. In Combo boxes, users are limited to one entry item at a time, whereas you can optionally allow users to select multiple items from a List. Note that a List object has a list property, which defines the set of choices presented to the user.

Regular Properties:

border, children, class, color, data, def, deferexit, edge, enabled, events, extent, font, help, helpcontext, hwnd, imagescale, instance, keys, limit, links, list, methods, mode, modified, modifystop, name, noredraw, opened, order, place, pointer, prompt, properties, range, scale, scrollaccel, scrollmargin, self, size, state, style, suppress, tabgroup, tabparent, tabstop, targetformats, theme, tooltip, tooltipenabled, tooltipstyle, tooltiptime, tooltipwidth, translate, value, visible, where, and Δudp .

list

A vector of enclosed character vectors, a character vector with `\t`-delimited strings, or a character matrix of choices in the list. When you reference it, it is returned as you specified it.

style

An integer vector that contains selected values or an integer scalar that is the sum of the selected values. The table below shows the possible values. Use either of the primary values to determine whether you allow the user more than one selection; add any of the add-on values to modify the display of the list. The initial value of style for any newly created control object is 0.

Values for the List Class style Property

Value	Description
Primary Value: Choose only one value	
0	Single selection
1	Multiple selections
Add-on Values: Choose none, any, or all of the values	
2	Allow the display of a vertical fraction of a line
16	Vertical scroll bar
32	Multiple columns (must also set limit)
64	Horizontal scroll bar (must also set limit)

Value	Description
128	Hide scroll bar if not necessary (short list)
256	Select multiple items independently (only with 1)

If you include 1 in the `style` property, the user can select all, some or none of the items in the list. You make multiple selections by holding the Shift key or Ctrl key down while clicking on items to select them. If the Shift key is down, the system selects a range of items between the first and last item clicked. If the Ctrl key is down, each item clicked is selected; the system does not unselect an item already highlighted unless it is clicked a second time. If the `style` property has 256 in its sum, the system ignores the Shift key and acts as though the Ctrl key is always down; that is, the selection of each item in the list is toggled independent of the others.

When the number of items of the list exceeds the height of the list box, you can choose to extend the list either vertically or horizontally (by using multiple columns). If you choose multiple columns, the system places as many items as fit vertically in each column and wraps the next item to another column. The list does not extend below the bottom of the list box. (Thus, it is pointless to code styles 16 and 32 together; although the vertical scroll bar appears, it is not functional.)

range

A two-element integer vector that describes what lines of a vertically scrolling, multiline List control are visible. The first element is the origin—one line number of the top row; the second element is the number of visible lines (which can vary with font size). You can set only the first element; if you change the second, you change the size of the control to accommodate the setting.

limit

You do not have to have a scroll bar. A user can highlight an item in the list and use the cursor keys to move either through a vertical list or across multiple columns. If you create multiple columns, with or without a scroll bar, you must set the `limit` property to specify the column width; the system uses as many columns as it takes to exhaust the list. If you choose a horizontal scroll bar in a single column format, you must also set the `limit` property to activate the scroll bar; `limit` specifies the width of the scrollable column. If `limit` is empty, there is no horizontal scrolling.

value

An integer vector that indicates the origin 1 index of the item or items selected. If `value` is zero or an empty vector, no item is selected. If the List object is a single selection List (default; the `style` property value is an even number), you can have 1 or 0 elements in the `value` property vector. If the List control object is a multiple selection List (the `style` property has 1 in its sum), you can select all, some, or none of the choices offered. Setting `value` to `-1` selects all items.

Event-handler Properties:

`onAction`, `onClick`, `onClose`, `onDbClick`, `onDelete`, `onDestroy`, `onDragEnter`, `onDragLeave`, `onDragover`, `onDrop`, `onExit`, `onExitError`, `onFocus`, `onHelp`, `onHide`, `onKeyDown`, `onKeyPress`, `onKeyUp`, `onMemory`, `onModified`, `onMouseDouble`, `onMouseDown`, `onMouseDown`, `onMouseDrag`, `onMouseEnter`, `onMouseLeave`, `onMouseMove`, `onMouseUp`, `onMove`, `onOpen`, `onPaint`, `onReopen`, `onResize`, `onSend`, `onShow`, and `onUnfocus`.

Methods:

`Close`, `Create`, `Defer`, `Delete`, `Draw`, `Event`, `Exec`, `Focus`, `Help`, `Hide`, `Modify`, `New`, `Open`, `Paint`, `Popup`, `Ref`, `Resize`, `Send`, `Set`, `SetLinks`, and `Show`.

Parentage:

Lists are most often children of a `Form`, `Frame`, `Page`, or `CommandBar`. They may also be children of some control objects: `Button`, `Check`, `Combo`, `Edit`, `Label`, `List`, `Option`, `Picture`, `Scroll`, `Spinner`, `Status`, and

Toolbox. The system allows most classes as children; it specifically prohibits Form, MDIForm, CommandBar, CommandButton, Page, and Printer.

Example:

This example shows how you create a List object and handle a user's selection; the items in the list correspond to rows in DataTable.

```
ls1 ← 'fmSample.lsList1' ⌵wi 'New' 'List'
ls1 ⌵wi 'where' 4 3 5 16
ls1 ⌵wi 'style' 16
ls1 ⌵wi 'list' (lsItm ← '/List item /,I2'⌵fmt 16)
ls1 ⌵wi 'onClick' "Process DataTable[(⌵wi 'value');]"
```

theme

The theme property of a List class controls the visual styling theme used for drawing the control. The default value is an empty character vector (' '), which uses the new visual styling for the control. To suppress the new visual styling for the control, set thememode to "~".

Listview

Prefix: lv

Summary:

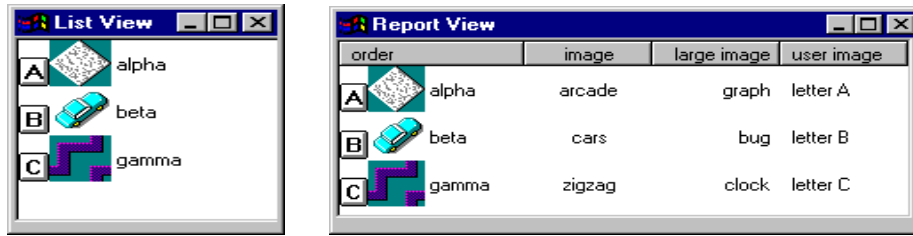
A Listview provides a generalized capability for displaying nonhierarchical information including images and their associated labels. You can make it possible for the user to manipulate this information in a number of ways.

Uses:

You can display the contents of a Listview object in four ways, using the viewmode property:

- You can display each item as an image with a label underneath the image; this is known as largeimage view, although you determine the size of the image you want to display. The images for largeimage view are contained in the Imagelist object specified in the imagelistlarge property.
- You can display each item as an image with a label beside the image; this is known as smallimage view, although you determine the size of the image you want to display. The images for smallimage view are contained in the Imagelist object specified in the imagelist property.
- You can display each item as an image with a label beside the image in one long list; this is known as list view. List view uses the same images as smallimage view; the images for list view are contained in the Imagelist object specified in the imagelist property, and each item uses the same index to those images as for smallimage view.
- You can display each item with multiple columns of information including headers; this is known as report view. You can use a Listview object without images to display multiple columns of text information effectively. If you include images, report view uses the same images as smallimage view and list view. The images for report view are contained in the Imagelist object specified in the imagelist property, and each item uses the same index to those images as for smallimage view.





You can add a second image to the left of the primary image; this is known as a user state image. These images are contained in the imagelist specified in the `imagelistuser` property. You specify a different index to these images than you do for the other imagelists. User state images are intended for use in list view and report view.

Regular Properties:

border, children, class, color, columndisplay, count, data, def, deferexit, dragimage, edge, enabled, events, extent, font, help, helpcontext, highlightcut, highlightdrop, highlightfocus, hwnd, infotipmask, imagelist, imagelistlarge, imagelistuser, instance, keys, labeledithwnd, links, list, methods, mode, modified, modifystop, name, noredraw, opened, order, place, pointer, poslargeimage, possmallimage, prompt, properties, roworigin, scale, scrollaccel, scrollmargin, searchstring, self, size, sortorder, sourceformats, state, style, suppress, tabgroup, tabparent, tabstop, targetformats, theme, tooltip, tooltipenabled, tooltipstyle, tooltiptime, tooltipwidth, translate, value, viewalign, viewmode, visible, where, and *Δudp*.

viewmode

A character vector containing one of the four views: 'largeimage' 'smallimage' 'list' or 'report'. The default value is 'report'.

viewalign

A character vector containing one of the three alignments for largeimage or smallimage view: 'none' 'top' or 'left'. The default is no alignment.

In largeimage and smallimage views, if you assign 'top', Windows aligns the first images with the top of the client area of the control and arranges them in columns; if there are more than will fit, the user must scroll vertically.

In largeimage and smallimage views, if you assign 'left', Windows aligns the first images with the left side of the client area of the control and arranges them in rows; if there are more than will fit, the user must scroll horizontally.

You can assign specific positions to images in either view using the `SetPos` method and either the `possmallimage` or `poslargeimage` property.

imagelistlarge

The name of the `Imagelist` object that contains the images you want to display in largeimage view. You use the origin one index to the images in this object to associate an image with a label.

imagelist

The name of the `Imagelist` object that contains the images you want to display in smallimage view, list view, or report view, and in the column headers. You use the origin one index to the images in this object to associate an image with a label.

The Listview control uses the same index to this imagelist as it does to the imagelist used for the largeimage view. If you want to use largeimage view as well as one of the three views that use imagelist, you should assign the corresponding images in the same order.

imagelistuser

The name of the `Imagelist` object that contains the user-state images you want to display to the left of the image from the `imagelist` property in list view or report view. If you switch or allow users to switch among view modes, you should reset the `imagelistuser` property to an empty character vector when `largeimage` and `smallimage` views are selected. Particularly in `smallimage` view, the underlying control does not handle user state images well.

Due to a Windows limitation, you cannot index more than 15 images in an `imagelist` being used for user state images. In APL, using an index greater than 15 results in a `DOMAIN ERROR`.

By setting the `style` property to include `65536`, you can have the system display check boxes in place of user state images. See the "Using Check Boxes in a Listview" section below for details of the interaction between that style and this property.

list

The `list` property contains or references all the visible information that the system displays in a `Listview` object. This property is a nested matrix containing one row for each entry in the list. You can have a variable number of columns; these columns can contain the actual information or they can contain placeholders that allow you to construct the information at the time the entry is about to be displayed using the `onGetVirtualData` event handler. A row of the `list` is organized as follows:

RowID	Images	Cell[1]	Cell[2]		Cell[n]

RowID

Unique positive integer item identifier. You use this number to identify the item. If you specify zero, the system assigns the next available ID in sequence. The maximum number of rows supported is 2147483647.

Images

This column can contain one, two, or three numbers. If you specify one number, it is the index to the small or large image for this item. This number is used as an index into the `Imagelist` objects you specify in both the `imagelist` and `imagelistlarge` properties.

The second number is the index to an overlay image in the same `Imagelist` object as the first number. The indices to overlay images are specified in the `overlays` property of the `imagelist`. You specify the index to the list in the `overlays` property. You can have up to fifteen overlays in one `imagelist`. Thus, if your overlay images were the eighth, ninth, tenth, and eleventh images in the `imagelist`, the `overlays` property would be the vector `8 9 10 11`. If you wanted to use the third overlay with the first image in the `imagelist`, you would specify this column of the appropriate row of the `list` property as `1 3`.

The third number is an index to a user-state image in the `Imagelist` object you specify in the `imagelistuser` property.

You can specify zero for either the second or the third number; you cannot use zero for the first number. If you specify `-1` for an entry, the system triggers the `onGetVirtualData` event handler when it needs information about the image. If you specify `-2` for the image, the system inserts the default value when you are adding or inserting a row, and does not change the value if you are changing other values in an existing row. The defaults are `1 0 0`.

Cells

You can have one or more columns of display information. Each cell contains a character vector. One of the columns contains the primary cells. The values contained in this column display as the labels in all the views. You can display any of the cells in report mode. You designate the column of primary cells by indexing it in the first row of the `columndisplay` property. By default, it is the first column of cells, which is the third column of the `list` property.

You can also assign an empty numeric vector ($\emptyset$) to a cell to mark it as virtual; this triggers the `onGetVirtualData` event handler when the system needs information to display.

You can assign the list property directly, or you can build it using the `AddRows` or `InsertRows` methods to create new items, or the `SetRows`, `SetCells` or `SetImages` methods to change values in existing rows.

If you want to work with virtual data, you can specify an n by 0 matrix (the system creates a matrix of n virtual rows with RowIDs of $1:n$) or you can specify an n by 1 matrix with the RowIDs you want to use. The images column defaults to 1 0 0, and all cells are virtual until you specify them.

The example below shows the list property for the earlier picture:

```
'fmMain.lvSampleLV' [wi 'list']
 1  1 0 3 alpha letter A  arcade graph
 2  2 0 2 beta  letter B   cars  bug
 3  3 0 1 gamma letter C   zigzag clock
```

count (read-only)

A two-element integer vector: the first element returns the total number of entries in the list. In list view or report view, the second element is the number of entries that will fit fully visible in the client area of the Listview; in largeimage and smallimage views, the second number matches the first.

roworigin (read-only)

An integer scalar that contains the row number of the topmost visible entry in list view or report view; in the other views, it is an empty vector.

style

An integer vector that contains selected values or an integer scalar that is the sum of the selected values. The initial value of `style` for any newly created control object is 0. The table below shows the possible values. The `style` property controls aspects of how you format and display the information.

Values for the Listview Class style Property

Value	Description: Choose none, any, or all of the values
0	Highlight image and label when selected (default).
1	Highlight entire row; not just image and label.
2	Stop highlighting at end of last column (only with 1).
4	Highlight selected items even when control loses focus.
8	Disable multiple selections; single selection only.
16	Re-arrange images automatically in both image views.
32	Do not wrap long labels in largeimage view.
64	Disable scrolling. (This setting is ignored in report view.)
128	Do not show headers in report view. (See note 1.)
256	Disable column header buttons in report view. (See note 1.)
512	Enable Edit of primary column text
1024	Enable user to reorder columns by moving headers in report view. (See note 2.)
2048	Enable user to reposition items in largeimage or smallimage views. (See note 3.)
4096	Display gridlines between cells in report view.
8192	Item is selected if cursor hovers over it.
16384	Trigger <code>onActivate</code> when item is clicked
32768	Trigger <code>onActivate</code> on second click (not necessarily a double click).
65536	Enable automatic check boxes as user state images. (See Check Boxes below.)
262144	Suppress unfolding of label tips.
524288	Suppress info tips.

Note 1: If you select style 128, you still must define `columnndisplay` to specify the attributes of columns to use report view effectively. When headers are enabled, each one acts like a button, appearing to be pressed down when the user presses the mouse down on it. You can use the `onColClick` event handler, for example, to sort the items by alphabetizing the cells in the column the user clicks. If you select style 256, it disables the `onColClick` event handler. If you include 128 in style, setting 256 has no meaning.

Note 2: If the user reorders columns, the change is reflected in the `Column Order` element of the `columnndisplay` property.

Note 3: If the user repositions items, the changes are reflected in the `possmallimage` or `poslargeimage` property, depending on the view mode.

columnndisplay

To make full use of report view, you must define the attributes of the columns of information you want to display. Each row of this property defines the attributes of one column of information in the display. The text that displays is from the cells of the list property matrix. The fourth element is an index to the columns of cells as you specified them in the list property. The fifth element specifies the order in which you want the columns to display from left to right.

The first row of the `columnndisplay` matrix defines the primary column; this is the column in which images appear, regardless of the value of its index or order. It also determines which labels appear in views other than report. `columnndisplay` is a seven-column nested matrix. The columns are:

Column Label

A character vector containing the text you want to use as the column header in report view.

Column Width

The width of the column in current scaling units. You must include enough width in the primary column to show the images. The column of primary cells are the labels a user can edit if 512 is included in style; it is the column defined by the first row of `columnndisplay`.

For information on editing labels, see the `BeginLabelEdit` and `CancelLabelEdit` methods, the `onBeginLabelEdit` and `onEndLabelEdit` event handlers, and the `labeledithwnd` property.

Justification

A character vector defining how you want your information justified. (This element assumes character information; it may be extended in future releases to enable formatting of numeric data.) The allowable values are: 'left', 'right', or 'center'. The primary column must be left justified (first row in this property's matrix).

ColumnID

A positive integer specifying the origin one index of the column of cells in the list property; that is, the value 1 refers to the third column of the list property, after the `RowID` and images.

Column Order

A positive integer specifying the order in which you want the columns to display. These values can change when the user reorders columns by dragging column headers if style includes 1024.

Header Image Index

The origin one index in the `Imagelist` to the image you want to display in the header for this column. The `imagelist` is the one specified in the `imagelist` property, which also contains the images used for the items. Note that the height of the header is determined by the font. If the `imagesize` of the `imagelist` is larger than the header, you may need to create images for the headers that contains a blank border on the top and bottom to make the column header images display satisfactorily.

Header Style

A numeric scalar defining how you want the header image justified. By default the value is zero, which places the image to the left of the text. For any column other than the primary cells column, if you set this value to 1, the image displays to the right of the text in the column header. The primary cells column is the one designated by the first row of the `columnndisplay` property; it holds the

images, and the value of its cells display as labels in viewmodes other than report. (This element may be extended in future releases to define more styles.)

The example below shows the `columnndisplay` property for the earlier picture.

```
'fmMain.lvSampleLV' owi 'columnndisplay'  
order      15 left   1  1  0  0  
image      10 center 3  2  0  0  
large image 10 right  4  3  0  0  
user image 10 left   2  4  0  0
```

sortorder

An integer vector of RowIDs permuted in the order the rows are sorted. You can use this property to quickly reorder the rows, for example in the `onColClick` event handler.

poslargeimage

A three-column numeric matrix that specifies the positions of items in `largeimage` viewmode. The first column contains the RowIDs of items; the second and third columns are the row and column coordinates of the upper left corner of each item in the current scaling units for the Listview.

If you create the listview with one of the other viewmodes, this property is empty initially. It obtains values when you set the viewmode appropriately, or you can set this property directly. You can reference and set values for selected items using the `GetPos` and `SetPos` methods. The values remain when you switch viewmodes.

If you set this property, you can choose to specify only a subset of the items. When you switch to the `largeimage` viewmode, the system places those items with values first; it then places the remaining items according to its default algorithms and sets the remaining values for this property. If you add an item when you are in `largeimage` viewmode, the system assigns a value only for that viewmode.

possmallimage

A three-column numeric matrix that specifies the positions of items in `smallimage` viewmode. The first column contains the RowIDs of items; the second and third columns are the row and column coordinates of the upper left corner of each item in the current scaling units for the Listview.

The behavior for setting and using this property is analogous to `poslargeimage`.

highlightfocus

Integer singleton identifying the item that currently has the focus or will be highlighted when the Listview gets the focus. Note that one or more items can be selected, but only one has the focus.

value

Integer vector of RowIDs for the items that are currently selected.

searchstring (read-only)

A character vector containing the current incremental search string. A Listview uses the incremental search string to select an item based on characters typed by the user. This is an empty vector if no incremental search string is in progress or if the Listview is closed. Note that when a user types characters that cause the focus to move, it changes both the `highlightfocus` property and the `value` property. In report view, the search string works on the column of primary cells.

highlightcut

Integer vector of RowIDs for items displayed as Windows displays items that are cut.

highlightdrop

Integer vector of RowIDs for items displayed as Windows displays targets for drag-and-drop operations.

sourceformats

One or more character vectors that specify the format for dragging items from this control. The valid values are 'Text' and 'ListviewItem'. See the section on Drag-and-Drop operations in the Programming the Windows Interface: Intermediate chapter for more information.

targetformats

One or more character vectors that specify the format for dragging items to this control. The valid values are 'FileName', 'FileDrop', 'Text', 'ListviewItem', and 'TreeItem'.

dragimage

The dragimage property controls the image the user drags when a Listview object is the source of the item being dragged and when `□wres` from the `onDragFeedback` event handler contains (or would contain) the value 'defaultcursor' (which is its default). The default value is an empty character vector, which means no special drag image, use the underlying system defaults.

You can set the property to 1, which means use the drag image provided by the class to which it applies; in the case of Listview, this is the icon and the label of the item being dragged. You can assign zero to the property to return to the default. The behavior for this property may be extended in the future to allow custom images while dragging.

infotipmask

The infotipmask property controls which places in the Listview "info tip" are triggered. This can be one or more of the following values: 2 = on icon; 4 = on label; and 8 = on state icon. Use ~1 to select all possible places.

Event-handler Properties:

`onAction`, `onActivate`, `onBeginLabelEdit`, `onClick`, `onClose`, `onColClick`, `onColResize`, `onDbClick`, `onDelete`, `onDestroy`, `onDragContinue`, `onDragEnd`, `onDragEnter`, `onDragFeedback`, `onDragLeave`, `onDragover`, `onDragStart`, `onDrop`, `onEndLabelEdit`, `onExit`, `onExitError`, `onFocus`, `onGetInfoTip`, `onGetVirtualData`, `onHelp`, `onHide`, `onKeyDown`, `onKeyPress`, `onKeyUp`, `onModified`, `onMouseDouble`, `onMouseDown`, `onMouseDrag`, `onMouseEnter`, `onMouseLeave`, `onMouseMove`, `onMouseUp`, `onMove`, `onOpen`, `onPaint`, `onReopen`, `onResize`, `onSelect`, `onSend`, `onShow`, and `onUnfocus`.

onColClick

The `onColClick` event handler is triggered when the user clicks on a column header in report view, unless the style property includes 128 or 256. You can use `onColClick` to allow the user to rearrange the data; for example, you could sort by the cell values in the column the user clicks. `□warg` contains the ID of the column header that was clicked and the shift key state (the sum of 1 for the Shift key, 2 for the Ctrl key, and 4 for the Alt key).

```
▽ ColClick;col;cells;sort;actual
[1] col←□warg[1]+2
[2] actual ← ('f.lv' □wi 'list')[;1]
[3] :if col=3
[4]   'f.lv' □wi 'sortorder' (actual[↑actual])
[5] :else
[6]   cells ← ▷('f.lv' □wi 'list')[;col]
[7]   sort ← □av ↑ cells
[8]   'f.lv' □wi 'sortorder' (actual[sort])
[9] :end
▽
```

onClick and onDbClick

When the user triggers the `onClick` or the `onDbClick` event handler, `□warg` contains a seven-element vector containing information about the location of the click. The first two elements are RowID and columnID, where columnID is defined as it is in the `columndisplay` property. The third element is one of the applicable

location values provided by the Windows `ListView_HitTest` call. It can be one of the following: `image`, `label`, `imageuser`, `above`, `below`, `toleft`, `toright`, `aboveleft`, `aboveright`, `belowleft`, `belowright`, `nowhere`, or an empty vector. The fourth element shows the shift key state: the sum of 1 for the Shift key, 2 for Ctrl, and 4 for Alt. The fifth and sixth elements show the row and column location of the mouse in current scaling units. The seventh element shows the mouse key depressed; 1 for the left, 2 for right. Note that these event handlers are triggered only by user mouse action.

onSelect

The `onSelect` event handler is triggered after the selection state of one or more entries has changed due to user action or setting the value property. This event handler occurs after the `onClick` handler when the user clicks a new item. It occurs in the absence of `onClick` when the user selects an item another way, for example by using the arrow keys. If you include `8192` in `style`, the selection occurs merely by moving the cursor over an item, even without clicking.

Because Microsoft changed the underlying message structure regarding mouse clicks and selection changes, you can sometimes see anomalies. In a `Listview`, in report viewmode where you have only one column, the left and right arrows do not change the selection as they do, for example, in `largeimage` viewmode, or in a `List` control, where the left and right arrows act as up and down arrows. Despite this, the operating system treats a left or right arrow as a selection change. If this is important to your application, you should suppress the left and right arrows in an `onKeyDown` event handler.

onGetInfoTip

When the event handler is triggered, `□warg` is a three-element nested vector. The first element is the `RowID` and `columnID` for which information is needed. The third element is a scalar defining the information needed; it can be one of the following values: 2 = on icon; 4 = on label; and 8 = on state icon.

The value of `□wres` on return from the handler determines the text to be displayed in the tooltip window. You can use `□wres` to display a custom value tip or suppress the value tip by assigning it to `' '`.

When an `onGetInfoTip` handler is defined, the `onGetVirtualData` handler will not trigger for the "tip" data. But when the `onGetInfoTip` handler is not defined, the `onGetVirtualData` handler is triggered as before but it does so without giving meaningful `columnID` and where information; it only triggers when the mouse pointer is above the label and only for the first column.

onGetVirtualData

When the `onGetVirtualData` event handler is triggered, `□warg` is a three-element nested vector. The first element is a character vector defining the information needed; it can be one of the following values: `'cell'`, `'image'`, `'imageoverlay'`, `'imageuser'`, or `'tip'`. The second and third elements are the `RowID` and `columnID` for which information is needed. Image data always has column index of 1.

The `GetVirtualData` event is triggered when a placeholder is encountered and the control needs information to display an item or its image (see the description the `list` property, above). Note that this handler is also triggered when the user hovers the mouse over an item (only the first column of the display in report view), which allows the program to provide an "info tip" for that item (see below).

The vector in the first element of `□warg` identifies the information the system needs for the identified item. You must assign a value to `□wres` for the item in `□warg`. When the event handler returns, if `□wres` contains an improper value or no value for the expected element, the system generates a reasonable default value and does not generate an error. You might structure your event handler in the following manner.

```

    ▽ GetVirtualData; item; row; col
[1] (item row col) ← □warg
[2] :select item
[3]     :case 'cell'
[4]         □wres ← . . . A Cell value
[5]     :case 'image'
[6]         □wres ← . . . A Image index

```

```
[7]      :case 'imageoverlay'
[8]          ␣wres ← . . . A Overlay image index
[9]      :case 'imageuser'
[10]         ␣wres ← . . . A User image index
[11]      :case 'tip'
[12]         ␣wres ← . . . A Text for info tip
[13]      :else
[14]         ␣wres ← 0
[15] :endselect
      ▽
```

An info tip is an item-specific tooltip. If you provide text for an info tip when the cursor hovers over an item and the listview has the focus, the user will see the info tip first followed by the tooltip for the control. The info tip does suppress the automatic behavior of the control that shows "label tips"; these are also like tooltips, but they show the full text of the label of an item when the column is too narrow to display it.

onActivate

This event handler responds when `style` includes either 16384 or 32768. The event is triggered when an item is activated by clicking on its image or primary label; it is not triggered by changing the selection, for example, by arrowing. `␣warg` contains the row id. This event handler differs from `onClick` in that it fires only when the user clicks on the image or primary label. It differs from `onSelect` in that it fires when a user clicks on an item that is already selected. If you have only the larger of the two styles specified, this event handler does not fire when a user clicks to select a new item. It fires when the user clicks a second time or clicks on an item that was selected by a means other than a click, for example, by arrowing to the item.

If you have either of the activate styles set, the text in the first column is highlighted when the user moves the cursor across it, similar to the `HotChange` effect on selectors or command bars. If you include 3 in the style, the effect occurs if the cursor is anywhere on the row.

Methods:

`AddRows`, `Arrange`, `AutoFit`, `BeginLabelEdit`, `CancelLabelEdit`, `Close`, `Create`, `Defer`, `Delete`, `DeleteRows`, `Draw`, `EnsureVisible`, `Event`, `Exec`, `Focus`, `GetCells`, `GetImages`, `GetPos`, `GetRows`, `Help`, `Hide`, `HitTest`, `InsertRows`, `Modify`, `New`, `Open`, `Paint`, `Popup`, `Ref`, `Resize`, `Send`, `Set`, `SetCells`, `SetImages`, `SetLinks`, `SetPos`, `SetRows`, and `Show`.

InsertRows

You can use this method to insert items into the list. The first argument is the RowID of the item you want to precede the new rows. The second argument is a matrix that has the same form as the `list` property. The result is an integer vector of the inserted RowIDs.

AddRows

You can use this method to add items to the list. The argument has the same form as the `list` property. The result is an integer vector of the added RowIDs. New rows are added at the end of the list; you can reorganize them by resetting the `sortorder` property.

SetRows

You can use this method to change the values of the cells and image indices. The argument is a matrix with the same form as the `list` property. All the RowIDs you specify must already exist. The system updates the image and cell information with the values you specify. You can use `-2` for the images entry to leave the indices unchanged. If you specify more columns than exist, they are added to the `list` property.

DeleteRows

You can use this method to delete items from the list. The argument is an integer array of RowIDs.

SetImages

You can use this method if you want to change only the image indices column of the list property. The first argument is an integer vector of the rows you want to modify. The second argument is an integer matrix with the new image, overlayimage, and userstateimage indices respectively.

SetCells

You can use this method if you want to change only the cells columns of the list property. The first argument is an integer vector of the rows you want to modify.

The second argument is an integer vector of the columnIDs you want to modify, where columnID is defined as it is in the `columndisplay` property: the origin one index to the columns following the RowID and images columns. You can add more cells to a row than there are columns in the existing list property. Specify the column indices with the next higher integer values. You cannot add rows with this method. The RowIDs must already exist.

The third argument is a nested matrix of character vectors with the new cell values. You must have one row for each RowID you specify and one column for each columnID you specify.

GetRows

You can use this method to reference the values in specific rows. The argument is an integer vector of RowIDs. The result is a matrix with the same form as the list property.

GetImages

You can use this method to reference the image indices in specific rows. The argument is an integer vector of RowIDs. The result is a three-column matrix with the image index values for the specified rows.

GetCells

You can use this method to reference the cell values in specific rows. The first argument is an integer vector of RowIDs. The second argument is an integer vector of column indices for the cells you want to reference. The result is a nested matrix of character vectors containing cell values with one row for each RowID you specify and one column for each columnID you specify.

EnsureVisible

You can use this method to ensure that a specific item in the list is visible. You can use this only when the Listview is open. The argument is a one- or two-element vector. The first element is the RowID of the item you want the user to see; the second element is a flag that specifies whether it must be fully visible (1) or can be partly visible (0).

If you do not specify the second element, it defaults to 1. Some systems may force the item to be fully visible even if you specify zero. The system returns 1 if the operation was successful.

Arrange

You can use this method in either `smallimage` view or `largeimage` view; the system moves the icons to fit a grid. If you omit the argument or use `'default'`, the system arranges items based on your settings in the `viewalign` property. You can also use `'left'` or `'top'`, which are similar to the left and top settings for `viewalign`. If you specify `'snaptogrid'`, the system adjusts each item to the nearest unoccupied grid location.

The `snaptogrid` argument is useful only if you have moved items to individual locations using drag-and-drop, positioning methods, or position properties (`poslargeimage` and `possmallimage`). See the description of the `SetPos` method for more details.

AutoFit

In report view, you can use this method to automatically fit the width of the columns to match the longest

text string in the column. You can use 'all' as an argument to size all columns or you can specify one or more integer column numbers. You can insert 'header' as the first argument to include the length of the column header in determining the column width.

Using Check Boxes in a Listview

The Listview control allows the display of check boxes in the place of user state images by setting a style value. If you do not have an imagelist defined in the `imagelistuser` property, and you do not have index values set for user state images, when you set `style` to include 65536, check boxes appear for each item in the listview. (As with other user state images, this is best used only in report view or list view.)

Each check box appears to the left of the image, if you have images defined, or to the left of the label if you are not using images; initially the box is unchecked. When the user clicks on a box, it toggles the box between the unchecked and checked states. When the user clicks the check box, it does NOT select the item.

You can tell when a user clicks a check box; the second element of `EventArgs` in the `OnClick` event handler contains the string 'imageuser'. You can tell the state of a check box by using the `GetImages` method and determining the value of the user state image indices in the `list` property of the listview. These values are 1 if a box is unchecked or 2 if it is checked, and they correspond to the positions of the unchecked and checked images in the implicit imagelist used by the control.

You can change the state of a check box by resetting the image values for user state image. You can assign a zero if you want to remove the check box entirely from a node. These values (zero, 1 or 2) appear as the third elements of the second column of the `list` property.

If you have set non-zero values for the user state images, the check boxes do not automatically appear when you set `style` to include 65536. Only those nodes that have 1 or 2 set will display check boxes. However, if the user clicks in the area where the user state image would be, the check box will appear for that item. Whether it appears in a checked or unchecked state depends on the previous value assigned to the user state image. Once it is clicked, the value becomes 1 or 2.

If you assign an existing imagelist to the `imagelistuser` property when you have 65536 in `style`, the imagelist you assign overrides the check boxes. However, if the user clicks on the user state image, the system increases the image index by one, without selecting the item.

The user state image that displays for a given item will cycle through the images available in the imagelist you are using. The system cycles from the last image to zero to the first image. This interaction between the behavior for user state images and the style for check boxes allows you to have a multi-state user image that is under the control of the user. You can query the state of the image as described above.

Parentage:

Listview objects are most often children of a Form, Frame, Page, or CommandBar. They may also be children of some control objects: Button, Check, Combo, Edit, Label, List, Option, Picture, Scroll, Spinner, Status, and Toolbox. Listview objects can contain Timers, Imagelists, or hidden Menus.

MDIForm

Prefix: mdi

Summary:

An MDIForm is a window that acts as the background of an application; it must be a top-level object. An MDIForm can have forms as children, so a Form object is not necessarily a top-level object. (MDI is an acronym for Multiple Document Interface, where "document" is used in the Windows sense that

corresponds to one instance of the application's major use: a document in a word-processing application, a worksheet in a spreadsheet application, or one form that holds a significant segment of an APL application.)

Uses:

The MDI form has some special behaviors that allow you to provide a more flexible user interface with multiple forms and the ability to move between them.

- If a child form has menus, the menu bar from the child form automatically replaces the menu bar of the MDI form when the child form becomes active.
- A child form is clipped so it fits within the MDI form's client area.
- A child form that is minimized appears as an icon within the MDI form.
- A child form must have the default border value (114), a thick border with a menu bar and minimize, maximize, and system menu buttons.

Regular Properties:

activeform, bars, border, caption, children, class, clienthwnd, clientwhere, color, data, def, edge, enabled, events, extent, font, help, helpcontext, hwnd, icon, imagescale, instance, keys, limitwhere, links, methods, mode, modified, modifystop, name, noredraw, opened, place, pointer, prompt, properties, scale, scrollaccel, scrollmargin, self, size, state, style, suppress, targetformats, theme, tooltipenabled, tooltipstyle, tooltiptime, tooltipwidth, translate, value, visible, where, and *Δudp*.

clientwhere

When Windows creates an MDIForm, it actually creates two objects; one is the MDI form itself, and the second is a separate window that exists in the client area of the MDI form (within its borders excluding the title bar). The second window is known as the MDI client. Normally, the MDI client occupies the entire client area of the MDI form. For most purposes, the existence of the separate MDI client object is transparent. However, you may need to be aware of its existence under certain circumstances.

If you want to put a control or use a status line on the MDI form itself, rather than on a regular form that is a child of the MDI form, you must change the size of the MDI client to make room for the control or status line. If you mistakenly make the control, for example a Picture, a child of the MDI form, without changing the size of the MDI client, the behavior of Windows when it repaints the MDI form is unpredictable. You may or may not see the Picture on top of a form that is a child of the MDI form.

style

Values for the style property from zero through 8 are the same for an MDIForm as for a Form; these values specify how the top-level form interacts with the Wait method and other forms. The value 16 for an MDIForm allows scroll bars automatically when needed. The corresponding value for a form means always on top; that effect is not available for an MDI form.

theme

The theme property of a MDIForm class controls the visual styling theme used for drawing the control. The default value is an empty character vector (' '), which uses the new visual styling for the control. To suppress the new visual styling for the control, set thememode to "~".

Event-handler Properties:

onAction, onClose, onDelete, onDestroy, onDragEnter, onDragLeave, onDragover, onDrop, onDropDown, onExitError, onFocus, onHelp, onHide, onKeyDown, onKeyPress, onKeyUp, onModified, onMouseDown, onMouseDrag, onMouseEnter, onMouseLeave, onMouseMove, onMouseUp, onMove, onOpen, onPaint, onReopen, onResize, onSend, onShow, onUnfocus, and onWait.

onDropDown

The onDropDown event handler for an MDI form fires only when there are no child forms active. When a

form is active, even if it is minimized, and even if it has no menus of its own, so that the menus are children of the MDI form, this event handler fires on the form and not the MDI form.

When you make a child form active in the MDI form, its menu appears in the menu bar of the MDI form. In some earlier versions of this system, the `onDropDown` event handler that was triggered by a user pulling down a menu from the menu bar fired on the MDI form rather than on the active child form. In this system, the event fires on the active form whose menu is in the menu bar. If you want this event to fire redundantly on the MDIForm to preserve your code, set the compatibility switch:

```
[Compatibility]
MDIDropDown=1
```

in the APLW.INI file.

onFocus

By contrast, the `onFocus` event handler for an MDI form fires whenever the focus comes to a child form from outside the MDI form. The `onFocus` event handler for the child form fires also. When switching among child forms, the `Focus` event occurs only for the children.

other events

The `onMouseXx` event handlers fire for a child form if the mouse cursor is over that child; they fire for the MDI form when the mouse is touching the MDI client area. Similarly, the `onKeyXx` event handlers fire for the MDI form only when all the child forms are minimized, hidden, or closed.

Methods:

Arrange, Close, ContextHelpMode, Create, Defer, Delete, Draw, Event, Exec, Focus, Help, Hide, Modify, New, Open, Paint, Popup, Ref, Resize, Send, Set, SetLinks, Show, and Wait.

Arrange

You can use the `Arrange` method to tile or cascade all non-minimized forms that are children of the current MDI form or to arrange the icons for minimized forms. You must supply a scalar argument. If you specify zero, the system positions the icons at the bottom of the MDI form. If you specify 1, the system tiles the non-minimized forms horizontally, which means you see the entire width of each form and a portion of the height. If you specify 2, the system tiles them vertically, which means you see the entire height of the left side of each form. If you specify 3, the system cascades the forms so that all the title bars are visible but the forms are overlapping.

See also the description of the `place` property for a form, which you can use to determine the positions of forms when they are restored from a minimized or maximized position on the MDI parent.

Parentage:

MDIForms are top-level objects. They can contain `CommandBar`, `Form`, `Frame`, `Imagelist`, `Label`, `Menu`, `Picture`, `Printer`, `Selector`, `Status`, `Timer`, and `Toolbox` objects.

Media

Prefix: med

Summary:

A utility class that allows you to play a variety of multimedia files or other sources, thus allowing you to add bells and whistles (either literally or figuratively) to your application.

Uses:

A media object is a two-part control. The master or parent control is a rectangular window that allows you to place the control on the screen. The child control is a player that processes files or provides an interface to control a device. The child control also generates a window that can occupy some or all of the parent window. If you are playing a video medium, the image appears in this location.

If you are playing a device, you can have a tool bar or a control mechanism in the window. This window also provides a location where a right mouse click can call a menu to send commands to the device.

There is a distinct possibility of semantic confusion with this control, which has three different types of players underlying it.

Important: The Media object does not support MP3 audio files containing the ID3V2 tag. The reason these MP3 audio files aren't supported is because the underlying Media Control Interface (MCI) used in the Media object doesn't support them.

Animation Control

This uses what Microsoft's Animation control; it plays silent .AVI files. Although the .AVI file type means audio-video interleaved files, the file does not have to have an audio portion. You can create .AVI files using a series of bitmaps.

AniWnd Player

This uses a custom built control that plays .ANI (animated cursor) files.

Multimedia Control

This uses the Windows MCIWnd window class control; MCI stands for Media Control Interface; it is Microsoft's standard for providing device-independent capabilities for controlling various devices using the drivers made for them. This standard allows you to control a variety of different types of devices that play their respective files or other sources, including wave, midi, video and CD audio.

In the simplest scenario, you need only create the parent Media control and specify a file. The system creates the appropriate player and begins playing the named file. Unless you specify otherwise, the first two players play the file repeatedly; the Multimedia control plays a file once.

Note that a Media control can continue to operate while the user takes other actions in your application. Thus you can provide ongoing accompaniment with one or more displays while your application continues.

Regular Properties:

border, children, class, color, data, def, deferexit, edge, enabled, events, extent, file, font, help, helpcontext, hwnd, imagescale, instance, keys, links, mediahwnd, mediatype, methods, mode, modified, modifystop, name, noredraw, opened, order, place, pointer, prompt, properties, range, scale, scrollaccel, scrollmargin, self, size, state, style, suppress, tabgroup, tabparent, tabstop, targetformats, theme, tooltip, tooltipenabled, tooltipstyle, tooltiptime, tooltipwidth, translate, value, visible, where, and *Δudp*.

file

A character vector that contains the name of the file or device to be or being played. If you are playing animated cursor files or silent .AVI files with the Windows Animation control, there is no device. You need to specify only the name of the file you want to play.

Note: The maximum character length supported in the file property is 127. When you specify a file exceeding this limit, the recommended solution is to convert the file to a short named file with the Windows function called GetShortPathName.

If you are using an MCI control, the player is a device. However, in many cases when you are using an MCI control, you can also specify just the file name. The system selects the appropriate device, if one is available, and plays the file. For example, you may specify: `'e:\bells.wav'`

If you specify a file but do not specify the mediatype (for the child control), or if you specify `'auto'`, the value in the file property determines the mediatype. Note that the Animation control cannot necessarily open all `.AVI` files. If you specify an `.AVI` file, and the animation control can open it, the system will use the Animation control. Otherwise, it will use the Multimedia player.

Some devices do not use files as their storage medium. If you want to use a device that plays something other than a file, you must specify the device name and use the available commands or buttons to choose the item to play. In general, these devices include audio CD players, videodisc players, videotape recorders or players, and digital audio tape players.

In many cases, the name of the device is the generic device type; for example, you may be able to specify `'cdaudio'` as the device name. If you have more than one similar device attached to your computer, you may need to end the designation with a number, for example: `'cdaudio2'`

If the device has an explicit name, you may have to specify the exact name; for example, `'ReelMagic'`. You can enumerate the list of devices attached to your computer by using the `mcidevices` function shown below and found in the Windows workspace.

If you are playing the Multimedia player with a file, you can choose a non-default device by specifying both the device name and the file name separated by an exclamation point, for example:

```
'fmHolder.medPlayer' □wi 'file' 'MPEGVideo!test.mpg'
```

mediatype

A two-element nested vector that contains the requested child control type and the actual child control type. When you specify `mediatype`, you specify only the first element; if you specify both elements, the system ignores the second. The default value for the first element is `'auto'`. When you specify the default, the file determines the actual child control type.

If you wish to specify the type explicitly, you can use `'ani'`, `'avi'`, or `'mci'` for the first element; the second element automatically becomes the same. You can use this capability, for example, to force a simple `.AVI` file to be played by a Multimedia player, even though an Animation control is capable of playing it. However, note that an Animation control has far less overhead than the Multimedia player.

range

A two-element vector that contains beginning and end values for the file or source. The default units of measurement depend on the device and file or source; often range is measured in frames or milliseconds. This property does not apply to the Windows animation control that plays silent `.AVI` files; if you query this property when the second element of `mediatype` is `'avi'`, the system returns an empty vector.

Caution: If you change the time format from the default, you may have to manipulate the value to have it be meaningful. See the examples below for further information.

value

An integer scalar that contains the current location in the file or source. As with `range`, the default units of measurement depend on the device and file or source. This property does not apply to the Windows animation control that plays silent `.AVI` files; if you query this property when the second element of `mediatype` is `'avi'`, the system returns an empty vector.

mediahwnd

The window handle of the child control window. This hwnd property is the window handle of the parent control.

style

An integer vector that contains selected values or an integer scalar that is the sum of the selected values. The initial value for **style** is zero. The valid values are shown below. You can use one of the primary values to position the player on the control. You can add in the other values. The higher numbers apply to an MCI player only.

Values for the Media Class style Property

Value	Description
Primary Value: Choose only one value	
0	Position the child control window in the upper left corner of the Media window.
1	Resize the Media window to match the size of the child control window.
2	Center the child control window in the Media window.
3	For an MCI player, stretch the child control window to fill the Media window. For other child controls, this is the same as value 0.
Add-on Values: Choose none, any or all of these values	
4	Draw the file transparently. (See note 1.)
8	Do not automatically play the file when it is opened. (See note 2.)
16	For an MCI player, include a toolbar in the child control window. (See note 3.)
32	For an MCI player, display a context menu in response to a right mouse click in the child window; if used with 16, place a menu button on the toolbar. (See note 4.)
64	For an MCI player with menu, do not include an Open menu item. Use with 32.
128	For an MCI player, allow recording. Use with 32. (See note 5.)
256	For an MCI player, inhibit the error message box. (See note 6.)

Note 1: For the Animation Control, the system draws the AVI using the Media window's background color rather than the background color specified in the file. For the Animated cursor player, the system draws the ANI file transparently on the Media control's parent window or whatever window(s) overlap the Media control.

Caution: If you put an ANI control on top of another animated control, you need to be aware that Windows, for efficiency, copies the background during WM_PAINT processing and redraws it during WM_TIMER processing. Thus, drawing done by overlapped windows NOT during WM_PAINT processing may be overwritten by the Media control during WM_TIMER messages.

Note 2: This setting controls the initial play only. Once playing, those files that automatically repeat will do so regardless of the style setting.

Note 3: The toolbar has buttons relevant to the particular type of media device being played; typically it contains a start/stop button and a slider for setting the playback position within the file. The toolbar allows the user to control the playback of the file directly.

Note 4: The menu allows the user to control the playback of the file in ways the toolbar alone does not. The user can even open a file directly from the menu, unless the value 64 is included in style. The items in the menu depend on the type of file.

Note 5: If the type of file being played allows recording, for example, .WAV file, the toolbar includes a Record button and the context menu includes appropriate commands for creating a new file and saving the recorded file.

Note 6: If an error occurs (except while setting the file property or invoking the Command method), the MCI player normally displays a message box with an error message; this style inhibits the message box.

theme

The theme property of a Media class controls the visual styling theme used for drawing the control. The default value is an empty character vector (' '), which uses the new visual styling for the control. To suppress the new visual styling for the control, set thememode to "~".

Event-handler Properties:

onAction, onClose, onDelete, onDestroy, onDragEnter, onDragLeave, onDragover, onDrop, onError, onExit, onExitError, onFile, onFocus, onHelp, onHide, onKeyDown, onKeyPress, onKeyUp, onModified, onMouseDouble, onMouseDown, onMouseDrag, onMouseEnter, onMouseLeave, onMouseMove, onMouseUp, onMove, onOpen, onPaint, onReopen, onResize, onSend, onShow, onStart, onStop, and onUnfocus.

onStart and onStop

The onStart and onStop events are triggered when the player starts playing a file and stops playing a file, respectively. For the onStop event handler, `¶warg` contains 1 if the device was stopped; zero if it was paused. (This is meaningful only for the MCI player.)

onFile

The onFile event handler is triggered when the current file was closed or a new file was opened by means other than setting the file property; for example, by using the Open command on the context menu. `¶warg` contains a character vector that is the name of the new file, if any. This event handler applies to an MCI player only.

onError

The onError event handler is triggered when an error is signaled by the MCI player. `¶warg` contains a two element nested vector: the first element is the integer error code; the second element is a character vector containing the error message. This event handler applies to an MCI player only.

Methods:

Close, Command, Create, Defer, Delete, Draw, Event, Exec, Focus, Help, Hide, Modify, New, Open, Paint, Play, Popup, Ref, Resize, Seek, Send, Set, SetLinks, Show, and Stop.

Stop

You can use this method to stop the current file from playing. For .ANI and .AVI files, and some other files, an image remains on the screen.

Play

You can use this method to restart, or start, playing a file. Optionally, you can include an argument of up to three elements. The first element specifies a starting location; the second element specifies an ending location; the third element specifies the number of repetitions for the animated cursor player or the Animation player. The third element is ignored by an MCI player.

The starting and ending locations are frame numbers, starting with zero, for the players that play .ANI and .AVI files exclusively. For MCI devices, the units differ depending on the device; they might be in seconds for example. Each of the elements can be a non-negative integer, or you can specify -1. For the starting position, this specifies the current position; for the ending position, it specifies the end of the file. For the third element, it means repeat indefinitely, (except for the MCI player). You can make an MCI player repeat by using the Command method with the 'play repeat' argument.

Seek

You can use this method to move to a specific position in the file and display that frame without continuing to play the file. The argument is a scalar integer that is the file position. You can reference the `range` property to find the valid values.

Command

The `Command` method applies only to an MCI device player. You can use this method to send a command to the device. The argument is a character vector. For example, you can send `'set door open'` to a `cdaudio` device to eject the CD, and `'play repeat'` to a `digitalvideo` (AVIVideo) device. See the documentation for the device or the Microsoft documentation for the Media Control Interface for a list of possible commands. The result is a two-element nested vector with the integer error code and a character vector error message if an error occurred, or a character vector with the response from the device (no error).

Parentage:

Media objects are most often children of a `Form`, `Frame`, `Page`, or `CommandBar`. They may also be children of some control objects: `Button`, `Check`, `Combo`, `Edit`, `Label`, `List`, `Option`, `Picture`, `Scroll`, `Spinner`, `Status`, and `Toolbox`. Media controls can contain a `Timer`, `Imagelist`, or invisible `Menu`.

Examples:

These examples show some of the Commands you can send to a Multimedia player. You can also use the `range` and `value` properties effectively. The first example shows a `.AVI` file that the simple animation player could not open; hence, the control used an MCI player.

```
'fmForm.medTest' [wi 'New' 'Media'
'fmForm.medTest' [wi 'onStop' '[]wevent []warg'
'fmForm.medTest' [wi 'file' 'c:\widgets\nyctraff.avi'
'fmForm.medTest' [wi 'mediatype'
auto mci
'fmForm.medTest' [wi 'Command' 'Pause'
Stop 0
'fmForm.medTest' [wi 'Command' 'Info File'
c:\widgets\nyctraff.avi
'fmForm.medTest' [wi 'Command' 'Status Audio'
on
'fmForm.medTest' [wi 'Command' 'Set Audio all off'
'fmForm.medTest' [wi 'Seek' 400
'fmForm.medTest' [wi 'value'
400
'fmForm.medTest' [wi 'file' '\Windows\media\chimes.wav'
'fmForm.medTest' [wi 'Command' 'Status Time Format'
milliseconds
'fmForm.medTest' [wi 'range'
0 720
'fmForm.medTest' [wi 'Command' 'Status Length'
720
```

In the next example, the `msf` time format stands for minutes/seconds/frames. On this device, there are 75 frames per second. The commands `Capability`, `Set`, and `Status` can each be followed by various arguments.

```
'fmForm.medTest' [wi 'file' 'cdaudio'
'fmForm.medTest' [wi 'Command' 'Capability Can Record'
false
'fmForm.medTest' [wi 'Command' 'Capability Can Play'
true
'fmForm.medTest' [wi 'Seek' 2400
'fmForm.medTest' [wi 'Command' 'Set Time Format msf'
'fmForm.medTest' [wi 'Command' 'Status Length'
24:58:51
'fmForm.medTest' [wi 'Command' 'Status Length Track 1'
4:10:17
```

When you change the time format for some devices, the control returns a formatted string in response to Commands; for example, the 24:58:51 for length above. However, if you query the range or value property, the system returns integer values. This integer may be an obscure value, for which you must know the format, if you want to use it. For example, when using the minutes/seconds/frames time format, cdaudio devices pack the values in reverse order. Thus you must not only represent the number in its constituent parts, you must also reverse the order.

```
'fmForm.medTest' □wi 'range'
0 3357208
  Φ 256 256 256 τ 3357208
24 58 51
```

The corresponding numbers in the default milliseconds time format allow for the leader at the beginning of the disc. Once you know the format, the numbers correspond. The end of the disc is at 1,500 seconds or 25 minutes, less two seconds of leader. The 681 ms corresponds to 51 frames.

```
'fmForm.medTest' □wi 'Command' 'Set Time Format ms'
'fmForm.medTest' □wi 'Command' 'Status Length'
1498681
'fmForm.medTest' □wi 'range'
2001 1500682

  60 60 1000 τ 1498681
24 58 681
  1681×.075
51
```

Enumerating available devices:

```
[0]   r← mciDevices;□io;n;i;z
[1]   □io←0
[2]   z←□wcall 'mciSendString' 'sysinfo all quantity' (255ρ
↳     □tcnul) 255 0
[3]   → (0≠ ↑z)/△Error
[4]   n← □fi ((1▷ z) ∩ □tcnul) ↑ 1▷ z
[5]
[6]   r← ''
[7]   :for i :in 1+ ∩n
[8]     z← □wcall 'mciSendString' ('sysinfo all name ', φi)
↳     (255ρ□tcnul) 255 0
[9]     → (0≠ ↑z)/△Error
[10]    r←r, c ((1▷ z) ∩ □tcnul) ↑ 1▷ z
[11]  :endfor
[12]  :return
[13]
[14]  △Error:
[15]  z← □wcall 'mciGetErrorString' (↑z) (255ρ□tcnul) 255
[16]  □error ((1▷ z) ∩ □tcnul) ↑ 1▷ z
```

Menu

Prefix: m

Summary:

A Menu object usually corresponds to a selectable name on a menu bar, a menu that drops down from the menu bar, or a menu that cascades from another menu item. Optionally, you can include an image on a child menu item. You can also define a visible Menu object as a separator line on a drop-down or cascading menu, and you can define an always-hidden Menu object as a child of a control to define a shortcut key for that control.

Uses:

You use menus to provide users a selection of actions they can take on a broad scale for the form. Menu objects use a hierarchical naming system. If you have a form named `fmMain` and you want a menu item displayed as File with choices Open and Exit, your first control object is `fmMain.mFile` and the menu choices are `fmMain.mFile.mOpen` and `fmMain.mFile.mExit`.

As part of defining the Menu object, you can specify an access key sequence and a shortcut key combination as additional ways for a user to select the item. Access keys work only with menu items that you can see on the screen; shortcut keys work even when you cannot see the item on the screen.

An access key is a letter (or other character) that a user can press to select the menu item when the menu item is on the screen. For a choice on a drop-down menu, this means the menu must be in a dropped-down position with the highlight on the caption of the menu in the menu bar or some item in the menu. For a top-level Menu object, which is on the menu bar, a user can use the access key if the focus is on any item in the menu bar. A user can place the focus on the menu bar by pressing and releasing the Alt key. For example, to open a file a user can typically press the sequence Alt, F (which causes the File menu to drop down), and O. Usually, the user can combine the first two keystrokes, Alt+F.

You do not have to provide access keys for every menu to allow a keyboard interface. For example, a user can press Alt to place the focus on the menu bar, use the left and right arrow keys to select the menu choice, press the down arrow key to open the menu, arrow down, and then press the Enter key for an item on the drop-down menu.

You can also define a shortcut key combination that allows users to select a menu item without actually exposing the menu. For example, to open a file users can typically press Ctrl+O. In APL64, either an access key or a shortcut key triggers the same action that clicking on the object or selecting it by using the Enter key does. Note that Menu objects that have children cannot generate Click events. You can have a shortcut and execute an `onClick` event-handler only for a low-level menu item.

You can create a menu, which does not show on the form, as a child of another control object for the purpose of defining a shortcut key. When the focus is on the object and a user presses the shortcut key combination, the system executes the `onClick` event handler for the menu. If you define the same `onClick` handler for the Menu as for its parent object, the system executes it without the user actually clicking on the object or tabbing to it and pressing Enter. You can use this technique with Button, Check, Combo, Edit, Frame, List, Option, and Scroll.

Regular Properties:

caption, children, class, data, def, enabled, events, help, helpcontext, imageindex, imagescale, imagelist, instance, links, methods, modified, modifystop, name, opened, order, prompt, properties, self, separator, shortcut, state, style, suppress, translate, value, visible, and *udp*.

value

Indicates whether the menu item is checked (1) or unchecked (0).

style

An integer vector that contains selected values from the following list or an integer scalar that is the sum of the selected values. The initial value of style for any newly created control object is 0.

Values for the Menu Class style Property

Value	Description
0	Normal menu.
1	User clicking the menu item toggles a visible check mark. (See note 1.)
2	Window menu; automatically shows children of an MDI form. (See note 2.)

Value	Description
4	Radio style; only one menu of a radio group can be selected at a time. (See note 3.)
8	Menu caption appears in bold; only one per group of siblings. (See note 4.)

Note 1. When a menu item is checked, its `value` property returns 1. You can set the `value` property of a style 0 menu to make a check visible, but the user cannot induce it by clicking.

Note 2. The Window menu is a top-level menu that you place on a form that is the child of an MDI Form. It automatically shows a list of all the forms that are children of the same MDI parent. The active form is checked. The user can select a different form by clicking on that menu item. If you place an identical Window menu on each child form, the menu will be available regardless of which form is currently active. If some of the child forms have no menu bar, you can add a Window menu to the MDI parent; the menu will then be available when the active child has no menus. You can also place other menu items on the Window menu; you might want to provide the user with the option to cascade or tile his child forms, for example. Such items appear above the list of child forms, separated from the list by a line.

Note 3. If you include 4 in the `style` property, the menu item is a radio style menu. There are two aspects to a radio style menu. The first is visual; when the menu is checked, that is when its `value` property is 1, a solid round marker appears instead of the traditional check mark. The second aspect is behavioral. When adjacent items are both or all radio style menus, they function as a group only one of which may be selected or checked at a time. This is the behavior of Option buttons, which are also called radio buttons; hence the name.

This style is most effective when combined with style 1, which allows the user to toggle the checked state of the menu. When the user clicks a style 5 menu that is unchecked (or when you change the `value` from zero to one), it toggles the radio marker on; if the menu is part of a radio group, the system sets the other menus in the group to be unchecked. If the menu item is a single radio-style menu, the user can also toggle the marker off; if the menu is part of a radio group, the user can cause the menu item to be unchecked only by clicking on one of the other members of the group. You can set the `value` property to zero under program control to have all the members of the group unchecked.

If you have assigned images to the menus, the system uses the second indexed image in lieu of the radio marker and the first indexed image in lieu of the blank unchecked appearance.

Note: Due to the underlying Microsoft implementation, there is an unintuitive quirk in this feature. If you set a menu to be a radio-style menu, and then change it back to a style 1 menu (which toggles the checkmark), the menu retains the radio marker. The APL system excludes it from the group, however.

Caution: As a result of implementing this style, the checkmarked state of a menu item is now preserved when a menu or its parent is closed and reopened. If you have been depending on the system to reinitialize your style 1 menus, you will now have to reset their `value` properties.

Note 4. You can specify one item on a menu as the default by including 8 in the `style` property; only menu items that do not have children may be default items. The term "default" in this context means it is the default choice for the menu of which it is a part; it is not the "normal" or default menu you get when you simply specify a menu object. The menu caption appears in bold. If you set a second menu item in the same group of siblings as a default, the first one is no longer bold. If the user doubleclicks the parent of a default menu, it triggers the `onClick` handler for that child.

caption

A character vector that holds the text of the menu or menu choice. If you want to define an access character for a menu or menu choice, place an ampersand (&) before the character you want underlined.

If you want to specify a shortcut key hint for the user, separate the `caption` into two parts using a TAB character (`\t`). The first part is the left-justified menu choice; the second part is the shortcut hint that appears to the right of the menu choice when the menu is open. The system does not display shortcut hints for top-level menu items (the names that appear in the menu bar).

Note that the shortcut hint specified in `caption` affects only the visual appearance of the menu. The `shortcut` property defines the actual key values. When you set the `shortcut` property, the system automatically modifies the menu's `caption` to reflect the shortcut key names (for example, `Ctrl+O`). If you do not want the shortcut hint to show or if you want to specify it differently (in another language, for example), modify the `caption` property after you set the `shortcut` property.

order

The index position of an item in its level of the menu hierarchy. If your menu bar reads: File, Edit, . . . Help, and your top-level menu objects are named `fmMain.mFile`, `fmMain.mEdit`, . . . `fmMain.mHelp`, their `order` values are 1, 2, . . . n.

On the same form, if your File menu has the choices Open and Exit, then the menu objects `fmMain.mFile.mOpen` and `fmMain.mFile.mExit` have `order` values 1 and 2.

Similarly, for the Edit menu, the objects `fmMain.mEdit.mCut`, `fmMain.mEdit.mCopy`, and `fmMain.mEdit.mPaste` have `order` values 1, 2 and 3.

Note also that the order of the top-level menu items is a series distinct from the `order` values of other controls, such as buttons and frames, that are children of the same form. As you use the `New` method to create control objects on a form, the system assigns each control the next available `order` number. You can rearrange the order in which menu items appear by setting the `order` property. You can use fractional values to insert an item between two existing items. The system reassigns the `order` numbers of all the items that have a higher value.

separator

Indicates whether the menu item is a separator line (1) or a regular menu item (0). A top-level menu item (one that appears in the menu bar) or a menu item that has children that are menu items (cascading menu) cannot be separators.

shortcut

A two-element vector that defines a shortcut key combination for the menu item. The first element designates the base key; the second element the shift keys that you combine with the base key. When a user presses this key combination, the system executes the menu's `onClick` event handler if the menu is enabled.

You can specify the first item as a character or integer singleton. A character singleton literally means that key value; that is 'A' specifies that the A key is the base key. You use a numeric value to denote a Windows virtual key code for keys you cannot represent with a single letter. A table of commonly used virtual key codes appears in the description of the `shortcut` property in the Windows Interface: Properties chapter in this manual. The entire list appears in the Virtual Key Codes appendix in this manual. The table below shows the values for the shift keys; the second element of the `shortcut` vector is the sum of these values.

Shift Key Values

Value	Description
0	Base key alone
1	Shift
2	Ctrl (the default)
4	Alt

imagelist

A character vector containing the name of the `Imagelist` object that contains the images you want to display on the menu. You may not place images on top level menus. The size of the font the system uses for the menu caption determines the size of the image that can be displayed. The system clips larger images. This size is determined by the system setting on each individual system; for Windows Display settings of small fonts, you may find this is 13 pixels high.

Caution: Because of the underlying call that Windows uses to place images on menus, it works best if you use monochrome bitmaps. Not all colors are reproduced faithfully.

imageindex

An integer singleton or two-element vector containing the origin one indices for the images to display. The first value is the index for the image when the menu is unchecked; the second value is the index for the image when the menu is checked. If you specify only a single value, that image appears both when the menu is checked and when it is unchecked. Specifying zero results in no image even when the `imagelist` property is set. Thus, if you want to have an image instead of the check mark, you can specify the `imageindex` property as `(0 1)`.

Event-handler Properties:

`onAction`, `onClick`, `onClose`, `onDelete`, `onHelp`, `onModified`, `onOpen`, `onReopen`, and `onSend`.

Note that the `onDropDown` event handler, which exists for a `Form` or an `MDIForm`, fires when the user clicks on a menu item on the menu bar that has children and therefore causes something to drop down, or when the user presses the `alt` key and causes the focus to move to the menu bar. You can use this event handler to set the state of a menu before the user sees the choices.

Methods:

`Click`, `Close`, `Create`, `Defer`, `Delete`, `Event`, `Exec`, `Help`, `Modify`, `New`, `Open`, `Ref`, `Send`, `Set`, and `SetLinks`.

Parentage:

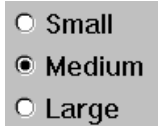
Menus are children of `Form`, `MDIForm`, or other `Menu` objects, primarily. They can be top-level objects to use as popup menus, and hidden children of control objects for assigning a shortcut key. Menus cannot be children of a `CommandBar`, `CommandButton`, `Imagelist`, `Printer`, or `Timer`. They can contain other `Menu` objects, `Imagelists`, or `Timers`.

Example:

The following example specifies the menu for a form named `Sample`. Under the `Chart` menu the two options labeled `Bar` and `Line` are active, followed by a separator and a greyed option named `Pie`. You can press `Shift+Alt+L` as a shortcut for `Line`.

```
la ← 'fmSample.mChart' □wi 'New' 'Menu'
la □wi 'caption' 'Chart'
(la, '.mBar') □wi 'New' 'Menu'
(la, '.mBar') □wi 'caption' 'Bar'
(la, '.mLine') □wi 'New' 'Menu'
(la, '.mLine') □wi 'caption' 'Line'
(la, '.mLine') □wi 'shortcut' 'L' 5
(la, '.mSep') □wi 'New' 'Menu'
(la, '.mSep') □wi 'separator' 1
(la, '.mPie') □wi 'New' 'Menu'
(la, '.mPie') □wi 'caption' 'Pie'
(la, '.mPie') □wi ('enabled' 0)
```

Option



Prefix: op

Summary:

A circle with associated text that is one of a set of mutually exclusive choices; also called a radio button control. When users click the mouse on the circle or its text, or place the focus on the object, they cause a filled-in center to appear, giving a visual indication of an on setting for the choice specified by the text. The same action triggers the `onClick` event-handler. If a user selects or moves the focus to another option in the same group, the system clears the filled-in center from this option.

Uses:

Options appear in groups of two or more. You can select only one of the option buttons in the group; when you select one, it unselects another that is already selected.

Regular Properties:

border, caption, children, class, color, data, def, deferexit, edge, enabled, events, extent, font, help, helpcontext, hwnd, imagescale, instance, keys, links, methods, mode, modified, modifystop, name, noredraw, opened, order, place, pointer, prompt, properties, scale, scrollaccel, scrollmargin, self, size, state, style, suppress, tabgroup, tabparent, tabstop, targetformats, theme, tooltip, tooltipenabled, tooltipstyle, tooltiptime, tooltipwidth, translate, value, visible, where, and *Δudp*.

caption

The text beside an Option control.

value

To set or reference whether an Option control is selected (1) or not selected (0). Each time the `value` is set to 1, it triggers the `onClick` event-handler, unless the `suppress` property is set to 1.

style

The initial value of `style` for any newly created control object is 0. The `style` property for an Option object is an integer vector or scalar that contains either zero or 1.

Values for the Option Class style Property

Value	Description
0	Radio button on the left of text
1	Radio button on the right of text.

theme

The `theme` property of a Option class controls the visual styling theme used for drawing the control. The default value is an empty character vector (' '), which uses the new visual styling for the control. To suppress the new visual styling for the control, set `thememode` to "~".

order

The sequential position of this control among all the controls (other than menu) that are children of the object that contains it. `order` values are a sequence of positive integers starting with 1. The `order` property defines where the focus moves when a user presses the Tab key. (See also the description of the `tabstop` and `tabparent` properties in the Windows Interface: Properties chapter.)

For objects that are grouped, the tab moves to the first object in the group and the arrow keys move among the members of the group. Pressing the Tab key when the focus is on any member of the group places the focus on the next ungrouped object or the first member of the next group, whichever immediately follows the order value of the last member of this group.

As you use the New method to create control objects on a form, the system assigns each control the next available order number. If the last object you created was an Option object, this new Option object is in the same group. This means you must create all instances of an Option group sequentially or you must assign the order numbers so they group properly.

Typically, there is a nongrouped object that separates a group, such as a `Label` above or a `Frame` that surrounds the group. If you do not have a natural separator, you can create a hidden `Label` to separate distinct groups of `Option` objects. (Alternatively, you can use the `tabgroup` property.)

Event-handler Properties:

onAction, onClick, onClose, onDelete, onDestroy, onDragEnter, onDragLeave, onDragover, onDrop, onExit, onExitError, onFocus, onHelp, onHide, onKeyDown, onKeyUp, onModified, onMouseDouble, onMouseDown, onMouseDrag, onMouseEnter, onMouseLeave, onMouseMove, onMouseUp, onMove, onOpen, onPaint, onReopen, onResize, onSend, onShow, and onUnfocus.

Methods:

Close, Create, Defer, Delete, Draw, Event, Exec, Focus, Help, Hide, Modify, New, Open, Paint, Popup, Ref, Resize, Send, Set, SetLinks, and Show.

Parentage:

Options are most often children of a Form, Frame, Page, or CommandBar. They may also be children of some control objects: Button, Check, Combo, Edit, Label, List, Option, Picture, Scroll, Spinner, Status, and Toolbox. The system allows most classes as children; it specifically prohibits Form, MDIForm, CommandBar, CommandButton, Page, and Printer.

Example:

The following examples create a group of three option buttons labeled Small, Medium, and Large. By setting the value property for the Medium option to 1, it becomes the default.

```
ops <- 'fmSample.opSmall' %wi 'New' 'Option'
ops %wi 'where' (w <- 1 1 1.5 15)
ops %wi 'caption' 'Small'
'fmSample.opMedium' %wi 'New' 'Option' ('where' (w+1.5 0 0 0))
%> ('value' 1) ('caption' 'Medium')
'fmSample.opLarge' %wi 'New' 'Option'
'fmSample.opLarge' %wi 'where' (w+3 0 0 0)
'fmSample.opLarge' %wi ('caption' 'Large')
```


Page

Prefix: pg

Summary:

A Page is a container for a group of related controls; it is one of a series of mutually exclusive children of a Selector object. Pages are usually represented in their parent Selector by rectangular notebook-divider-like tabs or buttons across the top of the Selector.

Uses:

When a Page is selected, its controls are displayed in the client area of the Page object's parent Selector. The Page is an alternative to a popup window that you create and destroy dynamically.

Regular Properties:

bars, border, caption, children, class, color, data, def, edge, enabled, events, extent, font, help, helpcontext, hwnd, imageindex, imagescale, instance, links, methods, mode, modified, modifystop, name, noredraw, opened, order, place, pointer, prompt, properties, scale, scrollaccel, scrollmargin, self, size, state, suppress, tabgroup, tabparent, tabstop, targetformats, theme, tooltip, tooltipenabled, tooltipstyle, tooltiptime, tooltipwidth, translate, visible, where, and Δudp .

where, extent, and size (read-only)

Because a Page is constrained to exactly fit its parent's client area, the extent, size, and where properties are read-only and depend on the extent of the parent Selector.

caption

The text to be displayed on the tab for the Page. The length of the caption affects the length of the tab on the page; typically the system expands the tab to show the full caption.

imageindex

The origin one index of the image you want to appear on the tab from the imagelist named in the imagelist property of the parent Selector.

order

The index position of the Page relative to other Page children of its parent Selector object. Like the order property for Menu nodes, the order for Page objects is separate from the order values of other child controls of the parent Selector.

theme

The theme property of a Page class controls the visual styling theme used for drawing the control. The default value is an empty character vector (' '), which uses the new visual styling for the control. To suppress the new visual styling for the control, set thememode to "~".

visible (read-only)

This property is read-only because the Selector control changes the visible property as pages are selected. The value is 1 for the currently selected Page, otherwise -1, except that a newly-created page that has never been opened can have a visible property of 1.

Event-handler Properties:

onAction, onClose, onDelete, onDestroy, onDragEnter, onDragLeave, onDragover, onDrop, onExitError, onHelp, onHide, onKeyDown, onKeyUp, onModified, onMouseDouble, onMouseDown, onMouseDrag, onMouseEnter, onMouseLeave, onMouseMove, onMouseUp, onOpen, onPaint, onReopen, onResize, onSend, onShow, and onValidate.

onValidate

This event handler fires when a new Page of a Selector object is selected, or when the Validate method is invoked. You can use the onValidate event handler to validate user input on the previously selected Page and to allow or disallow the Page to change.

During this event handler, `□warg` contains two items: a numeric code that indicates what caused the validation to occur and a nested vector of the newly selected page and the previously selected page. The code is 2 if the user changed the page by using the mouse, 1 by keyboard input, or zero if the reason is unknown. The code is negative if the program induced the change; -1 for the Validate method; -2 if the value property was changed; or -3 if the Close method was applied to the currently selected page.

The onValidate handler for the currently selected Page is executed first. If it succeeds, the onValidate handler for the Selector is called. When the event is fired, `□wres` is set to 1. The handler must set `□wres←0` before returning to indicate failure; any nonzero value indicates success. If the validation fails, a well-designed interface will display a modal message box indicating the reason for failure, and on return from the message box, put the input focus on the offending control so the user can fix the problem. In the case of edit controls this should usually be accompanied by selecting the text to make replacement easy.

Methods:

Close, Create, Defer, Delete, Draw, Event, Exec, Help, Modify, New, Open, Paint, Popup, Ref, Resize, Send, Set, SetLinks, and Validate.

Close

If you Close the currently selected Page, the system sets the value property of the Selector to zero.

Validate

The Validate method emulates a Validate event and triggers the onValidate handler. You can use this method to perform the validation that would be done if the user tried to switch Pages by clicking on a tab. This might be used in the onClick handler for an OK or Apply button.

When applied to a Selector, the onValidate event handler of the current Page is invoked first. If it returns a nonzero value in `□wres` then the onValidate event handler of the Selector is called. The method returns 1 if both onValidate event handlers return a nonzero value, otherwise it returns 0.

When applied to a Page, only the onValidate event handler for the Page is invoked. The result is 1 if the event handler returns a nonzero value, otherwise it is 0.

Parentage:

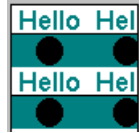
Page objects are children of Selector objects only. They can contain Button, Check, Combo, CommandBar, DateTime, Edit, Frame, Imagelist, Label, List, Listview, Media, Option, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, and Tree objects. A Page can also contain a hidden Menu object for the purpose of assigning a shortcut key to the Page.

When a new Page is selected, the following things happen in this order:

- 1 The onValidate handler for the old Page is triggered. If 0 is returned in `□wres` the remaining events do not occur and the current Page does not change.
- 2 The onValidate handler for the Selector is triggered. If 0 is returned in `□wres` the remaining events do not occur and the current Page does not change.
- 3 The onHide handler for the old Page is triggered and the Page is hidden.
- 4 The value property of the Selector is changed to reflect the new Page.
- 5 The onChange handler for the Selector is invoked.
- 6 The onOpen handler for the new Page is invoked if the Page is not already open.

- 7 The onShow handler for the new Page is invoked.
- 8 The onPaint handler for the new Page is invoked.

Picture



Prefix: pic

Summary:

A defined area that displays persistent bitmap or color-encoded graphics.

Uses:

A Picture control is a graphical element that you use to manipulate and display bitmap or color-encoded graphics. It differs significantly from a Frame or other control on which you draw graphics, because the graphic you draw on a frame is transitory. If you move the form, minimize it, or partially overlay the frame with another form, the graphic is blotted out, and you must redraw it. With a Picture control, the graphic is persistent; the system redraws it as necessary.

You can think of a Picture control as an observation window through which you view an underlying graphic. If you block the observation window temporarily, the graphic is not erased, as long as the control remains open. If you Close the Picture or its parent form, you must redraw the image on the Picture. Similarly, if you set a property that requires the system to close the Picture, for example, border, you must redraw its image.

When you create a Picture object, the size of the observation window is the same as the size of the underlying image area. If you change the size of the window, the size of the image area does not change; however, you can change the size of the image area independently. The control can display a portion of a graphical image that is larger or multiple copies of an image that is smaller than the client area of the control. If you make the size of the underlying image area larger than the Picture, you can add scroll bars to the Picture and allow the user to select the portion of the graphic that is visible.

This is analogous to the document in a word processing system that you load into active memory. You can see any portion of the document that fits on your screen, and you can scroll the document to see any portion of it. While the document is active, you can also make changes to it and view those changes. If you switch away from the document and return to it, the document and any changes you made are still visible.

Similarly, you can modify the actual image, whether you initialize it with a bitmap or not. You can use any of the operations of the Draw method to enhance the image. In addition, you can modify rectangular sections of the image using pairs of methods that save and paste the color encoded values. When you modify the image property with any of these methods, the resulting graphical image is persistent, as explained above, but not permanent. If you want to save a modified image, you must either save it as a separate variable or re-initialize the Picture object.

Regular Properties:

bitmap, bitmapsizes, border, children, class, color, data, def, deferexit, edge, enabled, events, extent, font, hdc, help, helpcontext, hwnd, image, imagescale, imagesize, instance, keys, links, methods, mode, modified, modifystop, name, noredraw, opened, order, origin, place, pointer, prompt, properties, scale, scrollaccel, scrollmargin, self, size, state, style, suppress, tabgroup, tabparent, tabstop, targetformats, theme, tooltip, tooltipenabled, tooltipstyle, tooltiptime, tooltipwidth, translate, visible, where, and $\Delta u d p$.

bitmap

The initial source for the displayed graphical image is contained in the **bitmap** property. The image may be either a device-independent bitmap or an array of encoded color values. If you use color values, each value initially represents one pixel. The value for each color must be the scalar value calculated using the base value primitive function: $256 \pm \text{blue green red}$ (reversed order from the vector representation of red-green-blue).

You use the contents of the **bitmap** property to create the graphic that shows in the control; this graphic is the **image** property. You can scale the bitmap up or down (make it larger or smaller), position it, or tile it (display multiple copies of the bitmap). You create the **image** property from the **bitmap** property using the **style** and **imagesize** properties.

bitmapsizes (read-only)

The **bitmapsizes** property returns the height and width of the most recently set **bitmap** in the scaling units defined for the control. If **bitmap** has not been set or if the **Picture** object (or the form containing it) is closed, **bitmapsizes** is an empty vector. It is a read-only property; you change it only by setting the value of the **bitmap** property.

image

The **image** property contains the graphic image that is wholly or partially visible in the control. It is created initially from the **bitmap**, as explained above, as modified by the **style** and **imagesize** properties. However, you can modify the **image** property dynamically using the **Draw** method or replace sections of the image using the **Paste**, **Pop**, or **Put** methods.

After you draw on the **image** property, you can cover any portion of the graphic without damaging the image. The image is persistent as long as the control is open. That is, if you minimize the form that the control is on, or overlay it temporarily, the modified **image** is still available when you make it visible again. If you close the form, however, the **image** reverts to its initial definition. Similarly, if you reset the **imagesize** property, the **image** reverts to its initial definition.

imagesize

The **imagesize** property defines the size (height and width) of the **image** property in the scaling units defined for the control. These two elements interact with the **style** property and **bitmap** size to create the image. Initially, these values match the size of the **Picture** control. Ordinarily, you set the **imagesize** property before you create the **image** using the **bitmap** and **style** properties.

style

The **style** property controls the placement of the **bitmap** in the **image** property when the two are not the same size. The default value places the **bitmap** in the center of the image. This assumes that the **image** property is larger than the **bitmap**. If the **bitmap** is larger than the **image**, and you do not increase the size of the **image**, the **image** contains only the upper left portion of the **bitmap**. In this case, the rest of the **bitmap** is not accessible even with scrolling.

Values for the Picture Class style Property

Value	Description
Primary Values: Choose only one value	
0	Initial bitmap is unscaled and centered in the image
2	Initial bitmap is tiled to fill the image
4	Initial bitmap is scaled to fit the image
Add-on Values: Choose none, one or both values	
16	Vertical scroll bar
64	Horizontal scroll bar

If you specify 2 in the **style** property, the system fills the **image** property, starting at the upper left corner, with multiple copies of the bitmap; this is called tiling. If the size of the **image** property is not an even multiple of the size of the bitmap, a partial copy fills the right and bottom of the image.

If you specify 4 in the **style** property, the system scales the bitmap proportionally to the size of the **image** property. Note that this can cause the image to be distorted, so that, for example, a circle on the bitmap is oval shaped on the image. If you want to keep the same proportions in a scaled image, the size of the **image** property must be proportional to the size of the bitmap.

With any of these basic styles, you can also specify scroll bars with the **style** property. These scroll bars automatically adjust to the relative sizes of the **image** property and the client area of the Picture control. Specifying scroll bars makes sense only if the image is larger than the client area.

origin

The **origin** property controls the portion of the image that shows in the control. The **origin** property is a two-element numeric vector that specifies the vertical and horizontal coordinates of the upper left corner of the visible portion of the image. These two elements determine what portion of the image the system actually displays in the Picture control. You can set these values under program control, and they change as the user manipulates the scroll bars. Initially, both values are zero.

theme

The **theme** property of a Picture class controls the visual styling theme used for drawing the control. The default value is an empty character vector (' '), which uses the new visual styling for the control. To suppress the new visual styling for the control, set **thememode** to "~".

Event-handler Properties:

onAction, onClose, onDelete, onDestroy, onDragEnter, onDragLeave, onDragover, onDrop, onExit, onExitError, onHelp, onHide, onModified, onMouseDouble, onMouseDown, onMouseDrag, onMouseEnter, onMouseLeave, onMouseMove, onMouseUp, onMove, onOpen, onPaint, onReopen, onResize, onSend and onShow.

Methods:

Close, Copy, Create, Cut, Defer, Delete, Draw, Event, Exec, Get, Help, Hide, Modify, New, Open, Paint, Paste, Pop, Popup, Push, Put, Ref, Resize, Send, Set, SetLinks, and Show.

You can modify the **image** property at any time, whether you initialize the **image** property with a bitmap or not. You can use any of the operations of the **Draw** method to enhance the image. In addition, you can modify rectangular sections of the image using pairs of methods.

Copy and Paste

You can move bitmaps between applications using the Windows clipboard and the **Copy** and **Paste** methods. You should use these methods carefully, because the clipboard is a shared resource. If you use these methods in an application, you could overwrite something the user placed on the clipboard, and you risk having the image overwritten between the time you copy something and then paste it.

Push and Pop

You can move bitmaps within your APL64 application using the **Push** and **Pop** methods. These are equivalent to **Copy** and **Paste** but they use protected memory rather than the clipboard.

Get and Put

If you want to examine or modify individual pixel values, you can use the **Get** and **Put** methods. These allow you to retrieve and return bitmap encoded color values using an APL variable.

When you modify the `image` property with any of these methods, the resulting graphical image is persistent, as explained above, but not permanent. If you save your form, the definition of the `image` property will be the `bitmap` property as projected by the `style` property. If you want to save a modified image, you must either save the `image` property as a separate (potentially very large) variable or replace the `bitmap` property with the `image` property.

Parentage:

Pictures are most often children of a `Form`, `MDIForm`, `Frame`, `Page`, or `CommandBar`. They may also be children of some control objects: `Button`, `Check`, `Combo`, `Edit`, `Label`, `List`, `Option`, `Picture`, `Scroll`, `Spinner`, `Status`, and `Toolbox`. The system allows most classes as children; it specifically prohibits `Form`, `MDIForm`, `CommandBar`, `CommandButton`, `Page`, and `Printer`.

Printer

Prefix: `pr`

Summary:

A utility class that you can use to draw graphics or print text on a printer.

Uses:

When you print under Windows, you work with one page at a time; you can place graphics and text anywhere on the page using the `Draw` method. It is your responsibility to place them within the size limits of the page. You can also place text on the page by using the `Print` method with one or more character vectors, which can include `Unicode` characters. When using the `Print` method, you must ensure that each line fits on the page horizontally; if you generate more lines than fit on a page, the system places additional lines on a new page. Depending on your system settings, the system may print a page as soon as it is complete, but typically you must call the `Close` method on the `Printer` object. When you create a `Printer` object, the system selects the current default printer that is defined in the Windows Control Panel. However, you can assign a different printer. You can also change the default printer under program control.

Regular Properties:

`bin`, `bins`, `caption`, `children`, `class`, `collate`, `copies`, `data`, `def`, `default`, `error`, `events`, `font`, `hdc`, `instance`, `links`, `margin`, `methods`, `modified`, `modifystop`, `name`, `opened`, `orientation`, `page`, `palette`, `paper`, `papers`, `position`, `prntername`, `properties`, `scale`, `self`, `size`, `state`, `suppress`, `translate`, `units`, and `update`.

page

Indicates which page of the print document you are currently working on. When you first open the `Printer`, its `page` property is set to 0. As soon as you use the `Draw` or `Print` method for the first time, the page number changes to 1. When you call the `Eject` method to eject the page or use the `Clear` operation with the `Draw` method, the page is complete. The system increments the setting by 1 if you use the `Draw` or `Print` method on a new page. When you `Close` the `Printer`, the system resets the `page` property to zero.

position

The location on the page where the `Print` method will next cause the printer to print output.

caption

Before you open the `Printer`, you can set its `caption` property to a string that identifies the document to Windows. This is the document title, which the system displays if you open the Windows Print Manager window. Windows also uses, and may require, the `caption` property for any title page that the system produces on a network printer. Note, however, that if you set the `caption` property when a print job is underway, the system closes the print job and opens a new print job with the caption you set. The same thing happens if you set the `hdc` property or use the `Setup` method. You can create a new `Printer` object if you want to have a second print job.

units

The units property works the same way for a Printer as it does for the system object, although the values differ because the number of pixels per unit is different for a printer than it is for the screen. This read-only property is a five-by-two integer matrix that defines scaling units in terms of pixels per scaling unit.

Values for the Printer class units Property

Row	Description
1	Characters (1 sys-font high; 1 sys-font wide)
2	Dialog units (1/8 sys-font high; 1/4 sys-font wide); used by external dialog box editors
3	Points (72 points per logical inch)
4	Twips (1440 twips per logical inch)
5	True pixels. See Note.
6	True pixels. See Note.

Note: Note that the Printer object supports scale mode 5 and 6 but these modes are both True Pixels on the printer. There is no concept of DPI scaling that applies to the Printer in the way it does to Window objects. Scalemode 6 was added to the printer just for symmetry with window scale modes.

The rows of the matrix correspond to the scalemode element of the scale property. The first column of units is the height of scaling element in pixels and the second column is the width of the scaling element in pixels. You can use this property to convert from scaled units to pixels by multiplying by the appropriate row of units, and you can convert from pixels to scaled units by dividing by the units row. When you reference this property, it may look like the following:

```
'p' owi 'units'
83      50
10.375  12.5
8.333333333  8.333333333
0.4166666667  0.4166666667
1          1
```

printername

The printername property is a character vector that tells the name of the current printer. You can set it to any valid printer name or to an empty character vector, which selects the default printer. The list of names of local and attached printers is returned in the first column by the Printers method on the system object.

default

The default property is a Boolean that tells whether the printer currently associated with this printer object is the default printer. If it is not the default printer, you can set this property to 1 to make the current printer the default printer on your system. You cannot set this property to zero.

copies

The copies property is an integer that specifies how many copies of subsequent pages you wish to print. The default value is 1. You must set this property before you place anything on the page to which you want it to apply. When you set copies, the Printer starts a new page with the next Print or Draw command.

collate

The collate property is an integer flag that specifies whether to collate the copies of a multiple-copy print job. If collate is set to 1, when printing multiple copies, the document is printed in page order multiple times. If you set this property to 0, all the copies of page 1 are printed before the first copy of page 2. This may significantly shorten the time required for printing. You can set collate to -1 to have the setting determined by Windows.

orientation

The **orientation** property is a character vector that tells how the subsequent pages will be printed; the allowable values are 'portrait' (the default value) or 'landscape'. You must set this property before you place anything on the page to which you want it to apply. When you set **orientation**, the Printer starts a new page with the next Print or Draw command.

margin

The **margin** property is a four-element numeric vector that specifies the left, right, top, and bottom margins of the printer in the current scaling units for the object. (You may not be able to set the right margin effectively.) If you set the margins outside the printable area, they are set to the maximum or minimum allowed. This means you can set the margins to zero to determine the maximum printable area for your printer. Internally, the margins are stored so that you can change printers, especially to one with a different resolution, while maintaining the same margins.

hdc

This property is a handle to the printer that you can use to make low-level API calls. You can assign zero to the **hdc** property if the printer is closed. This allows you to free the Windows resources associated with the Printer without deleting the Printer.

bins and bin

The **bins** property is a read-only, two-column matrix, consisting of bin number and bin name. A "bin" can be a paper tray or a method of supplying the printer, for example, manual feed. The **bin** property is an integer designating the current paper bin. You can set the latter.

papers and paper

The **papers** property is a read-only, two-column matrix, showing paper size number and paper size name. The **paper** property is an integer designating the current paper size. You can set the latter.

error

The **error** property is a scalar that you use to detect printing errors. If an error occurs during or after the print operation, the **error** property will have a non-zero value.

Event-handler Properties:

onAction, **onClose**, **onDelete**, **onError**, **onModified**, **onOpen**, **onPage**, **onReopen**, and **onSend**.

onPage

The **Page** event occurs each time the system begins a new page but before any printing is done. `□warg` contains the page number of the new page. A new page begins when the sum of the printer's position property plus the height of the character about to be printed exceeds the height of the page.

onError

The **Error** event is triggered whenever an error occurs while printing, for example, if the system runs out of memory. You can assign a non-zero value to `□wres` to specify that the **onError** event handler should ignore the error. This allows you to use the **Draw** method to a context defined by low-level calls by setting the printer **hdc** to the (non-printer) handle of the context.

Methods:

Abort, **Close**, **CloseDoc**, **Create**, **Defer**, **Delete**, **DevMode**, **DevNames**, **Draw**, **Eject**, **Event**, **Exec**, **Modify**, **New**, **Open**, **OpenDoc**, **PageSetup**, **Print**, **PrintDlg**, **Ref**, **Send**, **Set**, **SetLinks**, and **Setup**.

Eject

The **Eject** method ejects a page without closing the print job. It resets the **position** property for the current Printer object to the origin (0 0). The **page** property does not change until you use the **Print** or **Draw** method on a new page.

Abort

The **Abort** method aborts the current print job. The **Printer** object stops printing the current job and discards it. The object remains open, and you can continue printing if you have another job defined.

DevMode

The **DevMode** method allows low-level access to the printer; it returns a character vector that contains the current **DEVMODE** structure for the printer. You can set it using a character vector; the system merges the **DEVMODE** structure represented by the argument with the current settings. You can use this method, for example, to save printer settings from one printer to another or from one APL session to another.

DevNames

The **DevNames** method returns a character vector representing the **DEVNAMES** structure that you can use with `□wcall`. The **DEVNAMES** structure contains the printer name, the printer driver name, and the port name. If you simply want to change the printer name, you can use the **prntername** property.

Setup, PageSetup, and PrintDlg

When you invoke **Setup**, **PageSetup**, or **PrintDlg**, the system displays a standard dialog box where you can select a printer, and set various options for the next print job. In some cases, you can invoke other dialog boxes to set further options. The exact nature of the dialog boxes and the options depend on the printer driver and the operating system. The setup methods return a 1 if the user presses OK, zero if Cancel. **PrintDlg** returns a zero if the user presses Cancel, and an integer 1, 2, or 3 depending on the Print Range.

For **PageSetup** and **PrintDlg**, you can supply optional arguments that allow you to tailor certain settings or limit the options that the user may select. See the *Windows Interface: Methods* chapter for details.

CloseDoc and OpenDoc

The **Printer** object has always prematurely opened the document causing the printer icon to appear in the system tray even before any output is actually written to the printer. This means you cannot open a printer for reference purposes only (such as for print preview) without having that icon appear. In addition when you click on the icon it shows a document named "No Document Name" in progress even though nothing has yet been drawn to the printer. But more problematic is the fact that printing to print-to-file drivers (such as Acrobat (PDF) drivers) causes the user to be prompted to enter a filename. This last undesirable behavior can be suppressed on some print-to-file drivers (such as the Adobe Distiller) by setting the **caption** property to ' ' on the **Create/New** method line when creating the printer (before the implicit **Open** operation can take place). However, for other print-to-file situations this trick does not suppress prompting.

To address this issue, the new **OpenDoc** and **CloseDoc** methods were added to the **Printer** object. These are used to explicitly open and close printer documents.

Invoking the **CloseDoc** method ends the document if already started and suppresses autostart of the document on the next **Open** of object. It can be used in the **New/Create** argument list to suppress automatic opening of the document. Using **CloseDoc** in the **Create/New** argument list is the only way to suppress automatic start of the document when printing. This should always be used when doing print preview to enable checking of printer metrics without actually printing. This is important in order to avoid prompting for print-to-file drivers (such as the Adobe Distiller) when you are not actually printing.

After invoking the **CloseDoc** method you should explicitly start the document before you actually begin printing by calling the **OpenDoc** method. Although the system attempts to automatically start the document on demand when needed (such as when you call the **Print** or **Draw** methods) this may not work for certain printer drivers in all circumstances. Therefore you should make it a practice to use the **OpenDoc** method explicitly to make sure the printer is open before doing any printing operations. This is especially important when the printing is done by methods other than the **Draw** or **Print** methods (such as by directly drawing to

the hdc). The APL Grid printing is an example where the OpenDoc method should explicitly be used if the CloseDoc method was used to suppress automatic starting of the document.

Use the OpenDoc and CloseDoc methods as you once may have used the Open and Close methods to begin and end printing jobs.

The Close method has always had a problem that left the printer device context open after the printer object was closed. This means that the hdc property would continue returning a non-zero value after calling the Close method. It was decided to not fix this problem for compatibility reasons in case applications were using the hdc on a closed printer object. However when the Close method is called on a printer where you have called the CloseDoc method, then APL64 actually releases the device. So the hdc property returns 0 after the Close method when you first do the CloseDoc method.

Both the OpenDoc and CloseDoc methods return a boolean result indicating success or failure of the operation (1 = success; 0 = failure).

You can make the Printer object implicitly invoke the CloseDoc method before opening the document on the Create/New method line. This is done with the compatibility switch:

```
[Compatibility]
PrinterDeferStart=1
```

in the APLW.INI file. This means you will be required to invoke the OpenDoc method before you start printing. By default, the setting is 0, resulting in the old behavior of automatically starting a document when opened unless you override it with invocation of the CloseDoc method on the Create/New method line.

Note: The behavior of setting the caption property and PrintDlg, PageSetup and Setup methods was changed so that they do not automatically open the printer document if the CloseDoc method has been called.

Parentage:

A Printer object does not contain control objects, but it can contain a Timer or an Imagelist. It can be a top-level object or the child of a Form or an MDIForm.

Example:

The following statements draw a circle on the default printer:

```
prt←'prReport' □wi 'New' 'Printer' ('caption' 'Demo Circle')
prt □wi 'scale' 3
prt □wi 'Draw' ('Circle' 300 300 200)
prt □wi 'Draw' ('Font' 'Courier New' 50) ('Text' 'Page 1' 0 0)
status←prt □wi 'Close'
```

You can produce more than one drawing at the same time by opening multiple Printers. For example, the following expressions result in two print jobs in the Print Manager queue.

```
'prGraph' □wi 'New' 'Printer'
'prReport' □wi 'New' 'Printer'
'prGraph' □wi 'Draw' (GenGraph data)
'prReport' □wi 'Print' (GenReport data)
'prGraph' 'prReport' □wi''<'Close'
```

Progress

Prefix: pb



Summary:

A rectangular area that displays a succession of colored blocks or fills the region with a solid "thermometer" style colored fill.

Uses:

Typically, a progress bar is placed at the bottom of a window in the status area or in the center of a popup window to indicate the progress of an operation. Like a scroll bar, you can set the range and current position. Unlike a scroll bar, it has no user interface; this control is for display only. You change the position directly by setting the `value` property, or indirectly using the `Stepit` method.

If you use a block-fill style progress bar, Windows determines the aspect ratio of the blocks. If you increase the height of the bar, the length of an individual block increases. Thus, you do not specify directly the number of blocks it requires to fill the bar; if you want a specific number of blocks, for example, to show increments of ten percent, you must specify a length that corresponds to the height you select.

Windows divides the range you specify according to the number of blocks it can fit in the specified length. When the current position reaches a satisfactory value, the bar displays a new block. The first block appears before you reach the first increment and the last block displays before you reach the maximum value. If you want to show true progress, that is, display a block after a certain amount is completed, you must compensate by adjusting either the values of your range or the relationship of your calculation to the current position.

If you use the solid-colored style fill, the proportion of the filled area corresponds more closely to the ratio of current position to maximum value.

Note: Windows uses the Highlight (selected items) color for the progress bar blocks, so the appearance will vary depending on the user's choices. The background color is the color associated with the 3D objects Item under Appearance within Display in the Windows 2000/XP control panel.

Regular Properties:

`border`, `children`, `class`, `color`, `data`, `def`, `deferexit`, `edge`, `enabled`, `events`, `extent`, `font`, `help`, `helpcontext`, `hwnd`, `imagescale`, `instance`, `links`, `methods`, `mode`, `modified`, `modifystop`, `name`, `noredraw`, `opened`, `order`, `place`, `pointer`, `prompt`, `properties`, `scale`, `scrollaccel`, `scrollmargin`, `self`, `size`, `state`, `style`, `suppress`, `tabgroup`, `tabparent`, `tabstop`, `targetformats`, `theme`, `tooltip`, `tooltipenabled`, `tooltipstyle`, `tooltiptime`, `tooltipwidth`, `translate`, `value`, `visible`, `where`, and `Δudp`.

style

The `style` property for a Progress bar is an integer vector or scalar. The initial value of `style` for any newly created control object is 0. The table below shows the possible values.

Values for the Progress Class style Property

Value	Description
0	Default bar; oriented horizontally, filled with blocks.
1	Smooth fill. The "fill" color that shows the progress is solid.

Value	Description
2	Vertical bar; filled with blocks; progress starts from the bottom.
3	Vertical bar with solid fill coloring like a thermometer.

theme

The theme property of a Progress class controls the visual styling theme used for drawing the control. The default value is an empty character vector (' '), which uses the new visual styling for the control. To suppress the new visual styling for the control, set thememode to "~".

value

A four-element, integer vector that indicates the progress state. The table below shows the definitions of the elements. When you set the value property, you can omit trailing elements without changing their values. These values must be non-negative integers less than 65536; the maximum cannot be less than the minimum.

Elements for the Progress Class value Property

Number	Definition of the element
1	Current Position (Default 0)
2	Maximum Position (Default 100)
3	Step Increment (Default 10)
4	Minimum Position (Default 0)

Event-handler Properties:

onAction, onClose, onDelete, onDestroy, onDragEnter, onDragLeave, onDragover, onDrop, onExit, onExitError, onHelp, onHide, onModified, onMouseDown, onMouseDrag, onMouseEnter, onMouseLeave, onMouseMove, onMouseUp, onMove, onOpen, onPaint, onReopen, onResize, onSend, and onShow.

Methods:

Close, Create, Defer, Delete, Draw, Event, Exec, Help, Hide, Modify, New, Open, Paint, Popup, Ref, Resize, Send, Set, SetLinks, Show, and Stepit.

Stepit

Advances the Current Position of a progress bar and redraws it. You cause this behavior only by using this method. There is not a comparable system- or user-generated event. If stepping causes the current position to go out of range (above the Maximum Position or below the Minimum Position), the Current Position wraps around.

If you provide no argument, the current position (first element of the value property) is advanced by the step increment (the third element of value). You can provide an integer argument to this method that represents the amount by which you want the current position to change.

Parentage:

Progress bars are most often children of a Form, Frame, Page, or CommandBar. They may also be children of some control objects: Button, Check, Combo, Edit, Label, List, Option, Picture, Scroll, Spinner, Status, and Toolbox. Progress objects can contain only Imagelists, Timers, or hidden Menus.

Example:

The example below shows different styles and methods of advancing a Progress bar from empty to complete. The difference between the first two bars is more apparent if they each have exactly ten blocks. You may have to adjust the length (or height) of the bar to make this true on your monitor. The first bar has the system displaying discrete blocks as the value increases in small increments. The second bar uses the

Stepit method to display a block when the value reaches multiples of ten percent. The third bar has the system displaying the solid fill as the value increases.

```
'fmForm.Value' □wi 'New' 'Label' ('where' 1 3 1 9) ('caption' 'Value')
```

```
pbv ← 'fmForm.pbValue' □wi 'New' 'Progress' ('where' 2 3 2 25.5)
pbv □wi 'value' 0 100 10 0
```

```
pbs ← 'fmForm.pbStep' □wi 'New' 'Progress' ('where' 5 3 2 25.5)
pbs □wi 'value' 0 100 10 0
```

```
pbd ← 'fmForm.pbSolid' □wi 'New' 'Progress' ('where' 8 3 2 25.5)
pbd □wi 'Set' ('value' 0 100 10 0) ('style' 1)
```

```
▽ advance;inc;delay
[1] 'fmForm.pbValue' □wi 'value' 0
[2] 'fmForm.pbStep' □wi 'value' 0
[3] 'fmForm.pbSolid' □wi 'value' 0
[4]
[5] A Increment value from 1 to 100 by 1
[6] :for inc :in 1100
[7]     'fmForm.pbValue' □wi 'value' (inc)
[8]     'fmForm.pbSolid' □wi 'value' (inc)
[9]     'fmForm.Value' □wi 'caption' (inc)
[10]
[11]     :if (10|inc)=0
[12]         'fmForm.pbStep' □wi 'Stepit'
[13]     :endif
[14]     delay←□dl .2
[15] :endfor
▽
advance
```

RichEdit



Prefix: red

Summary:

A specialized Edit control that provides extended capability for displaying formatted text.

Uses:

You can use a RichEdit control when you want to display formatted output or you want the user to type formatted input. A RichEdit control is similar to an Edit control except that it supports text attributes such as bold characters, underlines and changes of font and color for parts of the text, and the RichEdit allows paragraphs to be formatted with bullets or numbers, indents, tab stops, and justification.

You can specify various combinations of formatting under program control. To allow a user to change these attributes, you must provide a program interface as there is nothing inherent in the control to perform these functions. The control will, however, handle drag-and-drop internally. This means that a user can drag text from one location within the control to another. It also means that a user can drag text from another application and drop it into a RichEdit control, even if the `targetformats` property is not set.

Note: The RichEdit formatting makes extensive use of the `selection` property. It may be useful to set the `translate` property for this control to a non-default value to ensure the accuracy of the formatting. See the description of the `translate` property for more information.

Regular Properties:

`border`, `canpaste`, `canredo`, `canundo`, `children`, `class`, `color`, `data`, `ddeApplication`, `ddeltem`, `ddeMode`, `ddeTimeout`, `ddeTopic`, `def`, `deferexit`, `edge`, `enabled`, `events`, `extent`, `help`, `helpcontext`, `hwnd`, `imagescale`, `instance`, `keys`, `limit`, `links`, `methods`, `mode`, `modified`, `modifystop`, `msversion`, `name`, `noredraw`, `opened`, `order`, `place`, `pointer`, `prompt`, `properties`, `range`, `redoname`, `rtf`, `scale`, `selalign`, `selbullet`, `selcolor`, `selection`, `self`, `selfont`, `selfontall`, `selindents`, `selnumstart`, `selnumstyle`, `selnumtab`, `selpargall`, `selrtf`, `selstyle`, `selstyleall`, `seltabs`, `seltext`, `size`, `state`, `style`, `suppress`, `tabgroup`, `tabparent`, `tabstop`, `text`, `theme`, `tooltip`, `tooltipenabled`, `tooltipstyle`, `tooltiptime`, `tooltipwidth`, `translate`, `undolimit`, `undoname`, `visible`, `where`, `zoom`, and `Δudp`.

style

An integer vector that contains selected values from the following list or an integer scalar that is the sum of the selected values. The initial value of `style` for any newly created control object is 0; if you want multiple lines in the control, you must select 4. If you want the Enter key to generate a new paragraph in the control, you must add 8192 as well. Without the latter, the user must press Ctrl+Enter to generate a new paragraph. You can select any meaningful combination of other values for `style`; many of them require that you have a multiline control.

Values for the RichEdit Class style Property

Value	Description
0	One-line; left-justified text (the default).
1	Centered text (only if 4 is also coded).
2	Right justified text (only if 4 is also coded).
4	Multiline
8	Inhibit automatic vertical scrolling (only with 4).
16	Vertical scroll bar (only if 4 also coded).
32	Inhibit automatic horizontal scrolling.
64	Horizontal scroll bar.
1024	Do not hide selection when tabbing out of field.
2048	APL entry field; accommodates APL symbols.
4096	Read only.
8192	Allow an Enter character as part of the text (only if 4 is also coded).
16384	Do not select text when tabbing back to control. (Often used with 4 and 1024.)
32768	Disable automatic word selection when highlighting with mouse (that is, allow partial words at beginning or end of mouse drag).
65536	Disable, rather than remove, unneeded scroll bars (with 16 and/or 64).
131072	Allow a Tab character as part of the text.
262144	Advanced typography.

rtf

A character vector that contains the text and associated control codes for the RichEdit control. You can set the `rtf` property on a closed RichEdit control, but you cannot set the `text` property until the object is opened.

text

A character vector that contains the text contents of the RichEdit box. All of the control codes are automatically removed in this property.

selection

A two-element integer vector that contains the number of characters before the first selected character and the number of characters of highlighted text. You can specify `-1` for the first element to place the cursor at the end of the field. You can specify `-1` for the second element; that selects everything after the insertion point to the end of the field.

range

A two-element integer vector that describes what lines of a vertically scrolling, multiline RichEdit control are visible in the edit window. The first element is the origin-one line number of the top row; the second element is the number of visible lines (which can vary with font size). When you set this property, you can specify only the first element; the second element is read-only.

seltext

A character vector of the selected text without RichEdit control codes. You can query or replace text using this property.

selrtf

A character vector of the selected text with RichEdit control codes. You can use this to query or replace formatted text.

msversion

An integer scalar (1, 2, or 3) that tells what version of the control is on the host machine. Microsoft provides at least three different versions of the RichEdit control. Version 3 is required for some features; for example, numeric or alphabetic bullet formats. If you have version 2, the control may store the formatting values but not display the effect; in this case, if you move the application to another machine that has a higher version, the effect does display. If a user has version 1, some property values may not be retained, and certain restrictions apply; for example, you are limited to a single Undo action.

theme

The theme property of a RichEdit class controls the visual styling theme used for drawing the control. The default value is an empty character vector (`' '`), which uses the new visual styling for the control. To suppress the new visual styling for the control, set `thememode` to `"~"`.

RichEdit Properties that Affect Editing Actions

The RichEdit control allows editing actions to be undone and redone using the Undo and Redo methods or the Ctrl+Z and Ctrl+Y keystrokes. You can also create menu items for these actions. This set of properties allow you to monitor and control this facility, enable and disable menus for undoing and redoing, as well as other standard actions such as cutting and pasting.

canundo

A Boolean flag indicating if the last action can be undone with the Undo method. You can set this property to zero to disable the Undo method; you cannot set it to 1.

undoname

A read-only character vector, when the control is open, that indicates the type of action that can be undone. Possible values include: `typing`, `delete`, `cut`, `paste`, `dragdrop`, `unknown`, or an empty vector. The value is `'unknown'` for character and paragraph formatting operations; it is empty if `caundo` is zero.

canredo

A read-only Boolean flag indicating if the last action that was undone can be redone with the Redo method.

redoname

A read-only character vector, when the control is open, that indicates the type of action that can be redone. The possible values are the same as for undoname.

undolimit

You can set a limit for the maximum number of actions stored in the undo queue. Setting this property to zero disables the undo facility. A version 1 RichEdit can undo only one action.

canpaste

A read-only Boolean flag that tells whether the Windows clipboard has anything that can be pasted in a RichEdit control. Note that such content may be inappropriate for a RichEdit control; for example, it could be an image file. This flag does not guarantee text.

zoom

This property allows you to magnify or reduce the display of text in a RichEdit. To enlarge the display you can enter any number greater than 1 up to, but not including 64. To reduce the display, you can enter a number less than 1; theoretically you can go to 1/64, but small numbers may have a visible effect only when the font size is very large. For any font, there may be threshold effects. Changing font size is not effective until you reset zoom to 1. This facility requires a version 3 RichEdit.

RichEdit Properties that are Paragraph Formatting Styles

Paragraph formatting styles apply to one or more paragraphs that are selected or partially selected when you set these properties. You can check whether multiple paragraphs have the same paragraph styles by referencing the `selfpargall` property.

selalign

A character vector describing alignment of the selected text. You must include 32 in the style property (inhibit automatic horizontal scrolling) for this property to be effective. There are two categories of values you can use. The valid values that have a visible effect on the screen are 'left', 'right', or 'center'.

Other values are preserved with the paragraph formatting; these values are compatible with Microsoft's Text Object Model (TOM) interface, and they will have their effect if you copy the formatted text to an application that uses such an interface. Two of the values provide block margins; that is, text that is both left and right justified. The 'interletter' value accomplishes this by adjusting the spacing between letters; the 'justify' value increases space between words. Other valid values are: 'scaled', 'glyphs', and 'snapgrid'; you can find documentation on these effects in your application that uses the TOM interface.

selbullet

A character vector indicating if the selected text is bulleted ("none" or "bullet"), or integer values that specify various sequential paragraph formats: zero is equivalent to "none"; 1 specifies the standard bullet (•) for each paragraph. Other values are shown in the table below:

<i>selbullet numeric value</i>	2	3	4	5	6
<i>description</i>	arabic numerals	lowercase letters	uppercase letters	lowercase roman numerals	uppercase roman numerals
<i>selnumstyle setting</i>	<i>default</i>	1 .	1	(1)	1)
<i>visual effect</i>	1)	a.	A	(i)	I)
	2)	b.	B	(ii)	II)
	3)	c.	C	(iii)	III)

Sequential values require msversion 3; for version 2, the values are preserved with the paragraph formatting but do not display (hence may work on another machine). For version 1, the values may not be recognized.

Alternatively, you can define the value using character strings; the values corresponding to 2 through 6 are: 'arabic', 'lcletter', 'ucletter', 'lcroman', and 'ucroman'.

selnumstart

A non-negative integer scalar specifying the starting value for sequential paragraphs. If you select letters, setting this property to zero or 1 starts with "A" or "a"; if you set this property to zero, roman numerals start with a blank.

selnumstyle

A character vector showing the formatting style for numbered or lettered sequential paragraphs; the default value is: '1)'; other valid values are: '(1)', '1.', '1', and '' (an empty vector).

If sequential paragraph formatting is not specified in **selbullet**, this property is an empty vector. If you have **selbullet** set to a sequential paragraph numbering style, an empty vector in this property suppresses the numbering for the paragraphs to which it is applied, but does not stop the sequence for all the following paragraphs to which the **selbullet** value and one **selnumstart** value apply. This allows you to have a multi-paragraph exposition for one number or letter in your sequence and continue automatically with the next number or letter instead of having to start a new sequence with a hard-coded, non-standard starting value.

selnumtab

A numeric scalar specifying the minimum distance between the start of a bullet or the letter or number for sequential paragraphs and the start of the text. This value affects only the first line of a bulleted or sequential paragraph. The second value of the **selindents** property determines subsequent lines. If both are set, the greater of the two applies to the first line. If **selbullet** is zero, this property has no effect.

selindents

A 3-element numeric vector of left indent, bullet indent and right indent, measured in current scaling units. The left indent, measured from the left margin, affects the entire paragraph, including any bullet or sequential paragraph format.

The bullet indent is measured from the left indent and affects the text of a paragraph in various ways. If you have no bullets, this value affects all lines after the first (creating a hanging indent). If bulleting or a sequential paragraph format is specified, this value determines the minimum distance for the text from the left indent (the start of the bullet, letter or number). If **selnumtab** is greater than this value, **selnumtab** determines the minimum for the first line and bullet indent the remaining lines.

Right indent is measured from the right margin of the control. This value and any effect on second or subsequent lines depend on the **style** property including 32 (suppress automatic horizontal scrolling).

seltabs

A numeric vector of up to 32 tab positions, measured in current scaling units, for the selected text. Note that, unless you include 131072 in the **style** property, a user cannot insert a tab by pressing the Tab key; this moves the focus to the next control. A user can insert a tab by pressing Ctrl+Tab.

You can also specify a type of tab stop with the advanced typography **style** (262144) set. Without that setting, the **seltabs** property is a vector of values specifying the location of tab stops, measured from the left margin. With the style setting you can specify **seltabs** as a nested vector of two- or three-element vectors. The first element is the location setting, as with all styles. The second element of each nested vector specifies the type of tab stop: 0=left justified (normal); 1=centered; 2=right justified; 3=decimal.

RichEdit Properties that are Character Formatting Styles

Character formatting styles apply to the text that is selected when you set these properties.

selcolor

A numeric scalar or three-element vector that specifies the color of the selected text. You specify a color in the same manner as one element of the **color** property.

selstyle

A five element numeric vector indicating styling for the selected text. The elements represent bold, italic, underline, strikethrough, and protected, in that order. If the element is 1, the style is applied to the selected text. If it is zero, the text does not have that style. You can specify -1 for an element to leave it unchanged when you specify values for other elements. When multiple characters are selected, referencing the **selstyle** property refers to the characteristics of the first one. You can check whether all the characters in a block of selected text have the same styles by referencing the **selstyleall** property.

If a user attempts to edit text that has the protected style, the system triggers the **onProtected** event handler. If you want to prohibit the editing, you must set `onProtected=0` in that handler.

selfont

A five-element nested vector that specifies the typeface, size, and (partial) style of the font that you want the system to use for the selected text. A RichEdit control does not have a **font** property, so you must set **selfont** for any text for which you do not want the default font.

You can specify **selfont** as a five-element vector similar to the **font** property plus an offset value (for subscripts and superscripts). However, you cannot specify formatting (bold, italic, underline, etc.) or quality in the third element. Trailing elements that you do not want to change can be omitted. You can specify other elements you do not want to change as `0`.

You can apply the font to individual characters or contiguous blocks. If you select multiple characters and reference this property, the system returns the values of the first one. You can check whether all the characters in a block of selected text have the same font values by referencing the **selfontall** property.

The elements of the **selfont** property are:

- *fontName* is the logical name of the font. If you specify `0`, the system selects a font based on the *fontStyle* value.
- *fontSize* is the logical size of the font (in the current scale units as defined by the **scale** property for this object). You can specify the height or the height and average width. Note that the system interprets *fontSize* differently for this property than it does for the **font** property of other objects, specifically the Edit control. See the description of the **font** property for more details.
- *fontStyle* is a numeric scalar or one element vector that indicates how the font is to be styled. The value is the sum of one value for Pitch and one value for Family. You cannot set formatting styles in the **selfont** property. If you set one of the **font** property formatting styles (1, 2, 4, or 8) in this property, an error is signaled. The values you can use are shown below.

***fontStyle* elements for the selfont property**

Pitch value	Description	Family value	Description
0	Default pitch	0	Do not care
32	Fixed pitch	128	Roman
64	Variable pitch	256	Swiss
		384	Modern
		512	Script
		640	Decorative

- *fontCharset* is the name of the character set (selection and order of symbols). Choose one of the following: 'ansi', 'oem', 'symbol', 'default', or an integer.

- *offset* is the distance a character is placed above or below the baseline, measured in current scaling units for this object. If you specify a positive value, the text is drawn above the baseline (superscript); if you specify a negative value, the text is drawn below the baseline (subscript).

Note: After you set a text formatting property, when a user types within the block of characters, including following the last character of the block, the new characters display the format characteristics. If you set a text formatting property without selecting any text, and the user starts typing at the point where the selection cursor was when you set the property, the new characters display the format characteristics. If you set a text formatting property without selecting any text, and the user moves the cursor within the control before typing a character, for example by using an arrow key, the format setting is lost.

selfontall, selstyleall, and selpargall (read-only)

These properties return values that tell whether all the text that is selected has the same characteristics. They return four-element, five-element and nine-element Boolean vectors, respectively. See the Windows Interface: Properties chapter for more information.

Event-handler Properties:

onAction, onChange, onClose, onDelete, onDestroy, onExit, onExitError, onFocus, onHelp, onHide, onKeyDown, onKeyPress, onKeyUp, onLimit, onMemory, onModified, onMouseDouble, onMouseDown, onMouseDrag, onMouseEnter, onMouseLeave, onMouseMove, onMouseUp, onMove, onOpen, onPaint, onProtected, onReopen, onReqResize, onResize, onScroll, onSelChange, onSend, onShow, onUnfocus, and onUpdate.

onProtected

This event is triggered whenever an attempt is made to change text that has a selstyle of protected. If you want to prohibit the editing, you can set `onwres←0` in this event handler.

onReqResize

When the contents of a RichEdit control are changed such that it would require a different size control, this event is fired. It returns a four-element vector showing the location, calculated height and current width. You can use this as an argument to the where property to resize the control. You can invoke this event by calling the ReqResize method.

Methods:

CharToLine, Close, Copy, Create, Cut, Defer, Delete, Draw, Event, Exec, Focus, Help, Hide, LineToChar, Modify, New, Open, Paint, Paste, Popup, Redo, Ref, ReqResize, Resize, ScrollCaret, Send, Set, SetLinks, Show and Undo.

Undo

The Undo method causes Windows to undo the last editing action taken either by the user or under program control. You can create a menu item that allows the user to trigger this method and use the canundo property to determine when to enable it.

ScrollCaret

The ScrollCaret method moves the selection cursor into the visible portion of the control.

CharToLine

The CharToLine method takes a scalar integer argument that is the position of a character in the control, for example, the first element of the selection property. It returns the line number, in origin zero, on which that character appears. You can use `-1` to find the line number of the last line in the control.

LineToChar

The LineToChar method takes a scalar integer argument that is the origin-zero line number in the control. It

returns the position of the first character on that line; that is, the number of characters that precede the character in the `text` property.

This is the same value you would find in the first element of the `selection` property if you selected that character or placed the selection cursor in front of it. You can use `-1` to find the first character of the line the cursor is on.

Parentage:

RichEdit boxes are most often children of a `Form`, `Frame`, `Page`, or `CommandBar`. They may also be children of some control objects: `Button`, `Check`, `Combo`, `Edit`, `Label`, `List`, `Option`, `Picture`, `Scroll`, `Spinner`, `Status`, and `Toolbox`. RichEdit objects can contain `Timers`, `Imagelists`, or hidden `Menus`.

Example:

```
'f' [wi 'New' 'Form' 'Hide' ('size' 8 40)
[wsself+ 'f.red' [wi 'New' 'RichEdit'
[wi 'style' 4 16384
[wi 'where' 0 0 6 36
[wi 'seltext' ('Einstein says:',[tclf,'E=mc2'])
[wi 'selection' 0 -1      A select all
[wi 'selfont' 'Arial' 1.5
[wi 'selstyle' 0 1      A italic
[wi 'selection' 14 1      A E
[wi 'selcolor' 255 0 0    A red
[wi 'selection' 18 1      A 2
[wi 'selfont' (0 1 0 0 .5) A superscript
[wi 'selection' 0 0      A select none
0 0 [f' [wi 'Show'
[wgive 0
0 0 [p[dl 3
[wi 'selection' 14 5      A select E=mc2
[wi 'seltext' ('develop at',[tcnl,'the speed of light')
```

Scroll



Prefix: sb

Summary:

A graphical device you use to select or indicate position when there is too much information to display in the space available.

Uses:

Typically, you size a scroll bar to extend the entire height or width of the window, and you allow the user to drag the thumb (scroll box) or click on the bar or its arrows to move the thumb. If you click on the bar, the thumb moves toward the mouse arrow; if you hold the left mouse button down while the mouse is on the bar, the thumb moves until it reaches the mouse arrow. If you click on one of the scroll bar arrows, you can move the thumb to its farthest extent.

You define event handlers to make your application react to the movement of the scroll bar thumb. For example, if you have more information than fits in the window, you can move the location of the cursor proportionally to the total amount of information that can be displayed in the window. This causes the visible portion of the information to scroll in the generally accepted fashion.

You can also use Scroll as an analog input and output device. You use these when the relative position along a scale is more meaningful than the numeric values along the scale. Examples are a sound volume control or a color intensity value. See the DEMOBALL workspace for a demonstration function.

Regular Properties:

border, children, class, color, data, def, deferexit, edge, enabled, events, extent, help, helpcontext, hwnd, imagescale, instance, links, methods, mode, modified, modifystop, name, noredraw, opened, order, place, pointer, prompt, properties, scale, scrollaccel, scrollmargin, self, size, state, style, suppress, tabgroup, tabparent, tabstop, targetformats, theme, tooltip, tooltipenabled, tooltipstyle, tooltiptime, tooltipwidth, value, visible, where, and Δudp .

style

An integer vector that contains selected values or an integer scalar that is the sum of the selected values. The initial value of style for any newly created control object is 0. The table below shows the possible values; use one of the primary values plus any of the add-on values concerning alignment.

Values for the Scroll Class style Property

Value	Description
Primary Value: Choose only one value	
0	Vertical
1	Horizontal
2	Size box (grayed area where scroll bars meet)
Add-on Values: Choose any of the values	
0	No alignment
4	Bottom/right alignment
8	Top/left alignment

When you use alignment, the system adjusts the width of a vertical scroll bar and the height of a horizontal scroll bar to the standard system scroll-bar size and aligns the bar with the top, bottom, left, or right edge of the “where” rectangle.

theme

The theme property of a Scroll class controls the visual styling theme used for drawing the control. The default value is an empty character vector (' '), which uses the new visual styling for the control. To suppress the new visual styling for the control, set thememode to " ~".

value

A five-element vector that indicates the scroll state. The first four elements must be integers; the fifth element must be 1 or zero. The table below shows the definitions of the elements.

Elements for the Scroll Class value Property

No.	Definition of the Element
1	Current position (default 0).
2	Maximum position (default 100).
3	Page size increment (default 1).
4	Minimum position (default 0).
5	Enable tracking of scroll box.

When you set the value, you may omit trailing elements without changing the values of those property elements. The page size increment specifies the change in the current position when the user presses the PageUp or PageDown key. The maximum plus the page size increment minus the minimum must be less

than 2*31. However, when you use very large numbers it is very difficult to control the bar with precision. When the fifth element of value is set to 1, a user who drags the thumb (scroll box) or causes it to scroll generates a string of Change events. If this value is zero, one Change event occurs when the user releases the mouse button to stop scrolling.

Event-handler Properties:

onAction, onChange, onClose, onDelete, onDestroy, onDragEnter, onDragLeave, onDragover, onDrop, onExit, onExitError, onFocus, onHelp, onHide, onKeyDown, onKeyUp, onModified, onMouseDouble, onMouseDown, onMouseDrag, onMouseEnter, onMouseLeave, onMouseMove, onMouseUp, onMove, onOpen, onPaint, onReopen, onResize, onSend, onShow, and onUnfocus.

Note that the Scroll bar captures the mouse when you press the left mouse button (to control the slider). Therefore, onMouseDrag and onMouseUp event handlers are valid only when the right (or center) mouse button is pressed or when you press the left mouse button while off the control and move the mouse over the Scroll bar.

Methods:

Close, Create, Defer, Delete, Draw, Event, Exec, Focus, Help, Hide, Modify, New, Open, Paint, Popup, Ref, Resize, Send, Set, SetLinks, and Show.

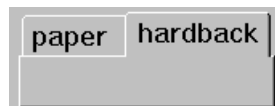
Parentage:

Scroll bars are most often children of a Form, Frame, Page, or CommandBar. They may also be children of some control objects: Button, Check, Combo, Edit, Label, List, Option, Picture, Scroll, Spinner, Status, and Toolbox. The system allows most classes as children; it specifically prohibits Form, MDIForm, CommandBar, CommandButton, Page, and Printer. You can also have a scroll bar in a Combo box, List, Edit control, or RichEdit control, but that scroll bar is not a separate APL object.

Example:

```
'fmForm.scPage' □wi 'New' 'Scroll'  
'fmForm.scPage' □wi 'style' 0 4
```

Selector



Prefix: s

Summary:

A rectangular area that serves as a container where you can place Page objects. A Selector allows very complex sets of choices without using cascading menus.

Uses:

A Selector object contains one or more Page objects, each of which can contain a set of controls. When the user clicks on the tab representing one of the contained Page objects, children of that page are shown and controls that are children of other Page objects in that Selector are hidden.

Regular Properties:

border, children, class, color, data, def, deferexit, edge, enabled, events, extent, fixedtabsize, font, help, helpcontext, hwnd, imagescale, imagelist, instance, links, methods, mode, modified, modifystop, name, noredraw, opened, order, padding, pages, place, pointer, prompt, properties, scale, scrollaccel, scrollmargin, self, size, state, style, suppress, tabgroup, tabparent, tabrows, tabstop, targetformats, theme, tooltipenabled, tooltipstyle, tooltiptime, tooltipwidth, translate, value, visible, where, and *Δudp*.

style

An integer vector that contains selected values or an integer scalar that is the sum of the selected values. The table below shows the possible values. Use one of the primary values to determine whether tabs, buttons, or neither are used to select child **Page** objects. If you specify neither, you must provide a mechanism for the user to access another page; for example, a Next button. Include any of the add-on values to modify the presentation or behavior of the object.

Values for the Selector Class style Property

Value	Description
Primary Value: Choose only one value	
0	Tabs
1	Buttons
2	NoTabs or Buttons
Add-On Values: Choose any of the values in meaningful combinations	
8	Multiline: Use multiple rows to display all tabs at once
16	RaggedRight: Do not increase size of tabs to fill entire width of control (only with 8)
32	FixedWidth: Make all tabs the same width
64	ForceImageLeft: Left justify the image, but leave the text centered (only with 32)
128	ForceLabelLeft: Left justify the image and the label (only with 32)
256	Limit focus: Tabs accept focus from tab key, but not from mouse click. (See note 1).
512	Immediate focus: Tabs accept focus on first mouse click; (not with 256). (See note 1.)
1024	Tabs on side (left): Tabs are oriented vertically on left edge with caption running up.
2048	Tabs on bottom or right: Tabs on bottom of selector or, with 1024, on the right edge.
4096	Keep lines intact: Forces multiline display; keeps lines in the same order. (See note 3.)
16384	Flat buttons: Gives the flat button appearance, like command buttons (only with 1).
32768	No separators: Do not put separators between flat buttons (only with 1 and 16384).
65536	Hot track: Change tab appearance when mouse cursor is moved over it. (See note 5.)

Note 1: Changing the Behavior of Focus: In a default selector, when the user clicks on a tab, the system displays the page associated with that tab and the focus moves to a control on the page. If the user clicks a second time, the tab can get the focus and the arrow keys move the focus from tab to tab. The selector usually is one of a group of siblings on its parent and can also receive the focus when the user presses the Tab key an appropriate number of times.

Two mutually exclusive **style** values change this behavior. If you set the **style** property to include 256, clicking the mouse does not cause the selector tab to get the focus at all. The user can still use the Tab key to reach the selector tab. Alternatively, if you set **style** to 512, the selector tab receives the focus on the first mouse click, and the focus does not move automatically to a control on the selected page.

Note 2: Changing the Orientation of Tabs: In a default selector, the tabs appear at the top. You can orient the tabs on the side by including 1024 in the **style** property. The tabs are oriented vertically on the left side with the captions running upward. If you include 2048 in **style**, the tabs are oriented on the bottom or right. That is, if you add 2048 to a **style** that has tabs on top, the tabs are oriented on the bottom; if you include both 1024 and 2048, the tabs are oriented vertically on the right edge of the selector with the caption running down.

Caution: When you change the orientation of the tabs from the top to a side, the usable area of the selector changes, because the space required for the tabs is subtracted from the width of the selector rather than the height. In addition, the selector automatically uses multiple rows of tabs if necessary to show all the captions. You may have to resize the selector to make room for your entire page.

Warning: If you have button appearance for tabs (style 1), and you also place them on the side, the underlying control allows much more space than necessary for the button tabs. Thus the placement of the page is awkward, and your page may be clipped unless you have a small page on a large selector.

Note 3: Changing the Multiline Behavior of Tabs: In a default selector, if you create too many pages or use long captions, not all the tabs can be visible at one time on a single row. If you include 8 in the style, the selector displays all the tabs by using multiple rows. When you have multiple rows of tabs, and when the user clicks on a tab to select a page, the row of tabs that includes the one selected moves so that it is the row closest to the contents of the page. As the user clicks among various tabs, the rows can move up and down (or left and right).

If you include 4 0 9 6 in style, the selector automatically becomes a multiline selector, but the rows stay in the same order; if there are any rows of tabs between the contents of the page and the tab the user clicks, those rows move to the far side of the selector so the clicked tab is closest to the contents of the page. Instead of rows moving up and down or left and right in relation to each other, one or more rows of tabs may jump from one side of the selector to the other.

Note that if you have multiple rows of buttons instead of tabs, the rows of buttons do not change when you select a new page. Thus, this style has no effect in conjunction with style 1.

Note 4: Changing the Appearance of Tab Buttons: If you have buttons instead of tabs for a selector (style 1), you can change those buttons into flat buttons (like command buttons) by including 1 6 3 8 4 in style. By default, the flat buttons are separated a visible vertical bar like a menu separator or the separators on command bars. If you also include 3 2 7 6 8 in style, there are no separators.

Note 5: Providing Hot Track Highlighting: You can provide a highlighting behavior for a selector that is similar to the behavior of a command bar. When the user moves the mouse cursor over a tab or a tab button, the system automatically provides an indication of its potential. If you have the default tabs or style 1 buttons, the system renders the caption in the highlight color. (This may have minimal noticeable effect, depending on the Control Panel settings on the user's system.) If you have flat buttons (style 1 6 3 8 4), the button gives a raised appearance when the mouse cursor moves across it. Unlike the command bar, there is no HotChange event, however.

value

The origin 1 index of the currently selected Page, or 0 if no Page is selected. If a Page is selected, this is equivalent to the order property of the Page.

fixedtabsize

A two-element numeric vector describing the height and width of each tab, in the currently selected scaling units for the Selector object. If FixedWidth style (3 2) is not selected, only the height is used. If you specify an empty numeric vector, Windows determines the size.

tabrows (read-only)

A scalar integer indicating the number of rows of tabs in the object; this is useful only when Multiline style (8) is selected. If there are no pages, or if NoTabs style (2) is selected, this is 0; otherwise, if Multiline style is not selected, this is 1.

padding

A two-element numeric vector specifying the (vertical and horizontal) padding, in pixels, around the image and label on each tab.

pages (read-only)

A nested vector of character vectors containing the last part of the name for each child of class Page. The

order of the Pages in this list is determined by the order property of the Pages. This is the same as the order in which the tabs on the Selector are displayed.

imagelist

A character vector containing the name of the Imagelist object from which to draw images to place on the Page tabs. You specify the index to the image you want to display on each Page in the imageindex property of the Page.

color

The color property has no direct effect on the Selector; if you set it, child pages inherit the settings. The system uses the color associated with 3D objects in the Windows control panel for the background color of the tabs; it uses the color associated with 3D objects text in the Windows control panel for the text color.

theme

The theme property of a Selector class controls the visual styling theme used for drawing the control. The default value is an empty character vector (' '), which uses the new visual styling for the control. To suppress the new visual styling for the control, set thememode to "~".

Event-handler Properties:

onAction, onChange, onClose, onDelete, onDestroy, onDragEnter, onDragLeave, onDragover, onDrop, onExit, onExitError, onFocus, onHelp, onHide, onKeyDown, onKeyUp, onModified, onMouseDouble, onMouseDown, onMouseDrag, onMouseEnter, onMouseLeave, onMouseMove, onMouseUp, onMove, onOpen, onPaint, onReopen, onResize, onSend, onShow, onUnfocus, and onValidate.

onChange

For a Selector object, a Change event occurs when a new page is selected. The Change event occurs after the current page is changed, but before the Show event occurs for the newly selected page. When the event occurs, `EventArgs` contains a two element vector with the origin 1 index of the new and previous pages (in that order).

onValidate

This event handler fires when a new page of a Selector object is selected, when the current page is closed, or when the Validate method is invoked. This event occurs before the Change event. The onValidate handler for the currently selected page is executed first. If it succeeds, the onValidate handler for the Selector is called. `EventArgs` contains two items: a numeric code that indicates what caused the validation to occur and a nested vector of the newly selected page and the previously selected page. The code is 2 if the user caused the page to change by mouse, 1 if by keyboard input, or zero if the reason is unknown. The code is negative if the program induced the change; -1 for the Validate method; -2 if the value property was changed; or -3 if the Close method was applied to the currently selected page. You can set `EventArgs` to zero to prevent the change.

Methods:

Close, Create, Defer, Delete, Draw, Event, Exec, Focus, Help, Hide, HitTest, Modify, New, Open, Paint, Popup, Ref, Resize, Send, Set, SetLinks, Show and Validate.

Parentage:

Selectors are most often children of a Form, MDIForm, Frame, Page, or CommandBar. They may also be children of some control objects: Button, Check, Combo, Edit, Label, List, Option, Picture, Scroll, Spinner, Status, and Toolbox. They can contain Page, Timer, and Imagelist objects. A Selector can also contain a Menu object, not as a visible menu bar, but for the purpose of assigning a shortcut key to the Selector itself.

Spinner

Prefix: spin



Summary:

A pair of arrows pointing in opposite directions that allow the user to increment or decrement a value or step through a list.

Uses:

You specify a range of contiguous values through which the spinner operates. If the user clicks on an arrow, the value changes up or down by one. If the user holds an arrow down, the values change in larger increments until the user releases the arrow. Typically, a spinner is paired with an object that displays a result each time the user clicks up or down on the spinner.

You can specify event handlers to display the result or you can formally associate the spinner with another control and have the system provide automatic updates. If you specify the event handlers, you can format the numeric value directly, or, for example, you can use the value as an index into a list.

If you want to associate a spinner with another control, which is deemed a "buddy" control, the spinner must have the same parent as the associated control. This association has two optional aspects. The first aspect concerns size and placement; when you associate the spinner, by default, the system aligns it with the right side of the buddy control and makes it approximately the same height. Alternatively, you can specify that the system align it with the left side of the buddy, or you can choose not to align the spinner. You specify alignment with different values of the `style` property.

The second aspect concerns the value of the spinner. You can choose to have the associated control updated automatically when the `value` property of the spinner changes, and correspondingly, the value of the spinner changes if the user or your program changes the appropriate property of the buddy control. Depending on the class of the buddy, the property is one of `text`, `value`, or `caption`.

If you associate the spinner with an `Edit`, `RichEdit`, or `Combo` control, the value of the spinner is displayed in the `text` property of the buddy. If a user types numerals in the buddy control that fall within the range of the spinner, the value of the spinner changes.

If you associate the spinner with a `List` control, the `value` property of the list (which highlights that item in the list and makes it visible) corresponds to (but does not equal) the value of the spinner. If a user selects an item from the list, for example by clicking on it, the value of the spinner changes correspondingly. However, the default range of the spinner starts at zero, while the APL implementation of the list has the value in origin one, so the value of the spinner is always one less than the `value` property of the list.

If you associate the spinner with a `Label`, `Button`, `Check`, `Option`, or `Frame` control, the value of the spinner becomes the caption of the associated control. There is no method for a user to change the caption of any of these controls, so this association is one-way for the user. However, if you change the caption under program control, the value of the spinner is updated.

You can make the location association with other controls, for example, a `Trackbar` or a `Progress` control, but you must program any association in the `onChange` event handlers for the controls. If you assign an `onChange` event handler for a spinner that has an associated control with automatic update, the actions of the event handler occur in addition to the built-in association.

Some objects that have a caption property cannot be a buddy control. For example, a **Page** cannot be a buddy, because a **Spinner** cannot be a child of a **Selector**. You also cannot associate spinners with objects that do not appear on the screen, for example, a **Printer**.

In order to establish the association, you assign the name of the control you want to associate to the **buddy** property of the spinner. The associated control must already exist when you assign the **buddy** property.

If you take an action that causes the associated control to close and reopen, you should close and reopen the spinner, or reset the **buddy** property, or you may lose the association. If you move the buddy control, you may retain the value association, but the spinner does not automatically move in tandem with its buddy.

Regular Properties:

border, **buddy**, **children**, **class**, **color**, **data**, **def**, **deferexit**, **edge**, **enabled**, **events**, **extent**, **help**, **helpcontext**, **hwnd**, **imagescale**, **instance**, **keys**, **links**, **methods**, **mode**, **modified**, **modifystop**, **name**, **noredraw**, **opened**, **order**, **place**, **pointer**, **prompt**, **properties**, **range**, **scale**, **scrollaccel**, **scrollmargin**, **self**, **size**, **state**, **style**, **suppress**, **tabgroup**, **tabparent**, **tabstop**, **targetformats**, **theme**, **tooltip**, **tooltipenabled**, **tooltipstyle**, **tooltiptime**, **tooltipwidth**, **value**, **visible**, **where**, **wrap**, and $\Delta u d p$.

range

A two-element vector that specifies the lower and upper bounds for the spinner values. The default range is 0 100. The magnitude of the range cannot exceed 32767. However, the upper limit is 32767, and the lower limit is -32767.

If you set the two elements of the range to the same value, that effectively disables the spinner, by inhibiting the **onSpin** and **onChange** event handlers. If you set the first element of the range to be greater than the second element, the actions of the arrows are reversed; that is, clicking on the up arrow decreases the value and clicking on the down arrow increases it.

value

An integer scalar that returns the current value of the spinner. When a user clicks the spinner, it generates a **Spin** event. $\square w a r g$ for the **onSpin** event handler contains the increment and the old value; by default, $\square w r e s$ contains the increment. If you don't set $\square w r e s$ to zero, the **onChange** event handler follows immediately; $\square w a r g$ for this event contains the new value.

wrap

A flag that specifies whether the spinner wraps around when the user reaches the upper or lower limit. The default value of zero specifies no wrapping. You can set this property to 1 to allow wrapping.

buddy

A character vector that specifies the name of the control with which the spinner is associated. Selecting certain **style** values causes the spinner and its buddy to be connected either by location or value or both.

style

An integer vector that contains selected values or an integer scalar that is the sum of the selected values. The initial value for **style** is zero. The valid values for **style** are:

Values for the Spinner Class style Property

Value	Description
Primary Value: Choose only one value	
0	Vertical spinner
1	Horizontal spinner
Alignment Value: Choose only one value.	
0	Right alignment with buddy control

Value	Description
4	Left alignment with buddy control
8	No alignment
Add-On Values: Choose any of the values with the appropriate buddy class	
16	Update buddy control automatically
32	Use thousands separator (only with 16). (See note 1.)
64	Keyboard arrow keys do not change spinner. (See note 2.)

Note 1. In the US, the thousands separator is the comma. That setting is a system setting.

Note 2. By default the up and down arrows on the keyboard increment and decrement the spinner. You can limit the user interface to using the mouse to click on the spinner's arrows.

theme

The theme property of a Spinner class controls the visual styling theme used for drawing the control. The default value is an empty character vector (' '), which uses the new visual styling for the control. To suppress the new visual styling for the control, set thememode to "~".

Event-handler Properties:

onAction, onChange, onClose, onDelete, onDestroy, onDragEnter, onDragLeave, onDragover, onDrop, onExit, onExitError, onFocus, onHelp, onHide, onKeyDown, onKeyUp, onModified, onMouseDown, onMouseDrag, onMouseEnter, onMouseLeave, onMouseMove, onMouseUp, onMove, onOpen, onPaint, onReopen, onResize, onSend, onShow, onSpin, and onUnfocus.

onSpin

You can use the onSpin event handler to control the results of user actions; this event occurs before the value changes. `□warg` is a two-element integer vector: the first element is the increment of change; it can be positive or negative. When the user first clicks the spinner the magnitude is 1, but if the user holds the mouse down on the spinner the magnitude of the increment increases. The second element is the current value.

`□wres` contains the increment of change. This value is used as the argument to change the value, so you can change the increment in `□wres` to modify the value by whatever amount you choose; you can also set `□wres` to zero to leave the value unchanged, which also suppresses the subsequent onChange event handler.

onChange

The onChange event handler occurs after the value changes. `□warg` is an integer vector that contains the new value.

Methods:

Close, Create, Defer, Delete, Draw, Event, Exec, Focus, Help, Hide, Modify, New, Open, Paint, Popup, Ref, Resize, Send, Set, SetLinks, and Show.

Parentage:

Spinners are most often children of a Form, Frame, Page, or CommandBar. They may also be children of some control objects: Button, Check, Combo, Edit, Label, List, Option, Picture, Scroll, Spinner, Status, and Toolbox. The system allows most classes as children; it specifically prohibits Form, MDIForm, CommandBar, CommandButton, Page, and Printer.

Example:

```
'fmMain.lsMonths' □wi 'New' 'List' ('where' 0.8 4 1.5 4)
'fmMain.lsMonths' □wi 'list' ('Jan' 'Feb' 'Mar' 'Apr' 'May' 'Jun' 'Jul')
```

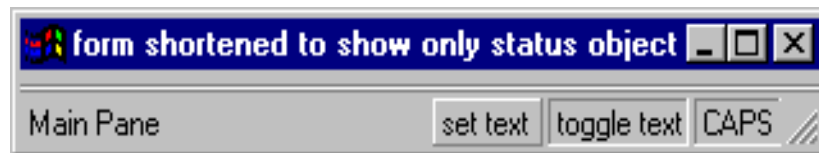
```

↳      'Aug' 'Sep' 'Oct' 'Nov' 'Dec')
'fmMain.spinMos' 0wi 'New' 'Spinner' ('buddy' 'f.ls') ('style' 20)
'fmMain.spinMos' 0wi 'range' 0 11

```

Note that you can still move through the list in the traditional manner, by pressing an arrow key or typing a letter. The value of the spinner changes as the value property of the list changes.

Status



Prefix: st

Summary:

A rectangular area at the bottom of a form in which you can display information in multiple subordinate rectangles called panes. The system automatically sizes the status bar and locates it along the bottom edge of its parent. The system adjusts the size and extent properties of the form or MDI form to exclude the area displaced by the status bar.

Uses:

There are two types of panes in a status bar: standard panes and overlay panes. You can display multiple standard panes of information of varying widths. You can control the width, border, justification, displayed image and text of each standard pane. You can vary the number of panes by removing the text of those you wish not to display; you can also leave a pane but remove the information by toggling text on and off or by establishing a minimum width and assigning an empty character vector as the text.

Overlay panes take up the entire bar except for the sizing grip at the right edge. (If the form does not have a sizable border, there is no sizing grip.) You can cause an overlay pane to display simply by assigning text to it. You can assign text to multiple overlay panes, but only one is visible at a time. When you remove the text from the visible pane, the next overlay pane appears on the form.

Typically, you use an overlay pane for transitory information and allow the standard panes to reappear when the reason for the overlay ceases.

Regular Properties:

border, children, class, color, data, def, deferexit, edge, enabled, events, extent, font, help, helpcontext, hwnd, imagelist, imagescale, instance, links, methods, mode, modified, modifystop, name, noredraw, opened, place, pointer, prompt, properties, scale, scrollaccel, scrollmargin, self, size, state, status, suppress, tabgroup, tabparent, tabstop, targetformats, theme, tooltip, tooltipenabled, tooltipstyle, tooltiptime, tooltipwidth, translate, visible, where, and $\Delta u d p$.

where, extent, and size (read-only)

Because the system automatically sizes the status bar, its where, extent and size properties are read-only.

status

The status property contains all the display information for the status bar. It is a seven-column nested matrix, with one row for each pane.

Important note: Additional columns may be added in a future release, so your application should be prepared to gracefully accept a result of more than seven columns if you reference this property.

When you assign the **status** property, you should allow for all the panes you will want to use. You can control the display of information by modifying the rows using the **SetStatus** method. If you want to add rows to the matrix, you must reassign the **status** property.

The default value for the **status** property is a two-row, seven-column matrix; it assumes the main standard pane and one overlay pane. The default value is shown below. Columns 2 and 7 contain empty character vectors.

```
1  ''  0  8  0  0  ''  
-1 ''  0  0  0  0  ''
```

Column 1: The first column is the non-zero integer number of the pane. Overlay panes have negative numbers; standard panes have positive numbers. If you use zero, the system assigns a unique positive number. You must have a pane numbered 1; this is the primary standard pane. It does not, however, have to appear first on the status bar. The panes appear in the order in which they exist in the matrix; if you have multiple overlay panes defined with text, the system displays the first one in the matrix.

Columns 2/3: The second and third columns determine the text and the width of the pane. You have two methods for using this pair of columns.

- **Flag method (Standard panes only):** This method requires that you place text in the third column. The second column is a flag. If you set the flag to 1, the text in column 3 is displayed. If you set the second column to zero, the text is removed, but the blank pane remains. If you want to remove the pane, you must remove the text from column 3. You can use this method most effectively if you want to toggle information on and off without changing text or pane attributes frequently.
- **Text method:** The second column contains the text you want to display. The third column determines a minimum width for the pane. If the text in column 2 is longer than the minimum, the system expands the pane to hold the text. You always use this method for overlay panes; since they take up the entire status bar, minimum width is irrelevant and the system ignores column 3. The third column determines the minimum width for the pane. If you specify a number, the system uses this as a minimum in current scaling units. If you specify a character vector, the system calculates the space it would require to hold that vector and treats it as the minimum value. You can use this method effectively if you want to change the text in standard panes frequently, and you want to avoid the flickering that would result from changing text lengths. Simply assign the longest message to column 3.

Column 4: The fourth column determines the border for each individual pane. (The border property determines the border for the status bar itself.)

Values for the Borders of Status Bar Panes (status Property Column 4)

Value	Description
Primary Value: Choose only one value for each pane	
0	No border.
1	Inset border.
2	Outset border
Add-on Values: Choose any one of the values for a given pane.	
8	Expand width of pane fully. (See note 1.)
16	Show Caps Lock state. (See note 2.)
32	Show Num Lock state. (See note 2.)
64	Show Scroll lock state. (See note 2.)

Note 1: For standard panes, if you add 8 to the border value, the system expands the width of the pane so that it takes all the available space on the status bar; that is, the space not actually required for other visible panes. If you specify this value for more than one pane, the excess space is divided evenly among these panes. This is the default value for the main pane only.

Note 2. You must use the `Flag` method for these panes. Define a standard pane; place a zero in column 2, your text, for example 'CAPS', in column 3; and the add-on value in column 4. When the user presses the Caps Lock key, it toggles the value in column 2 and the text in the pane on your status control. You can have only one of these in a pane. For example, to have the Nums Lock indicator: `'fmMain.stBar' □wi 'SetStatus' 1 0 'NUM' 33`

Columns 5/6: The fifth column is the origin 1 index to the image in the imagelist specified in the `imagelist` property that you want to display at the left of the pane when the pane is visible. The sixth column is the index into the `overlays` property of the same imagelist for an overlay image.

Column 7: The seventh column is initially an empty character vector; you can assign text for a tooltip that is specific to the pane. This pane-specific tooltip displays, whether or not the focus is on the form, when the user moves the cursor over the pane. If any visible pane has content in column 7, it disables the tooltip property for the control.

imagelist

The name of the `Imagelist` object that contains the images you want to display at the left of a pane. You use the origin one index to the images in this object to associate an image with a pane.

color

The first element of the `color` property determines the color of the background except for the grip at the right edge; the system ignores the foreground color designation. The system uses the color associated with 3D objects text in the Windows control panel for the text color.

theme

The `theme` property of a `Status` class controls the visual styling theme used for drawing the control. The default value is an empty character vector (' '), which uses the new visual styling for the control. To suppress the new visual styling for the control, set `thememode` to " ~ ".

Event-handler Properties:

`onAction`, `onClick`, `onClose`, `onDbtClick`, `onDelete`, `onDestroy`, `onDragEnter`, `onDragLeave`, `onDragover`, `onDrop`, `onExit`, `onExitError`, `onHelp`, `onHide`, `onModified`, `onMouseDouble`, `onMouseDown`, `onMouseDrag`, `onMouseEnter`, `onMouseLeave`, `onMouseMove`, `onMouseUp`, `onMove`, `onOpen`, `onPaint`, `onReopen`, `onResize`, `onSend`, and `onShow`.

Methods:

`Close`, `Create`, `Defer`, `Delete`, `Draw`, `Event`, `Exec`, `Help`, `Hide`, `HitTest`, `Modify`, `New`, `Open`, `Paint`, `PaneWhere`, `Popup`, `Ref`, `Resize`, `Send`, `Set`, `SetLinks`, `SetStatus`, and `Show`.

SetStatus

You use this method to change one row of the `status` property matrix. The full argument is a nested vector containing the pane number, text, width, border values, image indices, and pane-specific tooltip text. You can omit trailing elements of the argument. You can also provide a simple character vector; the system assumes this to be the (column 2) text for the main pane.

PaneWhere

This method returns a four-column matrix of values that are the equivalent of a `where` property for each pane in the `status` property (if you do not supply an argument) or each pane identified in the argument. The values are in current scaling units for the status bar; if a pane is not visible, all four elements are `-1`.

Parentage:

Status bars are most often children of a `Form` or an `MDIForm`. They may also be children of some control objects: `Button`, `Check`, `Combo`, `CommandBar`, `Edit`, `Frame`, `Label`, `List`, `Option`, `Page`, `Picture`, `Scroll`, `Spinner`, `Status`, and `Toolbox`. The system allows most classes as children; it specifically prohibits `Form`, `MDIForm`, `CommandBar`, `CommandButton`, `Page`, and `Printer`.

Example:

```
s ← 'fmMain.stStatusBar' □wi 'New' 'Status'
s □wi 'status' (⊃(1 'Greetings' 0 8) ((-1) ' ' 0 0)
↳ (2 ' ' 'longest message here' 1) (3 0 'CAPS' 17) )
s □wi 'SetStatus' 2 'Press Caps Lock'
s □wi 'SetStatus' -1 'NOW YOU KNOW'
```

Timer

Prefix: tm**Summary:**

A utility class that allows you to schedule regular callbacks to APL, which occur in the form of Timer events.

Uses:

You can create a timer object as a standalone control that you can open and close independently, or as a child of a form or another control. In the latter case, the Timer object sends Timer events to APL only when the parent object is open.

Regular Properties:

children, class, data, def, enabled, events, instance, interval, links, methods, modified, modifystop, name, opened, properties, self, state, suppress, and *△udp*.

interval

The frequency (in milliseconds) of the Timer events. The default is 1000 (1 second).

If you set the interval to zero, the Timer event occurs whenever the message queue is idle; that is, when there is no user input activity to any application and no other application is demanding CPU usage. When the interval is zero, your application performs background processing at the maximum speed that does not impact the responsiveness of the system to user input and does not keep other applications from processing efficiently in the foreground.

Event-handler Properties:

onAction, onClose, onDelete, onModified, onOpen, onReopen, onSend, and onTimer.

Methods:

Close, Create, Defer, Delete, Event, Exec, Modify, New, Open, Ref, Send, Set, and SetLinks.

Parentage:

Timers are top-level objects or children of any other object class. They do not contain other objects (or another timer).

Example:

This example creates an edit control that displays the current time to the nearest second. A timer control updates the display automatically.

```
▽ Clock
[1] 'fmMain.edClock' □wi 'text'
    (, 'G/99:99:99/' □fmt 100 □ts[4 5 6])
▽
'fmMain.edClock' □wi 'New' 'Edit' ('where' 0 0 1.5 8)
'fmMain.edClock.tm1' □wi 'New' 'Timer' ('interval' 1000)
'fmMain.edClock.tm1' □wi 'onTimer' 'Clock'
```


See the DEMOBALL workspace for another example of a timer.

Toolbox



Prefix: tb

Summary:

An array of icons or bitmaps in a toolbox format. These icons allow a user to click on a visible box to perform a common action quickly without accessing a menu or remembering a shortcut key combination.

Note that the Toolbox was an early Microsoft control. This description is included for completeness and compatibility. For new design, you can probably do anything with a floating command bar (or a form with border set to 512, the floating tool palette) more easily than with this class.

Uses:

When the user presses a mouse button over a toolbox cell, the toolbox captures it and triggers the `onMouse` event handler. The system reports subsequent mouse actions to that toolbox up to and including when the user releases the mouse button. When a Mouse event occurs for a toolbox, the `□warg` variable contains the row and column of the selected cell and a code that tells what kind of action occurred. This event handler allows you precise control of the consequences of user actions.

Generally, you want the affected toolbox icon or bitmap image to change when a Mouse event occurs so the user knows something has happened. Typically, three images are defined for each cell to represent three possible states. A 3-D image that appears "popped up" usually represents the unselected state. A 3-D image that appears "pushed down" usually represents the selected state. A 3-D image that appears "pushed down" more deeply than the selected state usually represents the transitional state, which occurs when the user presses a cell but does not release the mouse button.

Regular Properties:

`border`, `children`, `class`, `color`, `data`, `def`, `deferexit`, `edge`, `enabled`, `events`, `extent`, `font`, `help`, `helpcontext`, `hwnd`, `imagescale`, `instance`, `keys`, `links`, `list`, `methods`, `mode`, `modified`, `modifystop`, `name`, `noredraw`, `opened`, `order`, `place`, `pointer`, `prompt`, `properties`, `scale`, `scrollaccel`, `scrollmargin`, `self`, `size`, `state`, `style`, `suppress`, `tabgroup`, `tabparent`, `tabstop`, `targetformats`, `theme`, `tooltip`, `tooltipenabled`, `tooltipstyle`, `tooltiptime`, `tooltipwidth`, `value`, `visible`, `where`, and `Δudp`.

list

A nested matrix, or a nested vector that the system treats as a one-row matrix, that specifies the names of the bitmaps or icons you want in the toolbox. Each item of this nested matrix specifies the bitmaps or icons that occupy one cell of the toolbox. If you specify one name for each cell position, this icon or bitmap will always be the one displayed.

If you want to be able to toggle among several bitmaps or icons displayed in a cell, you can specify a nested vector of names. For example, when the user presses the mouse button over a cell, you may want your application to display an icon that appears to be pushed in, and an icon that appears to be highlighted or selected when the user releases the mouse button.

The icons and bitmaps in a toolbox can exist as bound resources referred to by the `Resource DLL=` entry in your `.INI` file, or as unbound resources in `ICO` or `BMP` files in a resource directory associated with your application. The `RESOURCES=` configuration parameter in your application's `.INI` file controls where the system searches for the icons.

style

An integer scalar or vector that determines whether the system interprets the names you specify for the list property as icons (`.ICO`) or bitmaps (`.BMP`) and also controls the spacing between images on the toolbar. The table below shows the possible values; use either of the primary values plus any of the add-on values.

Values for the Toolbox Class style Property

Value	Description
Primary Value: Choose only one value	
0	Items in list property refer to icons
1	Items in list property refer to bitmaps
Add-on Values: Choose any of the values	
0	No overlap or spacing between icons or bitmaps
16	One pixel spacing between icons or bitmaps
32	One pixel overlap of icons or bitmaps

theme

The theme property of a Toolbox class controls the visual styling theme used for drawing the control. The default value is an empty character vector (`' '`), which uses the new visual styling for the control. To suppress the new visual styling for the control, set `thememode` to `"~"`.

value

A matrix that contains one number for each cell in the Toolbox. The contents of this matrix determines which bitmap or icon the system displays for each cell. By default, each element of `value` is 0, which causes the system to display the first icon or bitmap for each cell. When the `value` property for that cell changes to 1, the system displays the second icon in that cell. When the `value` property for that cell changes to 2, the system displays the third icon in that cell. You can have the `onMouse` event handler set the `value` property when the user presses the mouse over a cell to display the various icons you assign.

Event-handler Properties:

`onAction`, `onClose`, `onDelete`, `onDestroy`, `onDragEnter`, `onDragLeave`, `onDragover`, `onDrop`, `onExit`, `onExitError`, `onHelp`, `onHide`, `onModified`, `onMouse`, `onMouseDouble`, `onMouseDown`, `onMouseDrag`, `onMouseEnter`, `onMouseLeave`, `onMouseMove`, `onMouseUp`, `onMove`, `onOpen`, `onPaint`, `onReopen`, `onResize`, `onSend`, and `onShow`.

onMouse

The `Mouse` event occurs whenever the user presses a mouse button over a toolbox cell. When the user presses a mouse button over a toolbox cell, the toolbox captures it. The system reports all subsequent mouse actions to that toolbox. Moving the mouse over other parts of the application interface does not affect other control objects until the user releases the mouse button. When a `Mouse` event occurs for a toolbox, `EventArgs` contains a three-element vector: *row column change*

The *row* and *column* parameters identify which cell of the toolbox is selected (the upper-left cell is row 1, column 1). The *change* parameter identifies which mouse action occurred and can have the following values:

- **Press:** The user presses the left mouse button down while the cursor is over a toolbox cell. Once a cell has been pressed, and until the mouse button is released, all subsequent Mouse events for a specific toolbox refer to this original cell. Releasing the mouse button can result in a Release, Click, or Double-click change code.
- **Drag-off:** The user drags the cursor off the cell originally selected, without releasing the mouse button, after a Press value occurs.
- **Drag-on:** The user drags the cursor back over the cell originally selected, without releasing the mouse button, after a Drag-off value occurs.
- **Release:** The user releases the mouse button when the cursor is not over the cell originally selected, after a Drag-off value occurs.
- **Click:** The user releases the mouse button over the cell originally selected after a Press value occurs.
- **Double-click:** The user double-clicks the mouse button over a toolbox cell. This event is always preceded by a Click on the same cell.

You can use the `Wttoolbox` function in the `WINDOWS` workspace to simplify processing of Mouse events and to manage the toolbox.

Methods:

Close, Create, Defer, Delete, Draw, Event, Exec, Help, Hide, Modify, New, Open, Paint, Popup, Ref, Resize, Send, Set, SetLinks, and Show.

Parentage:

Toolboxes are most often children of a Form, MDIForm, Frame, Page, or CommandBar. They may also be children of some control objects: Button, Check, Combo, Edit, Label, List, Option, Picture, Scroll, Spinner, Status, and Toolbox. The system allows most classes as children; it specifically prohibits Form, MDIForm, CommandBar, CommandButton, Page, and Printer.

Example:

The following example creates a Toolbox with two rows containing three icons each.

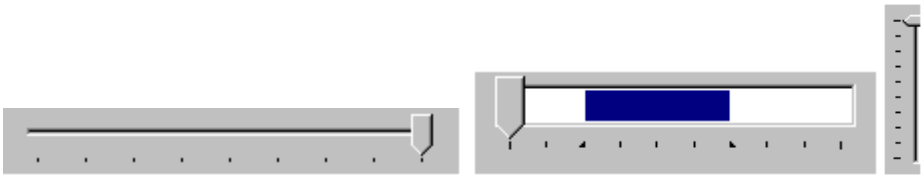
```
icons ← (2 3p'buttonup.ico' 'checkup.ico' 'comboup.ico'
↳ 'editup.ico' 'framedn.ico' 'hscrolup.ico')
'fmTools.tb1' □wi 'New' 'Toolbox' ('list' icons)
```

The following example creates a Toolbox that has three cells, each of which contains three icons.

```
icons ← ('buttonup.ico' 'buttondn.ico' 'buttonsel.ico')
↳ ('checkup.ico' 'checkdn.ico' 'checksel.ico')
↳ ('editup.ico' 'editdn.ico' 'editssel.ico')
'fmTools.tb1' □wi 'list' icons
```

Trackbar

Prefix: trk



Summary:

A graphical device that allows the user to select a discrete value and the programmer to display a current selection or range.

Uses:

A trackbar is similar to a scroll bar. It is useful any time you want the user to select a value from a small range like selecting a volume setting.

Regular Properties:

border, channelwhere, children, class, color, data, def, deferexit, edge, enabled, events, extent, font, help, helpcontext, hwnd, increment, imagescale, instance, keys, links, methods, mode, modified, modifystop, name, noredraw, opened, order, place, pointer, prompt, properties, range, scale, scrollaccel, scrollmargin, selection, self, size, sliderlen, sliderwhere, state, style, suppress, tabgroup, tabparent, tabstop, targetformats, theme, tickinterval, tickpos, ticks, tooltip, tooltipenabled, tooltipstyle, tooltiptime, tooltipwidth, translate, value, visible, where, and *Δudp*.

range

A scalar or zero-, one- or two-element integer-valued numeric vector that specifies the minimum and maximum values for the trackbar. If you specify a two-element vector, the elements are the minimum and maximum values. If you specify a negative scalar or one-element vector, the system treats it as the minimum and assumes a zero for the maximum. If you specify a positive scalar or one-element vector, the system treats it as the maximum and assumes zero for the minimum. If you specify an empty vector, the system restores the default values of 0 100.

value

A scalar that shows the current position of the slider. The initial value is zero. If you set value, the system moves the slider to the corresponding position on the trackbar.

increment

A two-element vector that specifies the page increment and line increment. The page increment is the number of values the slider moves when a user presses the PageDown or PageUp key or when the user clicks on the trackbar channel. The line increment is the number of values the slider moves when a user presses an arrow key. If you specify -1 for the page increment, the system sets it automatically; this is the default value. The default value for line increment is 1.

style

An integer vector that contains selected values or an integer scalar that is the sum of the selected values. The initial value of style for any newly created control object is 0. The table below shows the possible values; use one of the primary values and one of the tick placement values. You can also choose any of the add-on values.

Values for the Trackbar Class style Property

Value	Description
Primary Value:	Choose only one value

Value	Description
0	Vertical.
1	Horizontal.
Tick Placement Value: Choose only one value	
0	Ticks on bottom or right side.
2	Ticks on top or left side.
4	Ticks on both sides.
Add-on Values: Choose none, any, or all of the values	
8	Hide tick marks but display slider.
16	Track slider movement. (See note.)
32	Enable selection range.

Note: By default, one `Change` event occurs when the user releases the mouse button to stop scrolling. If you add in 16, a user who drags the slider generates a string of `Change` events.

selection

A two-element integer-valued numeric vector that specifies the beginning and end points of a portion of the channel that you want to appear in the Windows highlight (selected items) color. The default value is an empty vector; you must specify 32 in the `style` property to enable this property. If you set this style, Windows redraws the trackbar with a wider channel to display the selection range. The values you specify must be within the values of the `range` property.

tickinterval

By default, the system places tick marks at the minimum and maximum unless you specify 8 in the `style` property to hide all ticks. This property is a scalar that specifies the spacing of the tick marks along the range between the minimum and maximum values. The default value is 0; which means only the tick marks at the minimum and maximum display. You can specify any positive value up to the length defined by the `range` property. If you specify 1, the system sets a tick for each integer value. If you specify 2, the system places a tick at every second value, and so on.

If you want to assign the tick mark positions yourself, set this value to zero. The default ticks at the minimum and maximum appear. You can assign the tick positions you want the trackbar to display between the minimum and maximum in the `ticks` property. Whenever you set the `tickinterval` property, the system resets the `ticks` property to an empty vector.

ticks

An integer vector that contains the tickmark list for a trackbar if you set the ticks explicitly; otherwise, it is empty. If you set the `ticks` property, the system resets `tickinterval` to zero. If you specify ticks outside the range, they do not display.

tickpos (read-only)

A two-column matrix that contains the list of the currently displayed tickmarks for a trackbar. The first column contains the integer values corresponding to each tickmark. The second column contains the physical position in the currently selected scaling units. This matrix includes a row each for the ticks that correspond to the minimum and maximum values of the range. If you specify the ticks manually, this matrix is in the order you specify, not necessarily in ascending order.

channelwhere (read-only)

A four-element vector that specifies the position and size of the trackbar's channel. The elements, in order, are row, column, height, and width (similar to the `where` property).

sliderwhere (read-only)

A four-element vector that specifies the position and size of the slider. The elements, in order, are row, column, height, and width (similar to the `where` property).

sliderlen

A scalar that specifies the width of the slider in a vertical trackbar or the height of the slider in a horizontal trackbar. The default value is `-1`, which means the system sets the size automatically. You can determine the actual size by referencing the `sliderwhere` property. If you specify a positive value, you define the length of the slider in current scaling units. If you set this property to zero, the slider is hidden. The user cannot drag a hidden slider, and there is no visual indication of the current value; however, the user can change the value from the keyboard with the arrow keys and the `PageUp` and `PageDown` keys.

color

The first element of the `color` property determines the color of the background outside the channel area; the system ignores the foreground color designation. The system uses the color associated with 3D objects in the Windows control panel for the color of the slider; it uses the color associated with 3D objects text in the Windows control panel for the color of the ticks.

Note: You may have to close and reopen the trackbar to see the effects of a color change.

theme

The `theme` property of a `Trackbar` class controls the visual styling theme used for drawing the control. The default value is an empty character vector (`' '`), which uses the new visual styling for the control. To suppress the new visual styling for the control, set `thememode` to `"~"`.

Event-handler Properties:

`onAction`, `onChange`, `onClose`, `onDelete`, `onDestroy`, `onDragEnter`, `onDragLeave`, `onDragover`, `onDrop`, `onExit`, `onExitError`, `onFocus`, `onHelp`, `onHide`, `onKeyDown`, `onKeyPress`, `onKeyUp`, `onModified`, `onMouseDown`, `onMouseDrag`, `onMouseEnter`, `onMouseLeave`, `onMouseMove`, `onMouseUp`, `onMove`, `onOpen`, `onPaint`, `onReopen`, `onResize`, `onSend`, `onShow` and `onUnfocus`.

onChange

The `onChange` event handler occurs after the value changes. `newValue` is an integer vector that contains the new value. A change in the trackbar value does not trigger the `onModified` event handler or set the `modified` property.

Methods:

`Close`, `Create`, `Defer`, `Delete`, `Draw`, `Event`, `Exec`, `Focus`, `Help`, `Hide`, `Modify`, `New`, `Open`, `Paint`, `Popup`, `Ref`, `Resize`, `Send`, `Set`, `SetLinks`, and `Show`.

Parentage:

Trackbars are most often children of a `Form`, `Frame`, `Page`, or `CommandBar`. They may also be children of some control objects: `Button`, `Check`, `Combo`, `Edit`, `Label`, `List`, `Option`, `Picture`, `Scroll`, `Spinner`, `Status`, and `Toolbox`. `Trackbar` objects can contain `Timers`, `Imagelists`, or hidden `Menus`.

Example:

```
'fmForm.trkBar' []wi 'New' 'Trackbar'('style' 67) ('where' 2 5 3 26)
'fmForm.trkBar' []wi 'border' 1
'fmForm.trkBar' []wi 'range' 0 100
'fmForm.trkBar' []wi 'selection' 35 65
'fmForm.trkBar' []wi 'tickinterval' 5
'fmForm.trkBar' []wi 'increment' 4 2
```

Tree



Prefix: tr

Summary:

A Tree is a graphical representation of a hierarchical list.

Uses:

A Tree generally allows the user to navigate through a list formatted like an outline. A node of the tree that has children can either be expanded (the children are displayed) or collapsed (the children are not displayed). The user can expand or collapse a node by double-clicking on its label or, if the button style is selected, by clicking on the button to the left of the node label. You can also expand or collapse a node under program control by changing the state of the node. You control the appearance of the node by specifying or negating its state values, which are in the fifth column of the list property.

Each node of a tree can have one or more associated images that display to the left of the label. Images are contained in an Imagelist object. In lieu of a second image, you can have check boxes to the left of the label. You can also use a Tree object as either the source or the target of drag-and-drop operations. See the description of Drag-and-Drop in the Programming the Windows Interface: Intermediate chapter in this manual.

Regular Properties:

border, children, class, color, count, data, def, deferexit, dragimage, edge, enabled, events, extent, font, help, helpcontext, highlightbold, highlightcut, highlightdrop, highlightselect, hwnd, imagelist, imagescale, imagelistuser, indent, instance, keys, labeledithwnd, links, list, methods, mode, modified, modifystop, name, noredraw, opened, order, place, pointer, prompt, properties, scale, scrollaccel, scrollmargin, searchstring, self, size, sourceformats, state, style, suppress, tabgroup, tabparent, tabstop, targetformats, theme, tooltip, tooltipenabled, tooltipstyle, tooltipwidth, translate, value, visible, where, and *undp*.

list

An array describing the nodes in the tree. The array has one row with five columns for each node. When inserting new nodes by setting the list property or by using the `InsertNodes` method, you must locate a node after its parent and before its parent's next sibling. The columns contain information about the node and how it is displayed.

Node ID

The first column is the node identifier. The node identifier is a positive integer less than 2^{31} . When specifying the list property, you may specify the identifier as 0 and have APL64 assign identifiers. If you assign identifiers, they must be unique.

Depth

The second column is the relative depth of the node. These numbers must be a set of integers where each node has a depth that is one greater than its parent's depth. By default, the first level nodes are depth 1, and their immediate children are depth 2.

You can assign an arbitrary depth to the first node you create, and use the next higher integer for its children. You can have any number of levels, but you cannot skip levels.

Label

The third column is the text label. The text label is a character vector to be displayed for the node or a negative numeric identifier. Although you can specify longer labels, the control uses only 259 characters. You can specify -1 as the text label; the `Tree` control requests the label via the `GetVirtualData` event when it is needed to display or sort the node. You must program your `onGetVirtualData` event handler to provide the information. If you specify -2 for the label, the system inserts the default value when you are adding or inserting a row, and does not change the value if you are changing other values in the row.

Image Indices

The fourth column holds the image indices. This is a four-element numeric vector where the first element is the index to the image you want to display to the left of the text label when the node is not selected. The second element is the index to the image you want to display to the left of the text label when the user selects the node. The third element is the index to an overlay image, and the fourth element is the index to a user-state image. The first and second indices may not be zero.

The value for a selected or nonselected image is the origin one index to the image in the `Imagelist` object whose name you designate in the `imagelist` property of the `Tree`. The overlay value is the index of an overlay image from the same `Imagelist`; however, you must specifically designate an overlay image as such in the `overlays` property of the `Imagelist` object; in the `list` property of the tree, you use the index value from 1 to 15 of the `overlays` property of the `imagelist`, rather than the direct index into the `imagelist`. See the description of the `Imagelist` class for a description of overlay images.

The user-state image index identifies an image from the `Imagelist` object whose name you designate in the `imagelistuser` property of the `Tree`. A user-state image is displayed to the left of the nonselected or selected image.

Warning: Due to a quirk in the Windows system, you cannot access the first image in a user-state `imagelist`. Until this problem is fixed, you need to enter a dummy image in the first position and start your indices with 2. Also, due to a Windows limitation, you cannot index more than 15 images in an `imagelist` being used for user state images. In APL, using an index greater than 15 results in a `DOMAIN ERROR`.

The default values for the image indices are 1 1 0 0. You can specify -1 for a selected or nonselected image; the `Tree` control requests the image via the `GetVirtualData` event when it is needed to display the node. You can also specify -2 to retain the existing image index when the item already exists. You cannot specify zero for either the first or second index.

States

The fifth column contains a nested vector of character vectors describing the states of the node. The valid states are: 'highlightbold', 'highlightcut', 'highlightdrop', 'highlightselect', 'expanded', 'expandedonce', 'haschildren', and 'forceparent'; you can precede each word by a minus sign ('-') to turn the state off; the special value '?forceparent' means request the `forceparent` state via the `GetVirtualData` event when needed.

The four `highlight` states are explained below under the properties of the same names. The `expanded` state means the immediate children of the node are available for display. If the node is collapsed, the children are not visible even when the node is displayed. If a node is a parent, it returns the `haschildren` state; you cannot set this state directly. If you attempt to set this state, the setting is ignored, but the system does not signal an error.

If you set the `forceparent` state, the system treats the node as if it has children; for example, the node has a button if you set the `style` property for buttons. Clicking on such a node also triggers the `onExpanding` event handler. You can use this state to postpone inserting children until they are actually needed for

display. When you specify `?forceparent`, the system triggers the `onGetVirtualData` event handler when the node is about to be displayed. The system uses the result of `onGetVirtualData` to determine if there are children.

If the `expandedonce` state is set, the system does not fire `onExpanding` and `onExpanded` event handlers for the node. The system sets the `expandedonce` state each time the node is expanded or collapsed; if you want the event handlers to fire, you must specify `-expandedonce` to turn the state off every time the `onExpanded` handler is triggered. The example node list below describes the earlier sample tree picture.
`CommentHeader 7 ('fmExample.trSapling' 0wi 'list')`

AID	Depth	Label	Image	States
A			Indices	
1	1	English	1 1 0 0	highlightbold highlightselect
2	1	Greek	1 1 0 0	highlightbold expanded expandedonce haschildren
3	2	Alpha	1 1 0 0	expanded expandedonce haschildren
8	3	Little A	1 1 0 0	
9	3	Big A	1 1 0 0	
5	2	Beta	1 1 0 0	
12	2	Omega	1 1 0 0	
6	1	Latin	1 1 0 0	highlightbold

Note: You can assign the `list` property directly, or you can let the system build this list by using the appropriate methods to define your nodes.

count (read-only)

A two-element integer vector giving the total number of nodes in the `Tree` and the number of nodes that will fit fully visible in the client area of the `Tree`. (For some heights, you may think you can see one node more than this property reports.)

indent

A numeric scalar giving the distance a child is indented from its parent, in the current scaling units for the object, or an empty vector if the `Tree` is closed and the `indent` property has not been explicitly specified. Setting this to an empty vector restores the default indent.

style

An integer vector that contains selected values or an integer scalar that is the sum of the selected values. The table below shows the possible values. The default style has a button to the left of any node that is both a child and a parent indicating whether it is collapsed or expanded and has lines connecting children.

Values for the Tree Class style Property

Value	Description
0	Lines connecting children; buttons on parents below the top level (default value).
1	Does not have buttons.
2	Does not have lines connecting children
4	Has buttons for top-level parents (if 1 is not included) and has lines connecting children to top-level nodes (if 2 is not included).
8	Always highlight selected node, even when focus is lost.
16	Enable editing of labels.
32	Enable automatic check boxes as user state images. (See <code>imagelistuser</code> , below.)
64	Suppress info tips.
128	Suppress unfolding of label tips.

searchstring (read-only)

A character vector containing the current incremental search string. A `Tree` uses the incremental search

string to select a node based on characters typed by the user. This is an empty vector if no incremental search string is in progress or if the `Tree` is closed.

theme

The `theme` property of a `Tree` class controls the visual styling theme used for drawing the control. The default value is an empty character vector (`' '`), which uses the new visual styling for the control. To suppress the new visual styling for the control, set `thememode` to `"~"`.

value

The numeric identifier of the currently selected node; zero if no node is selected. Note that you can query and set this property using the `FindNode` and `ShowNode` methods with the `'select'` argument, as well as querying and setting it directly.

highlightbold

Integer vector of identifiers of nodes displayed with a bold font.

highlightcut

Integer vector of identifiers of nodes displayed as if cut.

highlightdrop

Integer vector of identifiers of nodes displayed as drop targets. Note that you can display one or more nodes as drop targets, but there is only one actual drop target. You can determine the drop target with the `FindNode` method; you set the drop target with the `ShowNode` method. The actual drop target is independent of the display state.

highlightselect

Integer vector of identifiers of nodes displayed as if selected. Note that you can display one or more nodes as selected, but only one node is actually selected. That node is identified by the `value` property, or you can query it using the `FindNode` method. You can select a node by setting the `value` property or using the `ShowNode` method; the user can also click on a node to select it.

imagelist

The name of the `Imagelist` object that contains the images you want to display to the left of the label. You use the origin one index to the images in this object to connect an image with a node. You can specify this name as a fully-qualified name or as a name beginning with a period, which is relative to the `Tree` object.

imagelistuser

The name of the `Imagelist` object that contains the user-state images you want to display to the left of the image from the `imagelist` property.

The `Tree` control also allows the display of check boxes in the place of user state images by setting a `style` value. If you do not have an `imagelist` defined in the `imagelistuser` property, and you do not have index values set for user state images, when you set `style` to include 32, check boxes appear for each node of the tree. Each check box appears to the left of the image, if you have (non-selected) images defined, or to the left of the label if you are not using images; initially the box is unchecked. When the user clicks on a box, it toggles the box between the unchecked and checked states. When the user clicks the check box, it does NOT select the node of the tree.

You can tell when a user clicks a check box; the second element of `EventArgs` in the `onClick` event handler contains the string `'imageuser'`. You can tell the state of a check box by using the `GetInfo` method with the first argument `'checkstate'`. The method returns 2 if a box is unchecked or 3 if it is checked. These values are the user state image indices in the `list` property of the `Tree`, and they correspond to the positions

of the unchecked and checked images in the implicit imagelist used by the control. (Remember that Microsoft designed the **Tree** so you cannot access the first image of a user state imagelist.)

You can change the state of a check box by using the **SetInfo** method with the same first argument. You can assign a zero if you want to remove the check box entirely from a node. These values (zero, 2 or 3) appear as the fourth elements of the fourth column of the list property. You can also query and set these values by referencing or resetting the image values for user state image.

If you have set non-zero values for the user state images, the check boxes do not automatically appear when you set **style** to include 32. Only those nodes that have 2 or 3 set will display check boxes. However, if the user clicks in the area where the user state image would be, the check box will appear for that node. Whether it appears in a checked or unchecked state depends on the previous value assigned to the user state image. Once it is clicked, the value becomes 2 or 3.

If you assign an existing imagelist to the **imagelistuser** property when you have 32 in **style**, the imagelist you assign overrides the check boxes. However, if the user clicks on the user state image, the system increases the image index by one, without selecting the node. The user state image that displays for a given node will cycle through the images available in the imagelist you are using.

This interaction between the behavior for user state images and the style for check boxes allows you to have a multi-state user image that is under the control of the user. You can query the state of the image as described above. The **GetInfo** method returns the user state image index currently assigned for that node. Because the Microsoft design for the **Tree** control does not access the first image of an imagelist assigned to user state images, the system cycles from the last image to zero to the second image.

WARNING: It is prudent to use an imagelist with 15 or fewer images when employing this feature.

color

The first element of the **color** property determines the color of the background outside the label text area; the system ignores the foreground color designation. The system uses the Window Text and Windows colors for the text foreground and background.

labeledithwnd

The window handle of the system-generated edit control that allows label editing. You can reference this property to subclass the window for low-level programming.

sourceformats

One or more character vectors that specify the format for dragging items from this control. The valid values are 'Text' and 'TreeItem'. See the section on Drag-and-Drop operations in the Programming the Windows Interface: Intermediate chapter in this manual.

targetformats

One or more character vectors that specify the format for dragging items to this control. The valid values are 'FileName', 'FileDrop', 'Text', 'ListviewItem', and 'TreeItem'.

infotipmask

The **infotipmask** property controls which places in the **Tree** "info tip" are triggered. This can be one or more of the following values: 2 = on icon; 4 = on label; 8 = on indent; 16 = on button; 32 = to right of label; and 64 = on state icon. Use ~1 to select all possible places.

Event-handler Properties:

onAction, **onBeginLabelEdit**, **onChange**, **onClick**, **onClose**, **onDblClick**, **onDelete**, **onDestroy**, **onDragContinue**, **onDragEnd**, **onDragEnter**, **onDragFeedback**, **onDragLeave**, **onDragover**, **onDragStart**, **onDrop**, **onEndLabelEdit**, **onExit**, **onExitError**, **onExpanded**, **onExpanding**, **onFocus**, **onGetInfoTip**,

onGetVirtualData, onHelp, onHide, onKeyDown, onKeyPress, onKeyUp, onModified, onMouseDouble, onMouseDown, onMouseDrag, onMouseEnter, onMouseLeave, onMouseMove, onMouseUp, onMove, onOpen, onPaint, onReopen, onResize, onSend, onShow, onSort, onUnfocus, and onValidate.

onClick and onDbClick

When the user triggers either the onClick or onDbClick event handler, `□warg` contains a six-element vector containing the identifier of the node that was clicked, a nested character vector that shows what part of the node was clicked, the shift key state, the row and column of the mouse in current scaling units, and whether the left or right mouse button was pressed.

The location word in the second element, provided by the Windows `TreeView_HitTest` call, can be one of the following: `image`, `button`, `label`, `imageuser`, `indent`, `noderight`, `above`, `below`, `toleft`, `toright`, `aboveleft`, `aboveright`, `belowleft`, `belowright`, `nowhere`, or an empty vector.

The shift key state is the sum of 1 for the Shift key, 2 for the Ctrl key, and 4 for the Alt key. The mouse button indicator shows 1 for the left button, 2 for the right.

onChange

The onChange event handler fires whenever the selected node changes; onChange occurs after onClick but before onDbClick. `□warg` is a two-element vector that contains the node identifiers of the newly selected node and the previously selected node.

onExpanding and onExpanded

The onExpanding and onExpanded event handlers fire when a node is expanding (has expanded) or is collapsing (has collapsed) and the `expandedonce` state is not set. `□warg` contains two elements: the first element is the node identifier; the second element is 1 if the node is expanding (has expanded) or zero if it is collapsing (has collapsed).

You can use the onExpanding event handler to perform actions before the node is expanded (or collapsed), for example to add children, or you can prevent the node from expanding by setting `□wres←0`. You can use the onExpanded event handler to perform actions after the node is expanded (or collapsed). This is the appropriate place to reset the `expandedonce` state.

onGetVirtualData

When the onGetVirtualData event handler is triggered, `□warg` contains two elements, the node identifier and an enclosed list of one or more of the following character vectors: `'children'`, `'image'`, `'imageselect'`, and `'label'`.

The vectors in the second element of `□warg` identify the information the system needs for the identified node. You must assign an array to `□wres` with one element for each name in the enclosed list in the second element of `□warg`. The values to assign are, respectively, the number of children, the index to a nonselected image, the index to a selected image, and the label. When the event handler returns, if `□wres` contains an improper value or no value for the expected element, the system ignores that element and does not generate an error.

The following example shows a typical event handler for onGetVirtualData. The `:else` clause in lines 11 and 12 allow for future extensions to virtual data handling without breaking this code.

```
▽ GetVirtualData;type
[1] :for type :in (1+□io) > □warg
[2]   :select type
[3]   :case 'children'
[4]     □wres ← □wres, . . . A Number of children
[5]   :case 'image'
[6]     □wres ← □wres, . . . A Index for normal image
[7]   :case 'imageselect'
```

```

[8]      □wres ← □wres, . . . A Index for selected image
[9]      :case 'label'
[10]     □wres ← □wres, . . . A Enclosed label
[11]     :else
[12]     □wres ← □wres, 0      A Safe programming practice
[13]     :endselect
[14] :endfor

```

▽

onValidate

The `onValidate` event handler fires just before the selected node changes. The first element of `□warg` contains a numeric code that identifies what action is causing the change. If the user changes the selection, the value is positive: 2 for a mouse click, or 1 for a keystroke. You can cancel the node change in the event handler with setting `□wres` to 0.

If the system does not recognize the cause, the value is zero. If the application causes the change, the value is negative: -1 if you invoke the `Validate` method, or -2 if you set the `value` property. You can set the `value` property directly or you can set it indirectly by using the `ShowNode` method with the `select` operation.

In all cases except using the `Validate` method, `□warg` is a three element vector; the last two elements are the numeric node identifiers of the previously selected and newly selected nodes. If you use the `Validate` method and supply an argument, `□warg` is a two-element vector with the argument being the second element. If you do not supply an argument, `□warg` is a scalar.

onSort

You can trigger the `onSort` event handler by using the `SortChildren` method with the proper argument. You can use this combination when you want to reorder a node's children other than alphabetically. When the handler is invoked, `□warg` contains the identifiers of two nodes to be sorted. The handler must return a negative value if the first identifier should precede the second or a positive value if the first value should follow the second. You can have `□wres` return zero if the two nodes have equivalent order. This event handler is not effective if the tree is closed.

onGetInfoTip

When the event handler is triggered, `□warg` is a two-element nested vector. The first element is the `NodeID` for which information is needed. The second element is a scalar defining the information needed; it can be one of the following values: 2 = on icon; 4 = on label; 8 = on indent; 16 = on button; 32 = to right of label; and 64 = on state icon. **Note:** This event is triggered for nodes unless style 64 is selected.

The value of `□wres` on return from the handler determines the text to be displayed in the tooltip window. You can use `□wres` to display a custom value tip or suppress the value tip by assigning it to ' '.

Methods:

`BeginLabelEdit`, `CancelLabelEdit`, `Close`, `Create`, `Defer`, `Delete`, `DeleteNodes`, `Draw`, `EnsureVisible`, `Event`, `Exec`, `Expand`, `FindNode`, `Focus`, `GetInfo`, `Help`, `Hide`, `HitTest`, `InsertNodes`, `Modify`, `New`, `Open`, `Paint`, `Popup`, `Ref`, `Resize`, `Send`, `Set`, `SetInfo`, `SetLinks`, `Show`, `ShowNode`, `SortChildren` and `Validate`.

InsertNodes

You use the `InsertNodes` method to add nodes to the tree; this method requires three arguments. To insert a single node:

- The first argument is the numeric identifier of the parent node. If the node you want to insert is a top-level node, use the character string 'root' as the first argument.
- The second argument is the numeric identifier of the sibling node (node on the same level) that physically precedes the new node in the tree. You can also substitute the characters strings 'first' or 'last' to place the new node at the beginning or end, respectively, of its group of siblings. You can also use the character string 'sort' as the second node. This causes the system to place the new node

in alphabetical order, assuming the previously inserted nodes are already in alphabetical order. If the previous nodes are not alphabetized, the position of the new node is undefined. **Note:** If you are using virtual data for labels, you should not use this method when the tree is closed.

- The third argument is a nested vector containing the structure for one row of the list property. You can use zero for the node identifier and allow the system to assign a unique value; you can omit the last two columns if you want to assign only the depth and the label. You can insert multiple nodes in one statement by making the third argument a nested matrix with one row for each new node and one column for each element of the structure for the list property. You can use zero for the node identifier and allow the system to assign unique node identifiers. You can omit the last two columns if you wish to define only the depths and labels.

Note: If you use a matrix to insert multiple nodes and you specify 'first', 'last', or 'sort' as the second argument, the placement argument applies only to nodes at the same depth as the node in the first row of the matrix. The system inserts children of the highest-level new nodes in the order in which they appear in the matrix.

DeleteNodes

You can delete one or more nodes by providing the node identifiers as the argument to this method. Deleting a node deletes its children.

FindNode

You can identify a node that has certain characteristics using this method. The first argument is one of the character vectors shown in the table below. The second argument for those that require it is the node identifier. This method returns a node identifier or zero if none qualify.

Operations for FindNode Method

Operation	Node Identifier Returned
'root'	First top-level node (no second argument)
'parent'	Parent node
'child'	First child node
'next'	Next sibling node
'prev'	Previous sibling node
'children'	All children of the node (returns a numeric vector)
'firstvisible'	First visible node (no second argument)
'nextvisible'	Next visible node
'prevvisible'	Previous visible node
'drop'	Drag and Drop target (no second argument)
'select'	Currently selected node (no second argument)

ShowNode

You can make a node the currently selected node or the current drag and drop target using this method; you can also position a node at the top of the visible portion of the tree. The first argument is one of the character vectors: 'select', 'drop', or 'firstvisible'. The second argument is the node identifier if you want to specify a node; you can also specify zero or the character vector 'none' to remove the attribute from the currently selected node or the current drag and drop target. You cannot use this method if the tree is closed.

This method returns 1 if successful; zero otherwise. This method can be unsuccessful because the onValidate event handler set `mwres` to zero.

Expand

You can use the Expand method to control the display of a node and its children. The possible first

arguments, and their effects, are shown in the table below; the second argument is the numeric node identifier. This method returns a 1 if successful; zero if unsuccessful, for example if you attempt to expand a node with no children or one that is already expanded.

Operations for Expand Method

Operation	Result
'expand'	Opens the children for display.
'collapse'	Hides the children so they do not or will not display.
'toggle'	Switches between expanded and collapsed.
'expandbranch'	Opens all descendants for display.
'collapsebranch'	Hides all descendants so they will not display. (See note 1.)
'collapsereset'	Removes all descendants. (See note 2.)

Note 1: If you collapse a node, its children are hidden but not necessarily collapsed, so that when you re-expand the node and display it, the node's grandchildren may or may not display. If you use the `collapsebranch` operation, the system collapses all the descendants of the selected node. If you then re-expand the node, its children are all collapsed so only the next generation displays.

Note 2: The `collapsereset` operation is intended for use with the `forceparent` state and the `onGetVirtualData` event handler to allow you to add children to the tree only when they are needed for display. If neither the `?forceparent` nor the `forceparent` state is set, the system sets `forceparent` when you use `collapsereset`.

SortChildren

The argument to the `SortChildren` method is the identifier of the node whose children you want to sort (or `'root'` if you want to sort the top level nodes), optionally followed by one or both of `'recurse'` or `'onSort'`. If you specify only the node identifier, the system uses the labels and sorts the immediate children nodes alphabetically. If you specify `'recurse'` the system recursively sorts all the children's children. Specifying `'onSort'` invokes the `onSort` handler to determine the relative order of the children. `❏wi` returns 1 if successful, 0 otherwise.

Note: You should be cautious in using this method when the tree is closed. The system cannot query a closed tree for labels if you have any marked as virtual. It uses the last value from the `onGetVirtualData` event handler, if any; otherwise it uses an empty vector. Using the `onSort` argument also is ineffective when the tree is closed.

EnsureVisible

The `EnsureVisible` method ensures that a node is visible in the tree, expanding its parent node if necessary and scrolling the tree if necessary. The argument to this method is the identifier of the node you want to make visible. The system returns 1 if the parent was already expanded but the node was scrolled to bring it within the visible portion of the tree; it returns zero otherwise. If the tree is closed when you invoke this method, the system signals an error.

GetInfo

You can query the specification for one or more nodes using this method. The first argument is one of the operation names detailed below. The second argument is the identifiers of the nodes you want to query. You can specify one or more nodes. You can also specify one node with some or all of its children by using a third argument. If you specify a third argument, you may specify only one node in the second argument.

The third argument specifies the depth of children to return; 0 means no children (the result will have 1 row), 1 means just the node's immediate children; 2 means children and grandchildren; etc. You can specify `-1` for all descendants. The result returned is a vector or matrix depending on the information

requested. If you specify the depth argument, the result is a matrix that includes the node identifiers as the first column, depth as the second column, and the requested information in subsequent columns.

You can use the following operations:

- 'depth', 'list', 'label', 'images', 'checkstate', 'highlightbold', 'highlightcut', 'highlightdrop', 'highlightselect', 'expanded', 'expandedonce', 'forceparent', 'haschildren', 'wherelabel', 'whererow', or 'handle'
- The 'depth' operation returns an integer vector containing the depth for each node.
- The 'list' operation returns a matrix with one row for each node and one column for each of elements 3, 4, and 5 from the list property: label, image indices, and states.
- The 'label' operation returns a vector of character vectors containing the text labels for the nodes.
- The 'images' operation returns a matrix with one column for each of the non-selected, selected, overlay, and user state image indices.
- The 'checkstate' operation returns a vector showing the user state image index for each checkbox: 3 if checked, 2 if not checked.
- The 'highlightbold', 'highlightcut', 'highlightdrop', 'highlightselect', 'expanded', and 'expandedonce' operations each return a Boolean vector denoting the specified state of the nodes.
- The 'forceparent' operation returns an integer vector with the force-parent state of each node. A 0 means normal: the behavior of the node is determined by whether the node has children. A 1 means the node behaves as if it has children whether or not it really does. A negative one (-1) means the GetVirtualData event is triggered to determine how the node behaves.
- The 'haschildren' operation returns a Boolean vector identifying whether or not each node has children. (This is not affected by the forceparent state; it is based on the actual existence of child nodes.)
- The 'wherelabel' and 'whererow' operations return a numeric matrix with the position and size of each node's label or entire row. The matrix has four columns containing the horizontal and vertical coordinates and height and width, respectively, of the appropriate part of the node in the client area of the Tree object using its current scaling units. If a node is not visible within the client area of the Tree, but could be scrolled into view without expanding any node, the result reflects the node's current position within the scrollable region relative to the upper left corner of the client area (thus the position may be negative or larger than the size property of the Tree). If any of a node's ancestors is collapsed, that row of the result is all zeros. If the Tree is closed, an error is signaled.
- The 'handle' operation is for low-level programming; it returns the handle used internally by the control to identify the node. Certain operations may change this value (e.g. re-opening the Tree), so this value should be retrieved each time it is needed rather than stored in an APL variable.

SetInfo

You can reset the specification for one or more nodes using this method, except that you cannot change a node's low-level handle, its depth, the physical location of a label or a row, or whether it has children. The first argument is one of the following operations: 'list', 'label', 'images', 'checkstate', 'highlightbold', 'highlightcut', 'highlightdrop', 'highlightselect', 'expanded', 'expandedonce', or 'forceparent'.

The remaining arguments identify the nodes to modify and the new attributes for that node. You may specify them as a vector of node identifiers followed by a vector or matrix containing the new values or as a matrix whose first column is the node identifiers and whose subsequent columns are the new values. The following statements are equivalent:

```
'fmA.trB' ⍳wi 'SetInfo' 'highlightbold' (1 5 10) (1 0 1)
'fmA.trB' ⍳wi 'SetInfo' 'highlightbold' (3 2 ρ 1 1 5 0 10 1)
```


The result is a Boolean vector showing 1 if successful, 0 otherwise for each node you specified.

Parentage:

Tree objects are most often children of a Form, Frame, Page, or CommandBar. They may also be children of some control objects: Button, Check, Combo, Edit, Label, List, Option, Picture, Scroll, Spinner, Status, and Toolbox. Tree objects can contain Timers, Imagelists, or hidden Menus.

Example:

Tree objects can contain and display an enormous amount of information, but retain a manageable display size, as long as you can organize the information in a hierarchical arrangement. The user can collapse the nodes that do not concern him at the moment and expand only those that have interest.

The key to managing a tree in your program is the node identifier, which is a positive integer in the first column of the list property. If you build a complete tree only for displaying information, you can specify zero for each node identifier and allow the system to assign node numbers. However, if you alter the tree in your application, you may want to assign the node numbers yourself.

Many of the event handlers identify the node in `□warg` when the event is triggered; some of them, for example, `onChange`, identify both the current node and the node that previously had the focus. You can use the node identifiers to determine how to process any data associated with the tree.

The example below shows a simplified event-handler function for various operations involving a Tree object. Assume that the tree has three levels of nodes: the top level node is a customer, the second level is a category of product or service, and the third (working) level node represents a data record, for example, a purchase order. When the user clicks on a third-level node, the application pops up a temporary form ('fmEdit') to display and edit the record.

```

▽ treeevent;node;oldnode;depth
[1]   :select (c □wevent)
[2]   :case (c 'Change')
[3]   node←1+□warg
[4]   oldnode←2+□warg
[5]   :if 1=('fmEdit' □wi 'modified')
[6]       table[oldnode;columns]←a info from 'fmEdit'
[7]       'fmEdit' □wi 'text' ''
[8]       'fmEdit' □wi 'modified' 0
[9]       'fmEdit' □wi 'enabled' 0
[10]  :endif
[11]  depth←('f.tr' □wi 'list')[node;2]
[12]  :if depth=3
[13]      'fmEdit' a info from (table[node;columns])
[14]      'fmEdit' □wi 'enabled' 1
[15]  :endif
[16]  :case (c 'Expanding')
[17]  □warg
[18]  node←1+□warg
[19]  :if 1=2+□warg
[20]      a unlock database records
[21]  :endif
[22]  :case (c 'Expanded')
[23]  node←1+□warg
[24]  0 0 p'f.tr' □wi 'SetInfo' 'expandedonce' node 0
[25]  :case (c 'BeginLabelEdit')
[26]  node←1+□warg
[27]  :if node≠2
[28]      □wres←0
[29]  :endif
[30]  :case (c 'EndLabelEdit')
[31]  node←1+□warg

```

#

```
[32]    table[node;3]←c[wres
[33]    :endselect
      ▽
```

If you manipulate the tree, you must have a method to keep track of node numbers. For example, if you use the `InsertNodes` method to add nodes, you must know the identifier of the parent node; you must also know the identifier of the sibling that you want to precede the new node, unless you are having the system sort the child nodes. Depending on the information in your tree, you can have the node numbers correspond to a row in a table or an element of a vector. Alternatively, you can build some intelligence into the numbering system, for example, by making all the topmost node identifiers be two-digit numbers, the second-level nodes be three digit numbers that correspond to the parent, such that the children of node 10 would be numbered 100, 101, 102, etc., the nodes at depth three four-digit numbers, and so forth.

#

Name:

System object

Summary:

A utility class, of which there is only one instance, to control high-level functions including closing the application.

Uses:

You cannot create an instance of the system object. You can, however, reference its properties with the octothorpe (#) symbol; for example:

```
'#' □wi 'size'
30 80
```

The `size` property shows the size of the Windows desktop in character units; the result varies depending on the display driver in use. Most of the system object's properties, including `size`, are read-only. Several change dynamically with your application. For example, the `focus` property shows which APL control, if any, currently has the focus; you can set this property to move the focus to a specified object. The `dragsource` and `droptarget` properties show APL controls that are currently involved in a drag-and-drop operation. If the user is dragging an item from another application or a different APL session, the `dragsource` property is blank. The `property` shows which APL control currently has the

You can also use the system object and its event-handler properties to control what happens when someone wants to shut down your application. If you set `□wres←0` in the `onClose` event handler, then the application does not terminate. You can also exit APL by invoking the `Close` method on the system object.

You use the `onNew` event handler when you create your own object classes. See the [Creating User-Defined Classes](#) chapter in this manual for information on this technique. See the [Windows Interface: Event-Handlers](#) and [Windows Interface: Methods](#) chapters for descriptions of the other events and methods.

The system object is the (implied) parent of top level objects, which can be `Form`, `MDIForm`, `Imagelist`, `Printer`, and `Timer` objects. A `Menu` that is called as a popup menu can also be a top-level object.

Regular Properties:

children, class, classes, contact, crashList, crashLogEvent, crashLogMode, crashLogPath, data

debugLogEvent, debugLogPath, dpi, dragsource, droptarget, empty, errorLogEvent, errorLogFilter, errorLogPath, eventLogName, enabled, events, focus, hwndmain, hwndsplash, links, methods, missing, name, newclasses, null, palette, properties, recursions, restartArg, restartCmd, restartDir, restartMsg, restartRate, self, server, sessions, size, themeactive, themed, thememode, tooltip, tooltipenabled, tooltipstyle, tooltiptime, tooltipwidth, units, version, workarea, xclasses, and Δudp .

sessions

The sessions property allows you to detect under program control changes in the session manager editor window.

themeactive (read-only)

Returns the boolean value of the Microsoft IsThemeActive() API. A 1 means a visual style for the current application is active.

themed (read-only)

Returns the boolean value of the Microsoft IsAppThemed() API. A 1 means that a visual style can be turned off in the Windows Control Panel, so that an application can support visual styles but not have a visual style applied at a given time.

thememode

Controls how visual styles are applied to the application windows. This property corresponds to the Microsoft GetThemeAppProperty and SetThemeAppProperty APIs. The value of the thememode property is the sum of the following values or zero:

Values for the System Class thememode Property

Value	Description
1	Specifies that the nonclient areas of your application windows have visual styles applied. Turning off this value will give you the Windows classic look in the borders of your top-level windows. (See note 1.)
2	Specifies that controls in your application have visual styles applied.
4	Specifies that all web content displayed in an application is rendered using visual styles.

Note 1: The default value is 3 for Windows XP and later. Setting this value to 1 gives you the visual styling of the non-client window frames around the forms but the non-visual styling for the child controls created on the forms after having set the property. This is more or less the same visual effect you have in past versions of APL64.

The visual styling used for a window depends on the value of this property that was in effect at the time the window was created. After a window is created, changing this property does not affect how it is displayed.

Event-handler Properties:

onAction, onClose, onComAction, onDestroy, onEditEnd, onEditNL, onEditLoad, onEditSaved, onEditStart, onExitError, onHelp, onNew, onPrompt, onResource, onSystemChange, and onTooltip.

Methods:

Close, Defer, Event, FormatMessage, GetSession, HwndToName, Notify, Printers, Ref, ReleaseObjects, Reset, Set, SetLinks, UseObject, VT, VV, and XInfo.

GetSession

The GetSession method can retrieve the contents of an edit session under program control.

Chapter 2: Windows Interface: Properties

Properties of objects are characteristics or attributes that determine how the object looks, where it is located and how it reacts. You do not have to set the value for every property. You may not need some properties, and the default values may be satisfactory in many cases. The properties in this chapter are listed alphabetically. They include:

activeform	deferexit	highlightcut	minsize	rtf	tabparent
apldata	description	highlightdrop	missing	savecontent	tabrows
autocomplete	dirty	highlightfocus	mode	scale	tabstop
bars	dock	highlightselect	modified	scrollaccel	targetformats
bin	dockable	himage	modifystop	scrollmargin	text
bins	dockbreak	hwnd	monthdelta	searchstring	tickinterval
bitmap	dockheader	hwnddropdown	msversion	selalign	tickpos
bitmapsizes	dockideal	hwndmain	name	selbullet	ticks
border	docklength	hwndsplash	newclasses	selcolor	today
buddy	dockmargin	icon	noredraw	selection	tooltip
canpaste	dockmax	image	normsize	self	tooltipenabled
canredo	dockmin	imagealloc	null	selfont	tooltipwidth
canundo	dockmindepth	imagecount	obj	selfontall	translate
caption	dockside	imageindex	opened	selindents	undolimit
captiondock	dockvert	imagelist	order	selnumstart	undoname
captionfloat	dragimage	imagelistdisabled	orientation	selnumstyle	units
channelwhere	dragsource	imagelisthot	origin	selnumtab	value
children	dropcolor	imagelistlarge	overlays	selfpargall	version
class	dropfont	imagelistuser	padding	selfrtf	viewalign
classes	droptarget	imagenames	page	selfstyle	viewmode
clienthwnd	edge	imagesize	pages	selfstyleall	visible
clientwhere	empty	imagespace	palette	selftabs	where
clsid	enabled	increment	paper	selftext	width
collate	events	indent	papers	separator	workarea
color	extent	instance	pending	server	wrap
colordepth	file	interface	place	shortcut	xclasses
columndisplay	firstday	interval	pointer	showtext	zoom
content	fixedtabsize	keys	position	siblings	Δ udp
copies	floatlayout	labeledithwnd	poslargeimage	size	ActiveX array
count	floatsize	limit	possmallimage	sliderlen	
data	floatwhere	limitwhere	printrname	sliderwhere	
ddeApplication	focus	links	progid	sortorder	
ddeItem	font	list	prompt	sourceformats	
ddeMode	format	margin	properties	state	
ddeTimeout	hdc	maskcolor	range	status	
ddeTopic	help	mediahwnd	recursions	style	
def	helpcontext	mediatype	redoname	suppress	
default	highlightbold	methods	roworigin	tabgroup	

The properties an object has depend on its class. Some properties belong to every object (except #, the system object, which is a special case). Every object has a **class** property, which you establish when you create the object. You cannot change **class**. Each object also has some read-only properties which are inherent to the class; for example, the **methods** property, which lists all the methods for that class. The list of methods is fixed.

Each object also has a **state** property, which lists the values of the properties of that object that have a value other than the default value. You cannot set the **state** property directly, but the system updates it when you set a property, when a user of your application takes an action that changes a property setting, and when your program performs an action that affects a property setting. Read-only properties are noted in the headings in this chapter. If the name is not annotated, the property is read/write.

Important Note: Names of properties begin with a lowercase letter. They are case-sensitive in APL64.

activeform

Description:

Character vector that is the fully-qualified name of a form that is active or that will be active when the MDI form with this property is opened.

Class: MDIForm.

Remarks:

You can reference this property to determine which form is active or set it to make a form active. This differs from the **focus** property of the system object in that it applies only to a form; when a control object has the focus, this property still applies to the form that is the parent.

This property exists only within the context of an MDI form. If your application has more than one MDI form, each has the property independently.

apldata

Description:

Boolean singleton that controls how character data is transmitted to and from an APL ActiveX server object.

Classes: ActiveControl and ActiveObject.

Remarks:

The values for the **apldata** property are:

0 = Old style data handling (character matrix and higher rank arrays are split into arrays of vectors and character scalars are converted into one-element vectors.)

1 = Character data is transmitted without modification.

This property can only be set on the APL ActiveX server objects APLW.WS and APLW.WSEngine. The default value is 1 for APLW.WS and 0 for APLW.WSEngine.

autocomplete

Description:

Integer scalar that defines the behavior of the AutoComplete feature.

Classes: Combo and Edit.

Remarks:

If you set this property to a non-default value, as the user types, the Windows system will pop up a small window like a tooltip with choices that match the letters typed so far. If the user presses the Enter key, the entire string is inserted into the edit box as the text.

The default value is zero. Non-zero values allow you to specify various sources from which the candidate strings are taken. As with a style property, you can specify multiple sources by setting a value that is the sum of the individual values.

Note that this feature is not compatible with a multiline edit control that accepts the Enter key as a character. The autocomplete property intercepts the Enter key and inserts the suggested string, even if the string is empty.

The values for this property define the following items as sources for the suggested strings:

- 1 Include the file system and the rest of the shell (Desktop, Computer, and Control Panel, for example).
- 2 Include the URLs in the user's History list.
- 4 Include the URLs in the user's Recently Used list (this includes entries in Start/Run, for example).
- 16 Include the file system only.
- 32 Include the file system directories, UNC servers, and UNC server shares (but not files).

The values 1, 16, and 32 are variations on the same set, so they should not be used together. Thus, any odd value 17 or greater is improper, as is any value greater than 46.

bars (read-only)

Description:

Nested vector of character vectors that lists the names of any command bars that are children.

Classes: Form, Frame, MDIForm, and Page.

Remarks:

This property is part of an object that has a window on the screen and that can have a CommandBar as a child. It is useful primarily for saving the state of the container object.

barwrap

Description:

Boolean flag that specifies whether children of the command bar causes its parent command bar to wrap to a new line.

Classes: All visible classes except CommandBar, Form, MDIForm, Menu, Page, and Printer.

Remarks:

This property is a synonym for the wrap property on the command bar.

Values for a barwrap Property

Value	Description
Primary Value: Choose only one value	
-1	Explicitly not wrapped (never wrapped)
0	Implicitly not wrapped
1	Implicitly wrapped
2	Explicitly wrapped (always wrapped)

The setting is effective both for docked and floating command bars. Note that when you float any bar with multiple children, the underlying Windows system may display it on multiple layers.

Do not confuse the effect of setting the barwrap property on a command button with the effect of setting the dockbreak property on a command bar. The former causes the next child of the bar to wrap to a new line within the same bar; the latter causes the entire bar to move to a new band on the side where it is docked.

bin

Description:

Integer designating the current paper bin; one of the numbers in column 1 of the bins property.

Class: Printer.

Remarks:

The default value varies among printers.

bins (read-only)

Description:

Two-column matrix, consisting of bin number and bin name as recognized by the printer driver of the current printer with which this object is associated.

Class: Printer.

Remarks:

A "bin" in this context is a paper tray (such as Multipurpose Tray or Lower Cassette) or a method of operating the printer, for example, Manual feed. This property shows the possibilities for the current printer. The bin name is merely descriptive; you can reference or change the current setting using the bin property.

bitmap

Description:

Character vector that specifies a device independent bitmap or, for a Picture control only, an array of encoded color values each representing one bit. The `bitmap` property provides the initial source for the image property that the system displays in a Picture control. It provides a background for a command bar.

Classes: CommandBar and Picture.

Remarks:

For a Picture control, if you use color values, the value for each color must be the scalar value calculated using the base value primitive function: `256 - blue green red` (reversed order from the vector representation of red-green-blue).

The color values specified are relative to the current palette, if one is defined. (See the description of the `palette` property of the system object later in this chapter.)

If you have multiple command bars docked on the same edge of a container, and they use the same bitmap, the image is drawn over the bars as a unit, not on each individual bar.

Examples:

```
'fmArt.picFlag' ⌵wi 'bitmap' 'oldglory.bmp'  
trident ← (50 10 ρ 255), (50 10 ρ 16777215), 50 10 ρ 16711680  
'fmArt.picFlag' ⌵wi 'bitmap' trident
```

bitmapsizes (read-only)

Description:

Two-element vector that shows the height and width of the bitmap, if one is defined, in the `bitmap` property of a Picture object.

Class: Picture.

Remarks:

The `bitmapsizes` property is a read-only property that reports the size of the most recently set bitmap in the scaling units defined for the control. If `bitmap` has not been set or if the Picture object (or the form containing it) is closed, `bitmapsizes` is an empty vector.

border

Description:

Numeric vector whose values are summed or a numeric singleton that specifies the border decorations that surround an object. The values of this property for a Form object also imply some things about the behavior of the form.

Classes:

Form, primarily. Also, Button, Check, Combo, CommandBar, DateTime, Edit, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

For all classes except Form and MDIForm, this property can be only a thin border (1) or no border (0). The

default is appropriate in most cases; for example, an Edit control, where the user enters text, has a border, but a Label does not.

For some objects, adding a border detracts from its appearance. If you want to enclose a set of control objects on a form (for example, a group of radio buttons), you can use a **Frame**. If you want to use a nondefault value, you should set this property when you create the object. If you reset the border of an object, Windows re-creates the object. If the object has the focus when you do this, the focus may move to another control.

Control objects also have an **edge** property that determines the appearance of the inside edge of the control, analogous to how the **border** property determines the outside edge. See the description of the **edge** property later in this chapter. The discussion below applies to **Form**.

You use this property with all its values for a **Form** object that comes up as a distinct window. The table below shows the values for the **border** property for a **Form** object. You can use one of the primary values plus any of the add-on values. These values also hold true for **MDIForm**, but you have less reason to change the default on an **MDIForm**.

Values for a Form Class border Property

Value	Description
Primary Value: Choose only one value	
0	No border
1	Thin border
2	Thick (sizable) border
3	Modal border (obsolete; now same as 1.)
Add-on Values: Choose none, any, or all of the values	
16	System menu and Close button
32	Minimize button (only with 16)
64	Maximize button (only with 16)
128	Title bar without buttons (and possibly captionless)
256	Context help "?" button (with 16, but not with 32, 64 or 512)
512	Windows tool palette (not with 32, 64, or 256)

If you include 16, the system places and allows items in a form's title bar that enable the user to manipulate it. You must have the system menu and the Close button to get the minimize and maximize buttons; if you choose one of the min/max buttons, both appear but the other is disabled. You must have the system menu and the Close button, but not the minimize/maximize buttons nor the palette to get the context help button.

If the form's **border** property does not include 16 or 128 in its sum and you set the **caption** to an empty vector, then the form does not have a title bar, in which case the user cannot drag it, although it still can be resizable. If you want a form with no border, **caption** must be empty; if you want a form with a blank title bar and a thin border, use border value 128 and an empty **caption**. You make a form modal by using the **Wait** method, but you can no longer have the same visual effect.

If you want to provide mouse-accessible context sensitive help for the children of a form, you can replace the minimize and maximize buttons with the context help button, which has a question mark as its caption. If a user clicks on this button, the cursor changes to a pointer augmented by a large question mark. When the user next clicks on a control on the form, it can trigger an **onHelp** event handler, either on that object or an object in its parentage chain. You do not have to use the context help button for this facility; pressing the F1 key triggers the same event. See the description of the **onHelp** event handler for more information.

You can create a tool palette by adding the value 512 to the border property of a form. You cannot combine this with minimize/maximize buttons. If you include 16 in the border property, you get a Close button but no system menu. A tool palette does not appear in the taskbar or in the Alt+Tab task list, unless you include 32 in the style property. The tool palette's appearance also differs from a standard form in that the title bar and its caption are smaller in height than normal. Note that this functionality came before command bars. You can create a richer toolbar using a floating command bar and associated command buttons.

buddy

Description:

Character vector that specifies the name of the control with which a spinner is associated. Selecting certain style values causes the spinner and its buddy to be connected either by location or value or both.

Class: Spinner.

Remarks:

If you associate the control by location, the system aligns the spinner at the side of the buddy control and makes them approximately the same height. If you associate them by value, the value property of the spinner changes in concert with a corresponding property (text, value, or caption) of the buddy control; that is, if you change the value of either control, the other changes to match.

In order for a control to be associated with a spinner, they both must have the same parent. The associated control must already exist when you assign the buddy property. If you take an action that causes the associated control to close and reopen, you should close and reopen the spinner, or reset the buddy property, or you may lose the association. If you move the buddy control, you may retain the value association, but the spinner does not automatically move in tandem with its buddy.

canpaste (read-only)

Description:

Boolean flag that tells whether the Windows clipboard has anything that can be pasted in a RichEdit control.

Class: RichEdit.

Remarks:

You can use this read-only property to decide whether to enable or disable a Paste menu item. The Paste method causes Windows to insert content from the clipboard. Note that such content may be inappropriate for a RichEdit control; for example, it could be an image file. This flag does not guarantee text.

canredo (read-only)

Description:

Boolean flag that tells whether Windows can redo the last operation that was undone in a RichEdit control.

Class: RichEdit.

Remarks:

You can use this property to decide whether to enable or disable a Redo menu item. The Redo method causes Windows to redo the last action that was undone. If there have been multiple actions undone, you can redo multiple actions. Any user or program action other than Undo that changes the content of the control zeroes this flag.

canundo

Description:

Boolean flag that tells whether Windows can undo the last editing operation in a RichEdit control.

Class: RichEdit.

Remarks:

The operation to be undone can be a user action or a program action. You can use this property to decide whether to enable or disable an Undo menu item. The Undo method causes Windows to undo the last action. You can set the canundo property to zero to render the Undo method ineffective; you cannot set canundo to 1.

caption

Description:

Character vector that specifies the text that labels an object or the choice if you select the object.

Classes:

Button, Check, CommandBar, CommandButton, Form, Frame, Label, MDIForm, Menu, Option, Page, and Printer.

Remarks:

The caption for a form or an MDI form displays in the title bar at the top of the form. This same text is shown with the form's icon when you minimize it. The default caption is the name of the form.

The caption for a control object is the text shown on or next to the object. For some objects, if you place an ampersand (&) in the caption text, the system underscores the character immediately to the right and defines it as an access key. The user can then press the Alt key followed by this key to access the control object without using the mouse. For those objects that have an onClick handler, this action clicks the object (Button, Check, CommandButton, Menu, Option). For a Frame, this action moves the focus to a child of the frame. For a Label, there is no effect.

For Form, MDIForm, CommandBar, Page, and Printer, the ampersand is a normal character. For those that accommodate an access key, if you want an ampersand to actually appear in the caption, you must enter two: `□wi 'caption' 'A&&P'`

You can also provide a hint about a Menu object's shortcut key combination in its caption. To do so, place a Tab character (`□tcht`) between the caption text and the text for the shortcut hint. When you set the shortcut property, the system automatically modifies the menu's caption to reflect the shortcut key names (for example, Ctrl+O). If you do not want the shortcut hint to show or if you want to specify it differently (in another language, for example), modify the caption property after you set the shortcut property. (See the description of the shortcut property later in this chapter for more information on shortcuts.)

The caption for a Printer object is the name that appears in the Windows Print Manager for the job and on the title page of the output on a network printer that is programmed to print a title page. It may be required to initiate a print job on a networked system.

The caption for a CommandBar object may appear both when a bar is floating and when it is docked, or it can be an alternative to either the `captiondock` or `captionfloat` property; that is, if one of those two properties is set, this caption shows for the opposite state. If both of the latter properties are set, the caption property is superfluous.

The caption for a `CommandButton` object appears only when the `showtext` property for its parent command bar is set to 1. The caption for the last command button cannot be empty. Under certain circumstances you may have to set the parent's `showtext` property to zero and reset it to 1 for the command buttons' captions to appear.

The caption for the `System` object is deprecated in APL64. The replacement is the new `⎕devcap` system variable. The new caption appears in the title bar to the left of the workspace path for your APL64 session. This is particularly useful when you have multiple APL sessions running. For example,
`⎕devcap←'my caption'.`

captiondock

Description:

Character vector that specifies the text you want to appear as the label for this command bar only when the bar is docked. The default value is empty.

Class: `CommandBar`.

Remarks:

If this property is empty, the contents of the `caption` property appear when the bar is docked. A mutually exclusive property for the same bar is `captionfloat`.

captionfloat

Description:

Character vector that specifies the text you want to appear as the label for this command bar only when the bar is floating. The default value is empty.

Class: `CommandBar`.

Remarks:

If this property is empty, the contents of the `caption` property appear when the bar is floating. A mutually exclusive property for the same bar is `captiondock`.

channelwhere (read-only)

Description:

Four-element vector that specifies the position and size of the trackbar's channel. The elements, in order, are row, column, height, and width (similar to the `where` property).

Class: `Trackbar`.

Remarks:

The values are in the scaling units of the trackbar and relative to its upper left corner.

children (read-only)

Description:

Nested vector of character vectors of the names of all objects that are immediately below this object in the hierarchy of fully qualified names.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, Tree, and #.

Remarks:

This property tracks all instances of objects that have the name of this object as the next-to-last part of their fully qualified name. The matrix of names of children is only one level deep.

Example:

Suppose you have an instance of a form named Stop that includes two buttons named Stop and Go and a frame named Light with three options. The value of the children property of Form is a nested vector:

```
fmStop.bnStop fmStop.bnGo fmStop.fLight
```

The value of the children property of the frame is also a nested vector. You can display it as a matrix using the Disclose function.

```
> 'fmStop.fLight' □wi 'children'
fmStop.fLight.opRed
fmStop.fLight.opYellow
fmStop.fLight.opGreen
```

class (read-only)

Description:

Character vector that is the name of the type of object; the template from which the system creates an object.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

The class property records the object class for each instance of an object. For a form, the class is always Form; for the controls on a form, the class property value is one of Button, Check, and so forth. This is a read-only property that you set for the life of an object when you define it using the New or Create method.

For an ActiveX class, the value returned is an alphanumeric string that begins with the brace symbol ({). This is sometimes called the GUID for an ActiveX object. You can use this string to create an instance of such a class. If the control is licensed, the license key may follow the GUID. The plain text name that is associated with the ActiveX object is in the description property.

classes (read-only)

Description:

Nested vector of character vectors that lists the classes available in the system.

Class: # (the system object).

Remarks:

This property provides a read-only list of the all the built-in APL64 WI classes you can use; you can reference it directly. The list includes `ActiveControl` and `ActiveObject`, which are categories of ActiveX classes that you can identify with the `xclasses` property or the `XInfo` method.

clienthwnd (read-only)

Description:

Handle of MDI client window; used for low-level programming.

Class: `MDIForm`.

Remarks:

When Windows creates an MDI form, it actually creates two objects; one is the MDI form itself, and the second is a separate window that exists in the client area of the MDI form (within its borders excluding the title bar). The second window is known as the MDI client.

When the system creates a displayable object, it creates a window handle that identifies the visual aspects of the object to Windows. If you are familiar with the Windows API, you can use this handle as a parameter to Windows API calls to gain explicit control over the window that represents the object. The MDI form has an `hwnd` property; the MDI client has the `clienthwnd` property.

clientwhere

Description:

Initially, an empty numeric vector; you can set this property to a four-element vector to restrict the MDI client to a specified region.

Class: `MDIForm`.

Remarks:

When Windows creates an MDI form, it actually creates two objects; one is the MDI form itself, and the second is a separate window that exists in the client area of the MDI form (within its borders excluding the title bar). The second window is known as the MDI client. Normally, the MDI client occupies the entire client area of the MDI form.

If you want to reserve a portion of the client area of an MDI form for something outside the MDI client, for example a toolbox or a status line, you can use this property to specify the size of the MDI client. This property works like the `where` property for any object; the first two elements specify the row and column of the upper left corner relative to the parent object's client area (which is the MDI form's client area in this case). The third and fourth elements are the height and width of the object in the scaling units of the MDI form.

clsid

Description:

Character vector that contains the globally unique identifier (guid) including braces that identifies an ActiveX control or server.

Classes: ActiveControl and ActiveObject.

Remarks:

For ActiveX controls, this is the value in the second column of the system object's xclasses property. ActiveX servers (ActiveObject classes) also each have a guid.

collate

Description:

Integer flag that tells the printer object whether to collate the copies of a multiple-copy print job.

Class: Printer.

Remarks:

If collate is set to 1, when printing multiple copies, the document is printed in page order multiple times. If you set this property to 0, all the copies of page 1 are printed before the first copy of page 2. This may significantly shorten the time required for printing. You can set collate to -1 to have the setting determined by Windows. See also the description of the PrintDlg method.

color

Description:

Numeric scalar or three-element vector, or a pair of scalars or nested three-element vectors or a single six-element vector, that specify background and/or foreground colors for an object. If you specify only one color, it is background; if you want to specify only the foreground (text) color, precede it with zilde. The system uses these colors for all subsequent output for the object. Setting this property causes the object to paint itself using the color values and triggers the object's onPaint event handler. Foreground colors are always solid colors; you do not get the "dithered" effect that you can with backgrounds.

For a Frame object, the color property applies only to the Frame's caption. For a normal Button object, the background color applies only to the dotted rectangle that surrounds the button when it has the focus. (You can define colors for buttons' text, face, highlight and shadow by using the Windows Control Panel's Display dialog. These are system-wide settings and affect all applications.) If you include 32 in a button's style property, the foreground color determines the color of the caption and the background color is ignored.

For RichEdit, Tree, and Status objects, the first element of the color property determines the color of the background; the system ignores the foreground color designation. For a Tree, this excludes the label text area where the system uses the Window Text and Windows colors for the text foreground and background. For a Status bar, this excludes the grip at the right hand side. For a RichEdit object, you can set the color of the text using the selcolor property.

For a Trackbar object, the first element of the color property determines the color of the background outside the channel area. The system ignores the foreground color designation. The system uses the color associated with 3D objects in the Windows control panel for the color of the slider; it uses the color

associated with 3D objects text in the Windows control panel for the color of the ticks. You may have to close and reopen the trackbar to see the effects of a color change.

For an MDI form, the color property affects the color of the child controls and forms that appear on the MDI form. It also affects the color of the area of the form outside the `clientwhere` area of the MDI form. However it *does not* affect the color of the MDI form's client area. This is always painted in the Application Workspace color as set by the Windows Control Panel.

For a Selector object, the color property has no direct effect, but if you set it, child pages inherit the settings. The system uses the color associated with 3D objects in the Windows control panel for the background color of the tabs; it uses the color associated with 3D objects text in the Windows control panel for the text color.

For a Progress object, by default Windows uses the Highlight (selected items) color for the progress bar blocks, so the appearance will vary depending on the user's choices. The background color is the color associated with 3D objects in the Windows control panel. You can set these colors explicitly.

For a DateTime object, you can specify up to six color values. Each color may be specified as an enclosed three-element vector or a singleton composite number. Unlike the color properties for other classes, a three-element vector that is not enclosed specifies three colors rather than a three-element single color. Similarly, a six-element integer vector specifies six colors. The six colors are: background within the month; text within the month; background for the title; text for the title and selected days; background between months; and text for leading and trailing days around the month.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

The color property specifies the colors for an object using composite color values. You specify a composite color value as three values of 0 to 255 for each of red, green, and blue. For example, bright green would be expressed as 0 255 0, black as 0 0 0, and bright white as 255 255 255. A single color values can be expressed three ways:

- a three-element integer vector
- three-element integer vector nested as a scalar.
- a scalar formed by using the base value primitive function with the three values to the base 256; that is, $256 \perp \text{blue, green, red}$ (note the reversed order)

The following three examples all achieve the same result: setting the label object named `lMessage` to have a bright red background and a bright blue foreground (the text). Note the RGB reversal in the argument to $\perp$.

```
␣wi 'lMessage.color' 255 0 0 0 0 255
␣wi 'lMessage.color' (255 0 0) (0 0 255)
␣wi 'lMessage.color' (256  $\perp$  0 0 255) (256  $\perp$  255 0 0)
```

When you use the special values shown below, you are using the color specified in the Windows Control Panel choice. This Windows dialog allows the user to choose colors for the indicated Windows elements. By using these special color values you can provide objects that use the end user's own custom color choices.

Special color Property Values

Value	Description	Value	Description
-100	Scroll Bar	-113	Highlight Background
-101	Background	-114	Highlight Text
-102	Active Caption	-115	Button Face

Value	Description	Value	Description
-103	Inactive Caption	-116	Button Shadow
-104	Menu	-117	Disabled Text
-105	Window	-118	Button Text
-106	Window Frame	-119	Inactive Caption Text
-107	Menu Text	-120	Button Highlight
-108	Window Text	-121	Dark shadow for 3D display elements
-109	Caption Text	-122	Light color for 3D display elements
-110	Active Border	-123	Text color for tooltips
-111	Inactive Border	-124	Background color for tooltips
-112	Application Workspace		

You can also specify the following special values for a composite color:

- 1 For a control object, you use the color specified for the form, frame, or panel on which the control is located. For a form, this means the color set for the Desktop object.
- 2 Use the color for this part of a specific class of object as defined in the Colors screen of the Windows Control Panel (the 25 choices listed above). For example, this means that the color selected for Caption Text in the Windows control panel applies to the text of the caption property for the object you are creating.
- ⊕ Do not change the present color value. You can use this as a placeholder when you change only the foreground color but do not want to change the background color value.

If you are changing only the background color, the color property value can be an integer scalar or a three-element vector. If you specify both the background and the foreground, the color property value can be any of:

- a 6-element integer vector
- a 2-element integer vector (using the base 256 representation)
- a nested vector of two composite color values
- a nested vector of two items, the first of which is ⊕ (which means make no change to the background color)

Examples:

This example sets a default background and a bright green foreground:

```
␣wi 'lMessage.color' -1 (0 255 0)
```

This example sets the foreground bright red without changing the background:

```
␣wi 'lMessage.color' ⊕ (255 0 0)
```

This example sets the background bright red without changing the foreground:

```
␣wi 'lMessage.color' 255 0 0
```

This example uses the highlight background and highlight text colors defined in the Colors screen of the Windows Control Panel:

```
␣wi 'lMessage.color' -113 -114
```

colordepth

Description:

Integer scalar that controls the number of colors and the palette used to represent images in an imagelist. The valid values are: -1 0 4 8 16 24 32.

Class: Imagelist.

Remarks:

The default value is zero; this specifies that the system use the default for the underlying Imagelist control, which is usually a 4-bit-per-pixel, device independent bitmap. However, it can sometimes be a device dependent bitmap.

You can use the value -1 to force a device-dependent bitmap. This value is recommended when you are using the imagelist only for controls on the screen; that is, for listview, tree, button. Using this value can result in a noticeable improvement in performance over using a device-independent depth.

If you are going to draw images on a Printer object, you should use the number that represents the highest color depth of any image in the imagelist. These values (4 through 32) designate the color depth in bits per pixel for a device independent bitmap.

Setting the colordepth property reinitializes the Imagelist; so if you have used the AddImages method, you must reconstruct the imagelist. If the images are listed in the imagenames property, the imagelist is automatically reconstructed with the changed depth of color.

Technical Note:

The ability to display graphics, bitmaps for example, with "bright" colors depends on the combination of the color depth of the video drivers and the support software. Color depth is basically the number of bits per pixel that describe the color of the image.

The Imagelist object was originally designed for graphics such as icons and low-color, small bitmaps. As resolution and color depth on systems have grown, bitmaps tend to be larger and are generated in multiple levels and with greater color depths. When systems with low color-depth desktops display such bitmaps, the images appear to be washed out or shadows of the intended image.

You can perhaps use the colordepth property to alleviate this problem somewhat. If you do not know what value to use, setting it to -1 produces the best results; zero is the default, however, to maintain existing behavior for those who do not wish to change. Setting this property to a positive value is most effective when the desktop is set for 16 million colors (2*24) or greater. Setting it is not very effective at all when the desktop is set to 256 colors or below.

When you develop applications that may run on different systems, some of which may be low resolution, you can use two sets of bitmaps for your images. You can query the resolution of the desktop by doing the following:

```
hdc=□wcall 'GetDC' 'HWND_Desktop'  
□wcall 'GetDeviceCaps' hdc 'bitspixel'  
16          A indicates 16 bits per pixel  
□wcall 'ReleaseDC' 'HWND_Desktop' hdc
```

You can detect when a user changes the desktop resolution by using the onSystemChange event handler on the system object.

columndisplay

Description:

Seven-column nested matrix that specifies the label, width, justification, index, and order for each column of information when the Listview is in report viewmode, and image index and alignment for the header.

Class: Listview.

Remarks:

To make full use of report view, you must define the attributes of the columns of information you want to display. Each row of this property defines the attributes of one column of information in the display. The text that displays is from the cells of the list property matrix.

The columns of this seven-column matrix are as follows:

- **Column Label**
A character vector containing the text you want to use as the column header in report view.
- **Column Width**
The width of the column in current scaling units. You must include enough width in the first column to show the images.
- **Justification**
A character vector defining how you want your information justified. (This element assumes character information; it may be extended in future releases to enable formatting of numeric data.) The allowable values are: 'left', 'right', or 'center'. The first column must be left justified.
- **Column Index**
A positive integer specifying the origin one index of the column of cells in the list property; that is, the value 1 refers to the third column of the list property, after the RowID and images.
- **Column Order**
A positive integer specifying the order in which you want the columns to display. The column specified as 1 in this element designates the primary cells column. The values in the primary cells column of the list property are used as the labels in smallimage view, largeimage view and list view, as well as being the leftmost column in report view.
- **Header Image Index**
The origin one index in the Imagelist to the image you want to display in the header for this column. Note that the height of the header is determined by the font. If the imagesize of the imagelist is larger than the header, you may need to create an image for the header that contains a blank border on the top and bottom to make the column header image display satisfactorily.
- **Header Style**
Currently this affects only the placement of the image in the header, but it may contain other style flags in the future. By default it is zero, placing the image to the left of the text. For any column other than the primary cells column, if you set this value to 1, the image displays to the right of the text in the column header. The primary cells column is the one designated by the first row of the columndisplay property; it holds the images, and the value of its cells display as labels in viewmodes other than report

The example below shows the columndisplay property for the picture of the listview in the Classes chapter.

```
'fmMain.lvSampleLV' □wi 'columndisplay'
order      15 left   1  1  0  0
image      10 center 3  2  0  0
large image 10 right  4  3  0  0
user image 10 left   2  4  0  0
```

When you set this property, you may specify only the first four columns. The fifth column (column order) will match the fourth (column index). The headers will be zero. When you reference the property, the system returns all seven columns.

contact

Description:

The data associated with the message box containing vendor contact information as listed in the [ContactInfo] section of the INI file.

Class: # (the system object).

Remarks:

You can use this property to query the contact information for your company such as the name, address, phone number and website , etc. This information will be shown in the message box that is displayed in the event of an unrecoverable failure in APL64.

Example:

In APLNOW32.INI file:

```
[ContactInfo]
APLNow LLC
PO Box 361
Brielle, NJ 08730
.
www.apl2000.com
301-208-7150
```

In APL64:

```
'#' [wi 'contact'
APLNow LLC
PO Box 361
Brielle, NJ 08730

www.apl2000.com
301-208-7150
```

content

Description:

Integer vector that contains the internal state of ActiveX properties for an ActiveX control.

Class: ActiveControl.

Remarks:

You can use this property for an ActiveX control similar to the way you use the def property for other controls. You can reference it to get the current state of the object, if the object is open, or the most recent state, if the object is closed. You can set it only by assigning a value that you obtained by referencing the

property earlier. **Warning:** If you attempt to assign other values, unpredictable and possibly damaging behavior may result.

The content property does not include event-handler bindings, nor does it contain values for properties that APL64 adds to the object. It does not appear in the `state` property for an object, but it is saved in `def`; thus you can save content in a `MAKE` function by referencing it and setting its value when you make the control.

You can control how and when the content property is saved by using the `savecontent` property.

copies

Description:

Integer that specifies how many copies of subsequent pages are to be printed. The default value is 1.

Class: Printer.

Remarks:

You must set this property before you place anything on the page to which you want it to apply. When you set `copies`, the Printer starts a new page. The value of this property can also be set by a user through a Print Dialog or a Page Setup interface. See also the description of the `PrintDlg` method.

count (read-only)

Description:

Two-element integer vector that shows the number of entries in the list of a Listview or the number of nodes in a Tree.

Classes: Listview and Tree.

Remarks:

For a Listview control, the first element returns the total number of entries in the list. In list view or report view, the second element is the number of entries that will fit fully visible in the client area of the Listview; in largeimage and smallimage views, the second number matches the first.

For a Tree control, the first element returns the total number of nodes in the Tree and the second returns the number of nodes that will fit fully visible in the client area of the Tree. (For some heights, you may think you can see one node more than this property reports.)

data

Description:

Arbitrary data associated with the object.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

These data are not displayed on or with the object; they are out of the user's sight. The value of `data` can be

any APL array. Possible uses for this property are storing the value of the data previously displayed in a field, so that you can provide an undo feature, or storing a default value for a field.

You can also create additional user-defined data properties for any object. See the description of `Δudp` near the end of this chapter.

crashList

Description:

Character vector of variable names that are entered into the system crash log file when the system terminates abnormally (crashes) or when a dump file is produced due to an error or the `Quit 'Dump'` command.

Class: # (the system object).

Remarks:

The character vector is a space delimited list of variables whose values are to be included in the dump. This list is defined in ANSI (and converted by `⊞AV` mapping) when stored in the INI file. When set via the system object's `crashList` property, the names are in APL character set to begin with.

Names can optionally be prefixed by a down-arrow (`↓`) to indicate you want to use the global definition. If you don't specify the down-arrow prefix, the localized value of the name (current value in symbol table) is displayed. The down-arrow (`↓`) prefix applies only to the next name following it. This is illustrated in the example further below where `GlobalA` has a prefix and `LocalA` does not. So `GlobalA` displays the global value while `LocalA` displays the local value.

Value are displayed in "value tip format" for up to 1024 characters by default. In addition to name specifications, you can insert "`#n`" codes (such as `#10000`) to indicate how many character of output you want to represent each object's value tip display). This defaults to `#1024` if not specified. Any `#n` settings take effect for the remaining items in the list. If the list begins with an `#n` specification, it is used to determine how long the value tips are for values displayed in the SIE stack. For example, to display SIE value tips as length 100000 and dump the values of `GlobalA` and `LocalB` each with up to 1000000 character tip lengths, specify this:

```
[Config]CrashList=#100000 #1000000 ↓GlobalA LocalB
```

crashLogEvent

Description:

Character vector that specifies the default Windows Event Log type of event used to log system crash events.

Class: # (the system object).

Remarks:

The default specification is `F 4 4000` where `F` is Fatal Error, `4` is the category, and `4000` is the event id used to log system crash events. This property uses the same set of codes as the `⊞LOG` function. See the description of the `⊞LOG` function in the System Functions Manual for more information.

Note: This property has a corresponding INI configuration file parameter named `[Config]CrashLogEvent` that defines the default values at startup time

crashLogMode

Description:

Integer that controls the amount of information (Log File or MiniDump) recorded when a system crash occurs.

Class: # (the system object).

Remarks:

If a MiniDump options are selected, it affects the quality of debugging that can be done with that file. CrashLogMode can have the following values (the default is 2 = Log and MiniDump files):

Value	Description
0	do not create Log or MiniDump files.
1	Log file only.
2	Log and MidiDump (essentials plus extra information) ~ 5.0 MB + WsSize + other memory (can be very big)
8	Log and MidiDump (essentials plus extra information) ~ 5.0 MB + WsSize + other memory (can be very big)
9	Log and MaxiDump (everything in process space) ~ 50+ MB + WsSize + other memory (can be very big)

In most cases, CrashLogMode =2 will be sufficient to allow effective debugging of APL64 system crashes by APL2000 Technical Support. Note that higher level CrashLogMode settings should not normally be used to produce MiniDump files unless the customer is directed to do so by APL2000 Technical Support. The potentially huge increase in file size is seldom warranted unless the smaller dump files are insufficient to pin-point the cause of the problem. Note that CrashLogMode > 2 cause all committed workspace memory up to workspace size (WsSize) to be dumped to file. For a workspace with 1 GB of committed memory, this can easily create a file that is too large to transmit electronically while a CrashLogMode =2 file probably would have been sufficient to diagnose the problem or at least understand where the system crash is occurring.

When thinking about WsSize keep in mind that the size of a normal saved workspace is only a fraction of the working memory size that is committed while the workspace is running. So the size of the MiniDump file with CrashLogMode > 2 can far exceed the size of a saved workspace. Even workspaces saved with a suspended)SI are not as large as a CrashLogMode > 2 MiniDump file would be because the system removes all unused "free" space from them before storing them on the disk. In a MiniDump file, all memory in the committed workspace, whether actually in use or reserved free space, is stored.

Note: This property has a corresponding INI configuration file parameter named [Config]CrashLogMode that defines the default values at startup time.

crashLogPath

Description:

Character vector that specifies the location of the system crash log files.

Class: # (the system object).

Remarks:

This may be a fully qualified name or a relative to the ApplicationLogPath. The default value of this property is: .\Log\Crash\Crash-*.log. If it begins with a dot (.) it is relative to [Config]ApplicationLogPath parameter.

The name must include one star (*) character. The system substitutes a timestamp in place of the star when generating the system crash log file.

Note: This property has a corresponding INI configuration file parameter named [Config]CrashLogPath that defines the default values at startup time.

ddeApplication

Description:

Character vector that is the name of the application that APL automatically starts if no server responds to a requested conversation on the ddeTopic. Used in the client side of a DDE conversation.

Classes: Edit and Label.

Remarks:

This property contains the fully qualified path name of a Windows application's .EXE file. If no running application has registered itself as a server on the value specified for a control's ddeTopic, APL attempts to start the program named in ddeApplication. Using ddeApplication is a technique that works in a limited number of cases. If you are concerned about being able to have full control over how the server application is started, you may prefer to call the Windows function `WinExec()` using `□wcall` or `□na`.

If you are linking to another APL application, the name of ddeApplication would be the name of the APL interpreter's .EXE file (such as `C:\APLWINxx\aplw` or whatever you have renamed it to be). The default value for ddeApplication is the empty vector.

ddeltem

Description:

Character vector that defines the name of the item in a DDE conversation with another Windows application. Used in both the client and server sides of a conversation.

Classes: Edit and Label.

Remarks:

This property contains the name of a Dynamic Data Exchange *item*, an individual piece of data that one application sends to another in a DDE transaction. For example, a spreadsheet program treats the value of one cell in a worksheet as a DDE item.

The name of that item might be the row and column numbers that identify the cell. You could also specify an arbitrary name that identifies the contents of the cell; the spreadsheet would have to recognize the same name as a label for the cell.

You use the ddeltem property when your application is acting as either a client or a server. In a client application, you set the ddeltem property to a non-empty value on a Label or Edit control, along with the ddeMode and ddeTopic properties. You also set the ddeApplication property if not evident from ddeTopic. When all four properties are set, the control attempts to begin a DDE conversation with a server.

In a server application, you set the ddeltem property for Edit and Label objects that are children of a form; the form must have a non-empty ddeTopic property. Applications that request information of your application are linked to the appropriate control automatically, and the text in the control is used to satisfy

the DDE request from the client. If a client control will only be issuing commands (using the `DdeExecute` method) and not receiving data, you do not need to set `ddeltem`. The default value is the empty vector.

ddeMode

Description:

Character vector that defines the manner in which a DDE client receives data from a server application. Used in the client side of a DDE conversation.

Classes: Edit and Label.

Remarks:

The `ddeMode` property has three possible values: `'hot'`, `'cold'` and an empty character vector. Setting `ddeMode` to `'hot'` requests an *automatic* link. A server that agrees to provide this kind of link sends a data item to your client control immediately whenever the data item (specified in `ddeltem`) changes. The system puts the new data item into the control's `text` property, if the control is an Edit object, or into its `caption` property, if it is a Label. The action also triggers the control's `onChange` event handler, which your application can use to process the incoming data item.

Setting `ddeMode` to `'cold'` specifies a *manual* link. The server sends a data item to your client control only when your application requests it by using the `DdeRequest` method.

The `ddeMode` property must be non-empty for a conversation to begin. Setting it to empty breaks the link.

ddeTimeout

Description:

Integer that defines how long a DDE client waits for a connection with a DDE server. Used in the client side of a DDE conversation.

Classes: Edit and Label.

Remarks:

The value of this property specifies the maximum time in milliseconds that a control attempting to initiate a DDE conversation waits for a response from a server. The default is 5000; the maximum value is $2^{31}-1$. The `❏wi` call that initiates the DDE conversation does not return until a connection is established. If the timeout expires, `❏wi` signals an error.

ddeTopic

Description:

Character vector that defines the topic of a DDE conversation with another Windows application. Used on both client and server sides of a conversation.

Classes: Form, Edit, and Label..

Remarks:

This property contains the name of a Dynamic Data Exchange *topic*. For Edit and Label controls (DDE clients), `ddeTopic` usually also includes the name of a DDE *application*. The application name precedes the topic, separated by the vertical bar character (`|`); for example:

```
'fmMain.edEdit' ❏wi 'ddeTopic' 'application | topic'
```

Note 1: For the vertical bar, you can use either the stile (|) or the split stile (|).

Note 2: To enable DDE connectivity in your application, you must set the setting for [Config]EnabledDDE in the APLW.INI file to 1.

A client control can specify the application name separately in the ddeApplication property, which causes the system to start the remote application's .EXE file; for example, EXCEL or APLW. It can be any name for the service that an application provides. The topic name commonly refers to one document (a file or a data set) that the server application owns; for example, a spreadsheet program treats an individual worksheet as a DDE topic and the individual cells in the sheet as data items.

For a Form (acting as a DDE server), the application name is always the same as the currently executing .EXE file (for example, APLW). Its ddeTopic property specifies only the *topic*.

Assigning a non-empty vector to ddeTopic for an open Form, or opening a Form on which it is already defined, causes APL to register this form with Windows as a DDE server. Other applications can then initiate conversations with your application. Your application can register any number of DDE topics at once, but a different form is needed for each topic.

Example:

The following statement defines the ddeTopic for an edit control to be the Excel worksheet named c:\amortize.xls. The ddeItem and ddeMode are then set, and the form is opened, which causes APL to attempt to establish the DDE link.

```
'fmMain.edClient' □wi 'ddeTopic' 'Excel|c:\amortize.xls'
'fmMain.edClient' □wi 'ddeMode' 'Hot'
'fmMain.edClient' □wi 'ddeItem' 'R1C1'
'fmMain' □wi 'Open'
```

debugLogEvent

Description:

Character vector that specifies the default Windows Event Log type of event used to log debug events.

Class: # (the system object).

Remarks:

The default specification is F 4 4000 where F is Fatal Error, 4 is the category, and 4000 is the event id used to log debug events. This property uses the same set of codes as the □LOG function. See the description of the □LOG function in the System Functions Manual for more information.

Note: This property has a corresponding INI configuration file parameter named [Config]DebugLogEvent that defines the default values at startup time.

debugLogPath

Description:

Character vector that specifies the location of the debug log files.

Class: # (the system object).

Remarks:

This may be a fully qualified name or a relative to the ApplicationLogPath. The default value of this property

is: `.\Log\Debug\Debug-*.log`. If it begins with a dot (.) it is relative to [Config]ApplicationLogPath parameter.

The name must include one star (*) character. The system substitutes a timestamp in place of the star when generating the system crash log file.

Note: This property has a corresponding INI configuration file parameter named [Config]DebugLogPath that defines the default values at startup time.

def

Description:

Nested array that contains the internal definition of an object and all its children.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, and Tree.

Remarks:

The def property returns an array that is a snapshot of the internal state of an object that includes all of its property settings and handlers, plus the same information for all of the object's children.

You can use it to record the state of a form or part of a form in an APL variable that you can then use to recreate the form. Because the definition vector is an ordinary APL array, you can store it as a variable in the workspace or keep it in a component file.

Note: The information in a definition vector is in precompiled form; its structure is not documented, nor is it intended to be manipulated by APL. Any change to a definition vector renders it useless.

You can use the definition vector returned by referencing def to create an identical form or part of a form at some later time. You can create multiple clones of a form from one definition using different names (see the description of the New or Create method). You can also set the def property for an existing object. This causes the object and all its children to be deleted and replaced with the new definition. This works for parts of a form in addition to entire forms, providing a quick means to swap menu structures or groups of controls in forms that already exist.

Example:

In your application, suppose you had two menu items: one that displayed a frame where users could select colors, and another that displayed a frame where they could select textures. You want the frames to appear in the same place on your main form. The menu item the user selects determines which frame the application displays.

Build the frames and save their def properties in variables.

```
colorTemplate ← 'fmMain.fColors' □wi 'def'
txtrTemplate ← 'fmMain.fTextures' □wi 'def'
```

To create an object, use the def property as an argument to the New method.

```
sink ← 'fmMain.fchoose' □wi 'New' colorTemplate
```

To replace an object, reset the def property of the object you want to replace.

```
'fmMain.fChoose' □wi 'def' txtrTemplate
```

For top-level objects (Forms, MDIForms, Imagelists, Timers, and Printers) and for Menus, you can replace objects only with those of the same class. Although you can replace control objects with those of another class, it is less confusing if you limit replacements to those of the same class.

default

Description:

Boolean that tells whether the printer associated with this printer object is the default printer for the system.

Class: Printer.

Remarks:

If this property is zero, you can set it to 1 to make the current printer the default printer for the system. You cannot set this property to zero.

defaultFont

Description:

Vector that specifies the default font rendered in the `□wi` system.

Class: System (#).

Remarks:

This property improves the default font selection process for the `□wi` system when running in DPI aware mode. Instead of allowing Windows to choose a font size for the `□wi` system, APL does actively participate in the font selection process when running in `dpiAware=true` mode. This dramatically improves font size versus window layout behavior when running at high DPI scaling levels.

Font selection is controlled by testing several strings to see how they render at the selected DPI settings and comparing this to how they render at `dpiAware=false` setting. APL will tweak the font size smaller and smaller until none of the test strings come out wider than they would render if the `dpiAware=false` rendering were simply stretched to the DPI. This assures that text will not be wider than it was in the previous systems where `dpiAware=false` was the norm and was always rendered at 100% DPI in an 8x16 box size.

Here's an example that queries the property, showing its default value:

```
'#' □wi 'defaultFont'  
16 8 [MS Shell Dlg] 13 {Label Caption} 65 {WMXwmxiiii} 56
```

Here's an explanation of all the defaultFont values:

```
16 8 [MS Shell Dlg] 13 {Label Caption} 65 {WMXwmxiiii} 56 ... {MoreTestPhrases} MoreTestWidths  
^ ^      ^          ^ ^          ^ ^          ^  
| |      |          | |          | |          |  
\ /      |          | |          | |          |  
A        B          C  D          E  F          G
```

A - Optional character box size at 100% scaling (height width); (defaults to 16 8)

B - Optional font name (defaults to [MS Shell Dlg])

C - Optional standard font height at 100% scaling; (defaults to 81.25% of character box height in A)

D - Test Phrase whose width is to be checked
E - Width of {Label Caption} test phrase at 100% scaling
F - Test phrase whose width is to be checked
G - Width of {WMXwmxiiii} test phrase at 100% scaling

An empty font name "[]" implies the standard font name "[MS Shell Dlg]". In order to specify the standard font height you must specify a font name or a "[]" placeholder for the standard font name before the font height. In other words you cannot specify the font height like either of the following:

```
13
16 8 13
```

But it can be set like any of the following:

```
[] 13
16 8 [] 13
[Some Font Name] 13
16 8 [Some Font Name] 13
```

You can specify the first three fields without any test phrases, but it is recommended to always include one or more test phrase. They are used as font sizing constraints to make sure the font isn't selected that would cause these phrases to appear relatively wider at high DPI settings than they do at 100% scaling. The font height specification is only a hint about the proper font size to use. The system scales the specified font height specification by the DPI scaling level and uses this scaled height as a starting point for font sizing. For example, if the standard font height is 13 but Windows is running at 200% DPI scaling, APL will scale the 13 by 200% and begin with a 26 pixel high font. APL then tests each of the Test Phrases to see how many pixels wide they render with this proposed font height. If any of the test phrases render wider than their width constraints (after scaling the test width constraint by the DPI scaling factor) APL rejects that font size and pick another one that is one pixel shorter. This is repeated until APL finds a font size where all of the test phrase constraints are satisfied, and that becomes the font size APL actually uses. In the default case, APL has the following two test phrases and width constraints:

Test Phrase	Width
-----	-----
Label Caption	65
WMXwmxiiii	56

At 200% DPI scaling APL scales these width constraints up by 200% to 130 and 112, respectively. Then APL tests to make sure that the width of the test phrases do not exceed these scaled-up width constraints. If they do, APL picks a smaller font and repeat until a font size is found where all constraints are satisfied. Note that bogus width constraints could in theory force the font size to be too small to read, so APL has a lower limit of 5 pixels as the minimum font size that will ever be picked. This is too small to read, especially at high DPI settings, but that's the limit at which APL stops searching. This is just a safety limit, not a practical one. If you change the defaultFont, you are responsible for picking appropriate width constraints that are realistic. The best way of doing this is to run APL at 100% scaling and let APL compute the widths for you, as the test phrases would render at 100% scaling. The technique for doing this is illustrated below.

Here's an example that sets the default font property to a different application-specific value:

```
'#' □wi 'defaultFont' '{TestPhraseOne} 74 {TestPhraseTwo} 75'
```

In the case above some fields (such as character box size, font name, and tentative size) are omitted and only the width constraint fields are specified. If you query the property after setting it like this, the system

will return the full specification as shown below:

```
'#' □wi 'defaultFont'
16 8 [MS Shell Dlg] 13 {TestPhraseOne} 74 {TestPhraseTwo} 75
```

The defaultFont property should be set before any □wi objects are created. Any □wi object created before setting the defaultFont property may exhibit unexpected behavior if a different value is set after they have been created. APL does not prevent you from setting defaultFont when □wi Forms are already in existence, but you are discouraged from doing so.

Now the only remaining question is how to compute proper width constraints for application-specific test phrases. This is easily done by setting the defaultFont property using test phrases that don't specify width constraints, like this:

```
'#' □wi 'defaultFont' '{TestPhraseOne} {TestPhraseTwo} {TestPhraseThree}'
```

Then query defaultFont to discover the actual widths these strings would be at 100% DPI.

```
'#' □wi 'defaultFont'
16 8 [MS Shell Dlg] 13 {TestPhraseOne} 74 {TestPhraseTwo} 75 {TestPhraseThree} 82
```

In addition to the dynamically settable defaultFont property of the system object, the default font can also be specified statically by the [Config]DefaultFont parameter in the APLW.INI file. You can use the system object defaultFont property to dynamically experiment with width constraints (letting APL compute default widths for the test strings as they would be drawn at 100% DPI) but then store these settings in the application INI file so that subsequent APL session will configure the default system font the same way for subsequent sessions. If an application experiences font scaling problems at high DPI levels, it can easily resolve them by picking the offending string that is rendering too wide at some DPI settings and adding that string as a test phrase as outlined above. Then use this as a new application-specific [Config]defaultFont setting. After restarting the application, it will now pick a font that is guaranteed not to be too large to create that problem in the future. Applications should try to pick the best test constraints they can to achieve good scaling behaviors. But in doing so they should keep in mind that some users might experience problems that are not uncovered during pre-release testing. That's because different devices behave differently with respect to font mapping. If a user of the application reports a problem, along with a screen shot of the offending issue, the developer can easily compute new test phrase constraints and have the user add the new constraints to their INI file in order to resolve the scaling issue for that specific user, or perhaps add that constraint to the next release of the application so that all users can benefit from this in the future.

Note: The defaultFont property can be set with missing TestPhrase width specifications at any DPI. APL simply selects the standard height (unscaled) font and uses it to compute test phrase widths. For example, set the defaultFont property using test phrases that don't specify width constraints, like this:

```
'#' □wi 'defaultFont' '{TestPhraseOne} {TestPhraseTwo} {TestPhraseThree}'
```

Then query defaultFont to discover the actual widths these strings.

```
'#' □wi 'defaultFont'
16 8 [MS Shell Dlg] 13 {TestPhraseOne} 74 {TestPhraseTwo} 75 {TestPhraseThree} 82
```

These test phrases can then be put into the INI file like this, in [Config] section:

```
[Config]
```

```
DefaultFont={TestPhraseOne} 74 {TestPhraseTwo} 75 {TestPhraseThree} 82
```

Note that you don't need to include the character box size, font name, or default font height in the [Config]DefaultFont setting. Those are all default values in this case, and so don't need to be specified. You can just supply the test phrases and their constraint widths. But if you prefer, you can do the whole spec like this:

```
[Config]
DefaultFont=16 8 [MS Shell Dlg] 13 {TestPhraseOne} 74 {TestPhraseTwo} 75 {TestPhraseThree} 82
```

Next time you start APL, these settings will be in effect and will determine what font size is selected for the session as a function of system-wide DPI scaling.

deferexit

Description:

Scalar Boolean that specifies whether attempting to move the focus to this object will trigger an onExit event handler for the object which currently has the focus.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Frame, Label, List, Listview, Media, Option, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

The deferexit property is used to denote objects that can receive the focus without triggering an Exit event. It defers exit validation until you tab to another field. This is generally useful for Cancel buttons. It allows you to click a cancel button without running exit validation. When you tab or click to a control that has deferexit set to 1, the system remembers the control you WERE on and if you then tab on to another control, the validation is executed against the control that had focus before you put focus on the deferexit control.

The default value is 0. Typically you set this property to 1 for Cancel buttons to allow the user to exit a form without triggering an Exit event, even with invalid data. If you click or tab to a control that has deferexit set to 1, the system moves the focus to that control, and delays triggering the onExit event handler for the control that had the focus until you click or tab to a non-deferred control.

description (read-only)

Description:

Character vector that returns the plain text name that is associated with an ActiveX control or server.

Classes: ActiveControl and ActiveObject.

Remarks:

For ActiveX controls, this is the value in the first column of the system object's xclasses property; it also appears in the third column of the result of the XInfo method for ActiveX controls and servers.

dirty

Description:

Boolean flag that specifies whether a page was printed or drawn onto.

Class: Printer.

Remarks:

The Printer object refuses to print and eject pages that it detects has not been printed to or drawn on. You can set the dirty property to 1 to make a page appear dirty (printed to or drawn on). Setting the dirty property to 0 does not reset the dirty state of the printer page (because there is no way to do that and allowing such reset would put the Printer object into a false state). The default value is 0.

dock

Description:

Integer scalar or one-element vector showing the side of the container where the bar is currently docked or if it is floating. The possible values are: 0 = floating ; 1 = top; 2 = bottom; 3 = left side; 4 = right side; and -1 to designate the bar is a client.

Class: CommandBar.

Remarks:

The negative value means that you are treating this bar similar to a normal control, and that you have or are planning to specify its location explicitly by setting the where property. Such a bar does not have a gripper, does not dock, and is not controlled by the user. If you change the dock property to client mode from one of the other values, the system hides the bar, and you must explicitly Show it.

dockable

Description:

Integer value that tells where a user can move a command bar. You can set the dockable property to a combination of any of the following values:

- 0 User may not change the docking location of the bar
- 1 Bar may be docked along the top edge of the container
- 2 Bar may be docked along the bottom edge of the container
- 4 Bar may be docked along the left edge of the container
- 8 Bar may be docked along the right edge of the container
- 16 Bar may be floated to any position on the screen.

The default value is 31, unrestricted docking and floating. Although dockable always returns a scalar sum of the values, you can set it as a vector of any combination of the above values.

Class: CommandBar.

Remarks:

If you limit the user's options by setting the dockable property, the bar can still be docked at another location by setting the dock property. This property limits the location of the bar only through the user interface.

dockbreak

Description:

Boolean flag that says whether the bar starts on a new band (1) or follows another bar (0). You can set this property except that any setting is ignored for the first bar on any side of the container (the bar has an `order` property of 1) or for a `style 2` bar, which is always positioned in the upper right corner if docked at the top or the lower left if docked on a side.

Class: CommandBar.

Remarks:

You can use this property when creating multiple bars to determine their initial arrangements.

dockheader

Description:

Integer that describes the length of the area that holds the gripper and caption in the same direction as `docklength`. It is measured in the bar's scale units.

Class: CommandBar.

Remarks:

The default value is `-1`, which specifies the minimum length required to hold the entire caption.

dockideal

Description:

Integer that describes the length of a command bar when it is expanded from zero but does not fill an entire band. It is measured in the bar's scale units. This length does not include the area that contains the gripper and caption.

Class: CommandBar.

Remarks:

The default value is `-1`, which specifies the minimum length required to show all the children of the bar. If you set it to another value, the bar expands to the specified length, for example, when the user double-clicks on it when its length is zero (only the caption visible).

This property retains its value when the bar is floated, but it resets to its default if the orientation of the bar changes from horizontal to vertical or reverse (that is, the `dockvert` property changes).

docklength

Description:

Integer that describes the length of a command bar when it is docked top or bottom or the height when

docked on a side. It does not include the area that contains the gripper and caption. It is measured in the bar's scale units, or you can specify one of the special values:

- 1 = set to ideal size
- 2 = set to maximum size
- 3 = set to minimum size.

Class: CommandBar.

Remarks:

This property does not affect the size of the bar when it is floating. (It retains the length it had most recently when docked if you have not specified one of the special values.) When a bar is moved to a new location, the system resets `docklength`. This property is useful primarily for saving the state of a bar, so you can restore it when the application is restarted. You should set the `docklength` and `dockbreak` properties for all bars before any bars are opened. Setting `docklength` to an explicit value after a bar is open can give unexpected results.

dockmargin (read-only)

Description:

Four-element vector that describes the lengths on the four sides of a command bar that are included in the total screen space required to hold the bar but are not included in the `normsize` values. The values are measured in the bar's scale units.

Class: CommandBar.

Remarks:

The margins include the area of the dockheader plus the size of the pixels required for the edges.

dockmax

Description:

Integer that describes the maximum length of a command bar when docked top or bottom or the height when docked on a side. It does not include the area that contains the gripper and caption. The default value is -1, which means unrestricted. If you set it to another value, it represents a length in the bar's scale units beyond which the bar does not expand unless it is the only bar on a band, in which case it fills the entire width or height of the container.

Class: CommandBar.

Remarks:

This property retains its value when the bar is floated, but it resets to its default if `dockvert` changes.

dockmin

Description:

Integer that describes the minimum length of a command bar when docked top or bottom or the height when docked on a side. It does not include the area that contains the gripper and caption. The default value

is -1 , which means the bar can collapse to zero (leaving only the caption visible). If you set it to another value, it defines a length in the bar's scale units shorter than which the bar does not contract.

Class: CommandBar.

Remarks:

This property retains its value when the bar is floated, but it resets to its default if dockvert changes.

dockmindepth

Description:

Integer that describes the minimum size of a command bar perpendicular to its length when docked.

Class: CommandBar.

Remarks:

The default value is -1 , which allows the bar to collapse to the minimum size of the largest child in the appropriate dimension. If you set it to another value, it represents a minimum depth measured in the bar's scale units. If you set this property smaller than the default, the system ignores the setting.

dockside

Description:

Integer scalar or one-element vector showing the side of the container where a floating command bar will dock, if the user double-clicks on the label. If the bar is docked, dockside has the same value as dock, and you cannot set the value to a different side; trying to do so gives a `NONCE ERROR`. If the bar is floating, you can set it to: 1 = top; 2 = bottom; 3 = left side; or 4 = right side.

Class: CommandBar.

Remarks:

If you do not change this property, it retains the value for the side where the bar was most recently docked. If you create a floating bar, it defaults to 1.

dockvert (read-only)

Description:

Boolean flag that specifies whether the docked orientation of a command bar is vertical (1) or horizontal (0).

Class: CommandBar.

dpi (read-only)

Description:

Returns a nested four-element vector that specifies the DPI-Aware state of APL. The return value are:

[1] the current DPI aware mode (1=enabled; 0=disabled)

[2] the system-wide DPI scaling percentage of the system (100 for 100% scaling (default), 200 for 200% scaling, etc.)

[3] the dots per inch of the system-wide DPI scaling (96 at 100% scaling, 192 at 200% scaling, etc.)

[4] the number of pixels high the default system font is drawn at.

Class: # (the system object).

Remarks:

APL enables system-wide DPI scaling when started with a manifest file (e.g., aplw.exe.manifest) in the same folder specifying the entry colored in red below:

```
...
<asmv3:application xmlns:asmv3="urn:schemas-microsoft-com:asm.v3">
  <asmv3:windowsSettings xmlns="http://schemas.microsoft.com/SMI/2005/WindowsSettings">
    <dpiAware>true</dpiAware>
  </asmv3:windowsSettings>
</asmv3:application>
...
```

When **true** is replaced with false, APL allows the operating system to do the image stretching for Windows that are drawn as if running at 100% DPI. This is the same behavior as when starting APL without a manifest file present in the same folder.

The DPI scaling percentage of the system is the represented by the slider setting for the "Change the size of text, apps, and other items" selection for the primary monitor in the Windows Display settings window.

dragimage

Description:

Flag that specifies an image during a drag-and-drop operation. When a Listview or Tree object is the source of an item being dragged and when `EventArgs` from the `onDragFeedback` event handler contains (or would contain) the value 'defaultcursor' (which is its default), if `dragimage` is set to 1, the image of the item being dragged appears when the cursor is over a target object. By default, this property is empty.

Classes: Listview and Tree.

Remarks:

This property must be set on the source of the drag operation. For a Tree, when you have multiple images for a node, the image that appears as the dragged object is the image associated with the node when it is not selected (the first image index). The image may appear over an object that has a non-empty `targetformats` property, even if it is not a potential target for the item being dragged; in this case, the image is accompanied by the slashed circle ("not allowed") icon.

dragsource (read-only)

Description:

Nested vector that specifies the name of the APL64 WI object from which an item is being dragged, the identification of the item or node being dragged, and the mouse state.

Class: # (the system object).

Remarks:

If the drag operation originated in a non-APL application or a different APL application, or if nothing is currently being dragged, the property is an empty vector. If an item is being dragged from an APL64 control

in this application, `dragsource` contains three elements. The first element is a character vector containing the full name of the object where the drag originated. The second element is the identifier of the item being dragged. For a `Tree`, this identifies one node; for a `Listview`, this includes the item where the mouse press occurred. The third element identifies the mouse button that was pressed when the drag started: 1 = left button; 2 = right button.

dropcolor

Description:

The `dropcolor` property for a `DateTime` control has the same form as the `color` property (which is different for a `DateTime` control than for all other controls with a `color` property). It applies to the drop-down calendar for those styles that have a drop-down calendar.

Class: `DateTime`.

Remarks:

The `dropcolor` property inherits values from the `color` property of this control rather than from the parent of this control.

dropfont

Description:

The `dropfont` property for a `DateTime` control has the same format as the `font` property. It specifies the font used by a drop-down calendar for those styles that have a drop-down calendar.

Class: `DateTime`.

Remarks:

The `dropfont` property inherits values from the `font` property of this control rather than from the parent.

droptarget (read-only)

Description:

Character vector that specifies the name of the APL64 WI object on which an item that the user is dragging will be dropped if the user releases the mouse.

Class: `#` (the system object).

Remarks:

If the item being dragged is over a non-APL window or a different APL application, or if nothing is currently being dragged, the property is an empty vector.

edge

Description:

Integer value that can be zero, 1 or 2. If you set `edge` to zero, there is no edge on your control. This is the default value for most controls. If you set `edge` to 1, Windows draws the object with a client edge, which

gives a deeply sunken appearance. This is the default value for edit boxes. If you set `edge` to 2, Windows draws the object with a static edge, which gives a less deeply sunken appearance than the client edge.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

Many controls have inside edge styles comparable to the outside edge styles you set with the `border` property. You can set the `edge` property to provide this appearance. The effects vary depending on the control and the interaction with the setting of `border`.

empty (read-only)

Description:

Returns a variant type 0 (zero) when referenced.

Class: # (the system object).

Remarks:

You can use this to pass data to an ActiveX control or server. In some applications, this signifies the absence or non-existence of a field. Note that this terminology differs from the traditional APL concept where an empty vector exists, but has no values. Compare this with the null property.

enabled

Description:

Boolean flag that specifies whether an object is enabled or disabled.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

A control that is enabled responds to mouse and keyboard user input. A disabled control uses the disabled color as a visual cue to the user that it cannot be used. A disabled control is said to be greyed out or dimmed. If you disable a control, a frame for example, that contains other controls as children objects, those objects are also disabled. The system does not reset their individual enabled property values however.

The default value for all objects is 1, which means enabled. For the system object, this property is read-only. On the system object, this flag specifies whether callbacks are enabled.

error

Description:

Integer value containing the error code reported by the Printer object when an error occurs.

Classes: Printer.

Remarks:

If an error occurs during or after the print operation, the `error` property will have a non-zero value. Reset the error state to zero by setting the error property to 0.

errorCode (read-only)

Description:

Integer value containing the error code reported by the ActiveX object when an error occurs.

Classes: ActiveControl and ActiveObject.

Remarks:

For ActiveX controls, the system sets this read-only property when an error occurs if the creator of the ActiveX object has defined error codes. This is reset to 0 when the next successful call is made to that instance of the object.

errorLogEvent

Description:

Character vector that specifies the default Windows Event Log type of event used to log error events.

Class: # (the system object).

Remarks:

The default specification is `E 3 3000` where `E` is Event Error, `3` is the category, and `3000` is the event id used to log error events. This property uses the same set of codes as the `□LOG` function. See the description of the `□LOG` function in the *System Functions Manual* for more information. .

Note: This property has a corresponding INI configuration file parameter named `[Config]ErrorLogEvent` that defines the default values at startup time.

errorLogFilter

Description:

Character vector that specifies a list of errors for which you want to have a error crash log generated whenever they occur.

Class: # (the system object).

Remarks:

The default specification is empty. This property uses the same pattern as the `:catch` statement, such as:
`INDEX ERROR;DOMAIN ERROR..`

Note: This property has a corresponding INI configuration file parameter named [Config]ErrorLogFilter that defines the default values at startup time.

errorLogPath

Description:

Character vector that specifies the location of the error log files.

Class: # (the system object).

Remarks:

This may be a fully qualified name or a relative to the ApplicationLogPath. The default value of this property is: .\Log\Error\Error-*.log. If it begins with a dot (.) it is relative to [Config]ApplicationLogPath parameter.

The name must include one star (*) character. The system substitutes a timestamp in place of the star when generating the system crash log file. .

Note: This property has a corresponding INI configuration file parameter named [Config]ErrorLogPath that defines the default values at startup time.

errormessage (read-only)

Description:

Character vector that specifies the error message reported by the ActiveX object when an error occurs.

Classes: ActiveControl and ActiveObject.

Remarks:

For ActiveX controls, the system sets this read-only property when an error occurs if the creator of the ActiveX object has defined error messages. This is reset to ' ' when the next successful call is made to that instance of the object.

One reason for errormessage is that `MsgBox` can only be 255 characters max but the error message from an ActiveX control does not have a limit. So errormessage can capture the full error text, which is important for debugging in some cases.

eventLogName

Description:

Character vector that specifies the name of a registered event source.

Class: # (the system object).

Remarks:

If this property is not specified, the event log name defaults to the setting of [Config]ApplicationName or the system object's `caption` property. If none of those are defined, it defaults to "APL64" for the development system or the name of the executable module (such as APLWR unless you rename the exe) for the runtime system. APL64 does not register event sources, although it has an event message file built into itself to provide event category and event numbers for the standard events is signals in the Windows Event Log.

[http://msdn.microsoft.com/en-us/library/aa363661\(v=VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa363661(v=VS.85).aspx) explains how the Registry must be configured to register an event source. The event source built into APL64 can be registered via this technique by specifying its EXE name as the value of "EventMessageFile" and "CategoryMessageFile" parameters.

NOTE: A message can be logged to the Windows Event Log without any message source being registered. However, Windows will then complain when it displays the message in the event viewer because it cannot find the registered event source associated with the name that the event was logged with. However, the EventLogName specified here, will show up in the Windows Event Log (in the Application log) as written by this parameter. If you specify "APL64" that will appear in the event log. If you specify "My Cool App" that will appear in the event log. So registering an event source is only of secondary (and cosmetic) importance. You can still see the event message even if the event source is not registered.

events (read-only)

Description:

Nested vector of character vectors that lists events for which you can write handlers for this object.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

This property provides a read-only list of the events that can occur for an object of this class; you can reference it directly. The event-handler properties have the same names with an on prefix; for example, the Unfocus event triggers the onUnfocus event handler. You can determine which events have handlers defined by referencing the state property of an object.

extent

Description:

Two-element vector that defines the height and width of the object's client area in the scaling units of the object's parent. This property is read-only for Page and Status objects.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

The client area is the area inside the object's border, if it has a border. The border of a form includes the title bar and menu line. If there is no border, the client area is the same as the size of the object. If you add items to a control, such as a scroll bar in an edit control or a status bar or command bar on a form, the client area is reduced. The client area changes as the user moves command bars around.

The values in this property are measured in the scaling units of its parent. For example, you might place a control on which you plan to display a bitmap on a form and size the control so that it matches the size of the bitmap in pixels. If the rest of the controls on the form are placed according to character sizes, you also need to know the size of the control in character units, which are the scaling units of the containing object. The extent property gives you that size.

The extent property is related to the size and where properties. See the description of the where property for the relationships among extent, size, and where. For a definition of scaling units, see the scale property.

file

Description:

Character vector that contains the name of the file or device to be or being played.

Class: Media.

Remarks:

If you are playing animated cursor files or silent .AVI files with the Windows Animation control, there is no device. You need to specify only the name of the file you want to play. If you are using an MCI control, the player is a device. However, in many cases when you are using an MCI control, you can also specify just the file name. The system selects the appropriate device, if one is available, and plays the file. For example, you may specify: 'e:\bells.wav'

If you specify a file but do not specify the mediatype (for the child control), or if you specify 'auto', the value in the file property determines the mediatype. Note that the Animation control cannot necessarily open all .AVI files. If you specify an .AVI file, and the animation control can open it, the system will use the Animation control. Otherwise, it will use the Multimedia player.

Some devices do not use files as their storage medium. If you want to use a device that plays something other than a file, you must specify the device name and use the available commands or buttons to choose the item to play. In general, these devices include audio CD players, videodisc players, videotape recorders or players, and digital audio tape players.

In many cases, the name of the device is the generic device type; for example, you may be able to specify 'cdaudio' as the device name. If you have more than one similar device attached to your computer, you may need to end the designation with a number, for example: 'cdaudio2'. If the device has an explicit name, you may have to specify the exact name; for example, 'ReelMagic'. You can enumerate the list of devices attached to your computer by using the mcidevices function from the Windows workspace.

If you are playing the Multimedia player with a file, you can choose a non-default device by specifying both the device name and the file name separated by an exclamation point, for example:

```
'fmHolder.medPlayer' □wi 'file' 'MPEGVideo!test.mpg'
```

firstday

Description:

Singleton integer or two-element integer-valued vector identifying which day of the week is displayed in the first column of the calendar for either the visible or drop-down calendar styles. Values 1 through 7 represent Monday through Sunday respectively; the value zero means the locale-specific default.

Class: DateTime.

Remarks:

When you specify firstday, you specify only the first element. If you specify two elements, the system ignores the second. When referencing firstday, if the property has not been set, or if it has been set to zero, firstday is a two-element vector, with zero as the first element and the current locale-specific first day as the second element.

If the property has been set to a nonzero value, it returns a scalar with that value. When the drop-down calendar is not dropped down, and the value is not set to a non-default value, this will be a scalar zero.

fixedtabsize

Description:

Two-element vector describing the height and width of each tab for a Selector object, in its currently selected scaling units.

Class: Selector.

Remarks:

If FixedWidth style (32) is not selected, only the height is used. If you specify an empty numeric vector, Windows determines the size.

floatlayout

Description:

Scalar integer showing how a bar should be sized when it is made to float. The possible values are:

- 0 No automatic wrapping (use wraps specified on command buttons only).
- 1 Horizontal layout mode; use one row.
- 2 Vertical layout mode; use one column.
- 3 Constrain the layout width by the value specified in the second element of floatsize.
- 4 Constrain the layout height by the value specified in the first element of floatsize.

Class: CommandBar.

floatsize

Description:

Two-element vector showing size (height, width) of the inside edge of a floating bar in command bar units. These values exclude the floating border and title bar.

Class: CommandBar.

Remarks:

If the command bar has never been floated and this property has not been set, it returns an empty vector. If the bar is closed or docked, it shows the previous settings. If you set this property, the system treats the values as constraints. You can constrain either dimension, height or width; the system adjusts the other to make all children of the bar visible. If you assign positive values to both dimensions, the system treats one as a priority. You can set the following values to specify how the system arranges the command bar when it floats.

Settings for floatsize (or floatwhere)

Height	Width	Result
0	0	fixed layout mode; system creates new rows after command button wrap
-1	0	vertical layout mode; system places one child on each row
0	-1	horizontal layout mode; system places all children on one row
<i>m</i>	0	constrain vertical dimension; system limits height to <i>m</i> units
0	<i>n</i>	constrain horizontal dimension; system limits width to <i>n</i> units
<i>m</i>	<i>n</i>	constrain priority dimension; use floatlayout or previous mode

If the system recently performed a layout, for example, because you set `floatsize`, it continues to use the same dimension as priority. If you set the `floatlayout` property, the system uses that setting to determine the layout mode; however, it will use the dimension-constraining mode rather than the one row or one column mode if `floatsize` is specified.

floatwhere

Description:

Four-element vector showing the location (row, column) relative to the screen origin, not the container of the bar, and size (height, width) of the outside edge of a floating bar in screen units (characters). If the bar is closed or docked, but the position has been defined, this returns a two-element vector showing location. If the bar has never been floated and this property has not been set, it returns an empty vector.

Class: CommandBar.

Remarks:

The third and fourth elements of `floatwhere` change along with the two elements of the `floatsize` property, although they may be in different units. They differ slightly because `floatwhere` defines the outside edge of the floating bar, while `floatsize` defines the inside edge. If you set `floatwhere` to an empty vector, it returns to an undefined state. If you set it to a two-element vector, it defines a new position.

Setting it to a four-element vector defines both position and size, except that the size dimensions are treated as constraints and not fixed values. See the description of `floatsize` above for settings you can assign. You can also set these constraints and special values in `floatsize`. It may be better to do so, because the settings are retained in `floatsize`, but not in `floatwhere`, when the bar is docked.

focus

Description:

Character vector that specifies the name of the object that currently has the focus.

Class: # (the system object).

Remarks:

You can query this property to determine which object currently has the focus, and you can set this under program control to change the focus. If no APL64 WI object has the focus, the result is an empty vector. The object that has the focus receives the next keyboard input action. There is usually a visual indication on the object that has the focus. For a form, the system highlights the title bar; some objects have their captions outlined with a dotted line. You can also use the `Focus` method to set focus.

font

Description:

Nested vector that specifies the typeface, size, and style of the font that you want the system to use for text associated with the object whose property is `font`. You can specify `font` as a four-element vector whose elements are described below, or you can use the standard 14-element Windows `LOGFONT` structure. You can generate this structure by calling the `WChooseFont` function in the `WINDOWS` workspace with an argument of `' '`. The system displays a window where you can choose a font and its attributes. The result of

this function is a two-element nested vector, the first element of which is the 14-element LOGFONT structure. The elements are documented in comments in the WChooseFont function.

If you want to place text on an object at an angle other than horizontally left to right, use the Draw method Font and Text operations. The Font operation uses the four values described below plus an angle designation as its arguments.

Note that the RichEdit control does not use the font property. See the description of the selfont property later in this chapter for specifying RichEdit fonts.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Printer, Progress, Selector, Status, Toolbox, Trackbar, and Tree.

Remarks:

The value for font consists of four elements explained below: Trailing elements that you do not want to change can be omitted. You can specify other elements you do not want to change as 0.

- *fontName* is the logical name of the font. If you specify 0, the system selects a font based on the *fontStyle* value. To get an APL font, you can use APL+WIN for *fontName* and set *fontCharSet* to 'Symbol'. Also, for Edit controls, you must select a style that includes 2048.
- *fontSize* is the logical size of the font (in the current scale units as defined by the scale property for this object). If you specify 0, the system chooses a font size that is approximately the size of the standard character cell (1 logical unit high and wide when scale=1 is in effect). A scalar or one-element vector specifies the font height. A two-element vector specifies the height and average width. Note that if you change the scale property for the object, the font size will change also; the system does not convert this value to maintain a given font size.

The system interprets *fontSize* differently for this property than it does for the selfont property of a RichEdit control. This is the size of the cell (including internal leading) that contains the character. For scalable fonts, TrueType for example, the *fontSize* element of selfont is the size of the character; which is therefore larger in appearance. If you want to specify the size of the character in font to match a RichEdit control, use a negative value of the same magnitude. The text in the two controls will be the same size. Note that they still may not line up exactly due to the leading of the control itself.

Some fonts designed for the screen or printer are not scalable; they may be defined in terms of numbers of pixels. When you specify a size for such a font, you get the largest font that does not exceed the size you specify. Thus, small changes in *fontSize* have no effect, but at some point there is a dramatic jump. These fonts may show no difference in appearance when specified with a negative value for size; this can be caused by incremental effects or that the font is constructed with no internal leading.

- *fontStyle* is a numeric scalar or one element vector that indicates how the font is to be styled. The table below shows the codes you can use for *fontStyle*; the value is the sum of one value from each section of the list: Format, Pitch, Family and Quality.

Values for font Property *fontStyle* Element

Format value	Description	Family value	Description
0	Normal	0	Do not care
1	Bold	128	Roman
2	Italic	256	Swiss
4	Underline	384	Modern
8	Strikeout	512	Script
		640	Decorative
Pitch value	Description	Quality value	Description
0	Default pitch	0	Default quality

Format value	Description	Family value	Description
32	Fixed pitch	1024	Draft quality
64	Variable pitch	2048	Proof quality

- *fontCharset* is the name of the character set (selection and order of symbols). Choose one of the following or an integer: 'ansi' 'oem' 'symbol' or 'default'

Ansi is the default. OEM character sets correspond to the original IBM PC's built-in text character set. Symbol character sets contain special characters used, for example, to represent formulas. The default character set means use a font with the given face name regardless of its character set. Microsoft recommends using default character set sparingly, as it prevents the font mapper from finding a similar font if the requested face name does not exist on the system; this is especially important if your application is to be distributed to PCs whose exact configuration is not known.

If you specify an integer for *fontCharset*, the system treats it as one of the values used in the *lfCharSet* member of the *LOGFONT* structure used with low-level programming. This allows you to specify a character set other than the four most commonly used that are specified above. For example, 177 is the Hebrew character set and 204 is the Russian character set.

Example:

To put a label in Arial font, 1 character scaling unit high, and bold, with the *ansi* character set (you can omit the fourth value):

```
'fmSample.lLabel' □wi 'font' 'Arial' 1 1 'ansi'
```

You could also choose Arial font from the window called by the following statement. The left argument of 1 limits your choices to screen fonts. You must have the *WINDOWS* workspace loaded or copy *WChooseFont* and the six functions it uses (*C2I*, *D2C*, *I2C*, *W2C*, *W_Mem*, and *WhwndOwner*) into your workspace.

```
'fmSample.lLabel' □wi 'font' (1>(1 WChooseFont'))
```

format

Description:

Character vector that specifies how the contents of a *DateTime* edit box are displayed for styles other than the visible calendar. You specify the fields using the codes listed below. They appear in the control in the order you specify them with the formatting characteristics associated with the codes.

literal strings

You can place literal strings in the format property by enclosing the string with quotes. Since the property is a character vector, you must use double quotes and single quotes or double your quotes. You can insert punctuation marks and spaces without using quotes; however, Microsoft warns that this behavior may give unpredictable results, except that a comma need not be quoted.

date and time fields

Day of the week	Enter ddd for the three-character weekday abbreviation, or dddd for the full weekday name.
Date	Enter d for the one- or two-digit day or dd for the two-digit, zero-filled day.
Month	Enter M for the one- or two-digit month number, MM for the two-digit, zero-filled month number, MMM for the three-character month abbreviation, or MMMM for the full month name.
Year	Enter yy for the last two digits, zero-filled, of the year,

	or yyyy for the full four-digit year.
Hour	Enter h for the one- or two-digit hour in the 12 hour format, hh for the two-digit, zero-filled hour in the 12 hour format, H for the one- or two-digit hour in the 24 hour format, or HH for the two-digit, zero-filled hour in the 24 hour format.
Minute	Enter m for the one- or two-digit minute, or mm for the two-digit, zero-filled minute.
Second	Enter s for the one- or two-digit second or ss for the two-digit, zero-filled second.
Morning/Afternoon designator	Enter t for the one-letter abbreviation (A or P) or tt for the two-letter abbreviation (AM or PM).

callback fields

You can specify one or more callback fields by using uppercase **Xs**. You must use a different number of **Xs** for each field, but that number does not modify the appearance or format of the field; it merely designates which callback field it is.

Class: DateTime.

Remarks:

You control callback fields using the `onFormatQuery`, `onFormatString`, and `onUserKeyDown` event handlers. You specify the size of the field in the `onFormatQuery` event handler and the contents of the field in the `onFormatString` event handler. If you want the user to be able to modify the field directly using the keyboard, you use the `onUserKeyDown` event handler.

hdc

Description:

Handle of a device context; used for low-level programming.

Classes: Picture and Printer.

Remarks:

When the system creates a `Printer` object, it creates a Windows device context. The device context is a data structure that defines the attributes of the printer. If you are familiar with the Windows API, you can gain explicit control over the printer by referring to this `hdc` property to obtain a handle of the device context. You can then use the handle as an argument to one of the Windows API routines for manipulating the printer. You can also assign zero to the `hdc` property if the printer is closed. This allows you to free the Windows resources associated with the `Printer` object without deleting it.

When the system creates a `Picture` object, it also creates a Windows device context. This property is the handle to a memory device context compatible with the screen that Windows uses for drawing to the `Picture`. It is distinct from the `hwnd` property that is the handle to the "window" on the display screen for the `Picture` object. The `hdc` property enables direct low-level drawing into a `Picture`. It can be especially useful when using third-party graphics libraries. For a `Picture`, this property is read-only.

help

Description:

Arbitrary data associated with an object; this value appears as the third element of `EventArgs` during the `onHelp` event handler when this object is the target control.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

You can use this property to provide context sensitive help when the `onHelp` event handler for this control or a control in its parentage chain fires with this control as the target. See the description of the `onHelp` event handler for more information.

helpcontext

Description:

Arbitrary integer associated with an object; this value appears as the second element of `EventArgs` during the `onHelp` event handler when this object is the target control.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

You can use this property to provide the topic identifier in a help file. When the `onHelp` event handler for this control or for a control in its parentage chain fires with this control as the target, and the handler invokes the Windows help facility, the single integer in this property can provide the appropriate index. See the description of the `onHelp` event handler for more information.

highlightbold

Description:

Integer vector of identifiers of nodes in a `Tree` object that are displayed with a bold font.

Class: `Tree`.

highlightcut

Description:

Integer vector of identifiers of nodes in a `Tree` or items in a `Listview` object that are displayed as if cut.

Classes: `Listview` and `Tree`.

highlightdrop

Description:

Integer vector of identifiers of nodes in a `Tree` or items in a `Listview` object that are displayed as Windows displays targets for drag-and-drop operations.

Classes: `Listview` and `Tree`.

Remarks:

You can display one or more nodes or items as drop targets, but there is only one actual drop target. You can determine the drop target in a `Tree` object by using the `FindNode` method; you can set the drop target with the `ShowNode` method. The actual drop target is independent of the display state.

highlightfocus

Description:

Integer singleton identifying the item in a `Listview` object that currently has the focus or will be highlighted when the `Listview` gets the focus.

Class: `Listview`.

Remarks:

The appearance of items that are selected can be identical to or nearly indistinguishable from the one that has the focus, but only one item actually has the focus.

highlightselect

Description:

Integer vector of identifiers of nodes in a `Tree` object that are displayed as if selected.

Class: `Tree`.

Remarks:

You can display one or more nodes as selected, but only one node is actually selected. That node is identified by the `value` property, or you can query it using the `FindNode` method. You can select a node by setting the `value` property or using the `ShowNode` method; the user can also click on a node to select it.

himage (read-only)

Description:

Handle of the underlying Windows control for an `Imagelist`.

Class: `Imagelist`.

Remarks:

If you are familiar with the Windows API, you can gain explicit control over the imagelist by referring to this `himage` property to obtain a handle.

hwnd (read-only)

Description:

Handle of a window; used for low-level programming.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

When the system creates a displayable object, it creates a window handle that identifies the visual aspects of the object to Windows. If you are familiar with the Windows API, you can use this handle as a parameter to Windows API calls to gain explicit control over the window that represents the object.

When you use this property, you should explicitly reference it rather than storing the value in a variable and using the variable. The system may destroy and re-create the Windows window associated with an object, thus changing the hwnd value, without your knowledge. For example, setting the style property of an object has this effect. Similarly, changing a form may change the hwnd properties of not only the form but also all its children. Note that ActiveX objects do not have this APL64 property; however, they frequently have a property named hWnd, which provides the same information.

hwndddropdown (read-only)

Description:

Window handle of the drop-down calendar when it is dropped down; otherwise, zero.

Class: DateTime.

hwndmain (deprecated)

Description:

Window handle for the main APL+Win session in the development system; zero for a runtime system

Class: # (the system object).

Remarks:

The hwndmain for the System object is deprecated in APL64. To retrieve the main APL64 window handle, the recommended solution is the command `9>⎕sysinit`.

hwndsplash

Description:

Window handle of the window created for a custom splash screen.

Class: # (the system object).

Remarks:

You specify the file name of a bitmap you want to use as a custom splash screen as a startup parameter in the

`SplashBitmap` entry in the `[Config]` section of the `APLW.INI` file. This window is displayed early in the initialization process, giving the user feedback that the application is starting.

If this parameter is specified, the normal APL splash screen (for the development system) is not displayed. The custom splash screen remains displayed (for both development and runtime) until you explicitly remove it by setting the `hwndsplash` property of the system object (' # ') to 0.

This feature allows the resourceful programmer to subclass the window and create custom visual effects during initialization of the application. An example of this is the Microsoft Developer Studio's splash screen, which colors in logos on the screen as it finds components such as MS Visual C++ or the Developer Network Library. You cannot assign any value other than zero to this property.

icon

Description:

Character vector that specifies the image for the icon the system uses to represent the form; for example, on the taskbar.

Classes: Form and MDIForm.

Remarks:

You can specify an image for the icon that represents a form or an MDI form in several ways:

- You can use a character vector that names the file that holds the image. The file can have any of the extensions `.ICO`, `.CUR`, `.ANI`, `.BMP`, `.EXE`, or `.DLL`. (If you specify an animated cursor file, or either of the executable files, only the first image is used.)
- You can specify the name of an icon resource in a resource `.DLL` file that is specified in the `Resource DLL` entry in the `[Config]` section of `aplw.ini` or a name this is contained in `aplw.exe`. Such a designation might be: `'fm1' □wi 'icon' 'ICON1'`
- You can specify a resource in a `.DLL` or `.EXE` file other than the specified resource `DLL` by using a specification such as is allowed for `Imagelists`. The designation following the greater than sign can be `ICO` or `CUR`, and the identifier can be a name or a number. For example,
`'fm1' □wi 'icon' '>ICO 147 c:\windows\system\shell32.dll'`
`'fmPic' □wi 'icon' '>CUR ARROW c:\aplwin70\examples\rel70.dll'`
- You can specify an integer icon handle that you know from another source or that you get using the `'?Icon'` operation to the `Draw` method.

image

Description:

This property has two very different meanings. For a `CommandButton`, it is an integer that is the origin one index into the `Imagelist` objects you specify for the `CommandBar` that contains this `CommandButton`. The same index is used for all three `imagelists`: `imagelist`, `imagelistdisabled`, and `imagelisthot`. The `imagelist` used, if all are set, varies depending on whether the button is enabled and whether the cursor is over the button.

For a `Picture` control, this property is an array of encoded color values, each value corresponding to one bit, that represents the current image of the picture. These values originate from the `bitmap` property, but may be scaled, tiled, or modified.

Classes: `CommandButton` and `Picture`.

Remarks:

For the array, the color values are scalar values calculated using the base value primitive function:
256 $\perp$ *blue green red* (reversed order from the vector representation of red-green-blue).

If you modify `image`, for example using the `Draw` method, the changes persist until the control is closed. When the control is closed, the values of `image` revert to the definition of the `bitmap` property as defined by the `style` and `imagesize` properties.

imagealloc

Description:

Two-element numeric vector that gives the number of spaces for images in an `Imagelist` object. The first element is the number of images for which you want to have memory allocated; the second is the incremental number of images for which to allocate memory if you add an image when there is no room for it. The default values are 1 5.

Class: `Imagelist`.

Remarks:

If you specify `imagealloc` after the `imagelist` is populated, the system removes the existing images and reinitializes the `imagelist`. The system then looks at the `imagenames` property to populate the list.

imagecount (read-only)

Description:

Integer that gives the number of images in an `Imagelist` object.

Class: `Imagelist`.

Remarks:

If the object is closed, this is an empty vector.

imageindex

Description:

Integer singleton that is the origin one index in the `Imagelist` of the image you want to appear on a `Menu`, `Button`, or the tab of a `Page` object. For a `Menu`, this property can also be a two-element vector, specifying different images for the menu when it is unchecked and checked. You name the `imagelist` in the `imagelist` property of the parent `Selector` for a `Page`.

Classes: `Button`, `Menu`, and `Page`.

Remarks:

If you do not define the `imagelist` property, specifying `imageindex` has no meaning or effect. If you do have an `imagelist` defined, specifying zero means no image.

For a `Menu`, the first value is the index for the image when the menu is unchecked; the second value is the index for the image when the menu is checked. If you specify only a single value, that image appears both when the menu is checked and when it is unchecked. Specifying zero results in no image even when the `imagelist` property is set. Thus, if you want to have an image instead of the check mark, you can specify the `imageindex` property as (0 1).

imagelist

Description:

Character vector containing the name of the Imagelist object from which to draw images you want to display.

Classes: Button, Combo, CommandBar, Listview, Menu, Selector, Status, and Tree.

Remarks:

You can specify this name as a fully-qualified name; as a name beginning with a period, which is relative to the object containing the `imagelist` property; or as a name beginning with a colon, which is relative to the top-level object containing the parent of the `imagelist`.

The `imagelist` you name in the `imagelist` property of a `Selector` contains the images that display on the `Page` tabs. You specify the index to the image you want to display on a tab in the `imageindex` property of the `Page`.

The `imagelist` you name in the `imagelist` property of a `CommandBar` contains the images that display on a child `CommandButton` when the button is enabled but the cursor is not over the button. These images also interact with the non-default settings of the `value` property for each command button. You specify the index to the image you want to display on each command button in its `image` property.

The same index is used for all three `imagelists`: `imagelist`, `imagelistdisabled`, and `imagelisthot`. The size of your images (it gives the best appearance if the images from all three `imagelists` are the same height) determines the height of the button and hence the size of the command bar, unless you set the size specifically or add controls that make it larger.

The `imagelist` you name in the `imagelist` property of a `Listview` contains the images that display in `smallimage` view, `list` view, or `report` view. You use the origin one index to the images in this object to connect an image with a `listview` item. The same index is used for the images in `largeimage` view, from the `imagelist` named in the `imagelistlarge` property.

In a `Tree`, `Combo`, `Menu`, or `Status` object, the images display to the left of the label, list item, caption, or text. You use the origin one index to the images in this object to connect an image with a node, list item, menu item, or pane, respectively.

For a `Button`, the `imagelist` property is a character vector containing the name of the `Imagelist` object from which to draw the image to place on the face of the button.

imagelistdisabled

Description:

Character vector containing the name of the `Imagelist` object that contains the images you want to display on command buttons when the buttons are disabled. These images also interact with the non-default settings of the `value` property for each command button.

Class: `CommandBar`.

Remarks:

You specify the index to the image you want to display on each button in its `image` property. The same index is used for all three `imagelists`: `imagelist`, `imagelistdisabled`, and `imagelisthot`. If you set only the `imagelist` property for the `CommandBar`, only one image appears for each button; it is modified by system greying when it is disabled.

The size of your images (it gives the best appearance if the images from all three imagelists are the same height) determines the height of the button and hence the size of the command bar, unless you set the size specifically or add controls that make it larger.

imagelisthot

Description:

Character vector containing the name of the `Imagelist` object that contains the images you want to display on each command button when the user moves the cursor over the button using the mouse. These images also interact with the non-default settings of the `value` property for each command button.

Class: `CommandBar`.

Remarks:

You specify the index to the image you want to display on each button in its `image` property. The same index is used for all three imagelists: `imagelist`, `imagelistdisabled`, and `imagelisthot`. If you set only the `imagelist` property for the `CommandBar`, only one image appears for each button; it is modified by the appearance of raised edges when the cursor is over the button.

The size of your images (it gives the best appearance if the images from all three imagelists are the same height) determines the height of the button and hence the size of the command bar, unless you set the size specifically or add controls that make it larger.

imagelistlarge

Description:

Character vector containing the name of the `Imagelist` object that contains the images you want to display in largeimage view.

Class: `Listview`.

Remarks:

You use the origin one index to the images in this object to associate an image with a caption.

imagelistuser

Description:

Character vector containing the name of the `Imagelist` object that contains the user-state images you want to display in a listview or a tree to the left of the image from the `imagelist` property. In a `Listview` object, these images display only in list view or report view.

Warning: Due to a quirk in the Windows system, the first image in the user-state `imagelist` for a `Tree` is not accessible. Until this is fixed, you need to put a dummy image in the first position and start with the second. Also, due to a Windows limitation, you cannot index more than 15 images in an `imagelist` being used for user state images. In APL, using an index greater than 15 results in a `DOMAIN ERROR`.

Classes: `Listview` and `Tree`.

Remarks:

In a `Listview` object, if you switch or allow users to switch among view modes, you should reset `imagelistuser`

to an empty character vector when `largeimage` and `smallimage` views are selected. Particularly in `smallimage` view, the underlying control does not handle user state images well.

imagenames

Description:

Vector of character vectors specifying image sources for an `Imagelist` object. When you specify this property and the object is open, or when the object is opened and `imagenames` is not empty, the system removes any existing images and loads the images specified in this property into the imagelist.

Class: `Imagelist`.

Remarks:

The `imagenames` property remains intact and is used whenever necessary to populate the imagelist until the first time the `AddImages` method is used. Once you use `AddImages`, the `imagenames` property is set to an empty vector. When `imagenames` is empty, existing images remain and the system adds additional images at the end of the imagelist. You can clear the imagelist by resetting the `imagealloc` or `imagesize` properties.

There are a number of ways to specify the image sources in `imagenames`; the simplest is to use file names with the appropriate `.bmp`, `.ico`, `.cur`, or `.ani` suffix. See the technical note under the description for the `Imagelist` class for alternatives in specifying image sources.

imagescale

Description:

Numeric scalar specifying the dpi scaling unit of the image source file, DLL resource or bitmap handle.

Classes:

`Button`, `Check`, `Combo`, `CommandBar`, `CommandButton`, `DateTime`, `Edit`, `Form`, `Frame`, `Imagelist`, `Label`, `List`, `Listview`, `MDIForm`, `Media`, `Menu`, `Option`, `Page`, `Picture`, `Progress`, `RichEdit`, `Scroll`, `Selector`, `Spinner`, `Status`, `Toolbox`, `Trackbar`, and `Tree`.

Remarks:

A DPI issue involves scaling of images such as bitmaps, icons, and cursors used in the application. As with `Window` (Windows Interface) layout units and font scaling, this too happens automatically, with rare exceptions that are noted below. Icons and cursor are the easiest to describe. Windows automatically scales them to an appropriate size as a function of system-wide DPI scaling. So they are drawn at the default icon and cursor size that Windows chooses for these UI (User Interface) elements as a function of DPI setting. However, bitmaps are more complex. In general, any bitmaps that you reference by name or DLL image resource notation are automatically scaled appropriately. If a bitmap used in a toolbar imagelist is designed as 10 pixels high and 20 pixels wide per image, and has 5 images, its total dimension will be 10 pixels high by 100 pixels wide, in the bitmap file or DLL image resource. But when this bitmap is used to in an APL application that is running at 200% DPI, it will automatically be scaled such that it is 20 pixels high by 200 pixels wide, and the width of each image will be 40 rather than 20 pixels wide. You would normally use such a bitmap in an `Imagelist` object and specify its size with the `imagesize` property, to tell the `Imagelist` how wide each image is. The `imagesize` returns pixel size of the image as corresponding to the bitmap as it existed in the bitmap file or DLL image resource. So while `imagesize` continues to be 10 20 for this particular example, internally the image has been stretched to 20 40 in size. If you think of the pixel units used in the `imagesize` property as being in scale 5 units, you'll be correct in thinking of them as being virtualized internally to the correct number of DPI scaled real pixels required to render the image to appear as it always has in relation to other element sizes (that have correspondingly been scaled by the DPI setting by exactly the same amount).

In some rare cases you might work with bitmap handles that you obtain from other means, perhaps from some ActiveX object that creates and draws into a bitmap for you. In this case, you will often be working with a bitmap that was scaled as it should be displayed at 100% DPI. That's the default assumption APL makes for working with such bitmap handles. If you pass a bitmap handle to the `AddImages` method of an `Imagelist` object, APL will assume the bitmap was drawn for 100% DPI viewing and scale it up automatically to the actual DPI scale that the application is running at. The same thing happens if you use the `Draw` method with the `Bitmap` operation that uses a bitmap handle as an argument (rather than a bitmap file name or DLL resource specification). In that case, the bitmap will be scaled up to the actual DPI scale and draw onto the object you specified.

There may be times when automatic re-scaling of bitmap images is not what you want to happen. You can control this image scaling behavior by using the `imagescale` property. All Window based `□wi` objects have this property (such as `Form`, `Picture`, `Label`, etc) as well as the `Imagelist` object. The `imagescale` property specifies the assumed scaling of the image source file, DLL image resource, or bitmap handle that you are using to draw or load images from. The default is 100 which means that APL will assume that images are scaled for 100% DPI rendering. In this case, when running at a different DPI scale than 100%, the system will automatically re-scale the image for the DPI scale you are running at.

You can disable automatic image scaling by specifying `imagescale` as 0. In that case, APL will not re-scale any bitmaps it draw or loads for that particular object that specifies `imagescale = 0`. Use this setting to suppress automatic image scaling for a particular object. If you have an image that it actually drawn for DPI viewing scale other than 100%, you can specify that percentage, and then if you are running APL at a different scale, it will be automatically scaled up or down as needed to the DPI level you are running at.

There is an important exception for the `Picture` object's image property. When you reference this property it returns a matrix of color codes corresponding to each actual pixel in the picture. If you are working in `scalemode = 6` (true pixels) the shape of the image property and the value of the `imagesize` property will be the same. But for other scale units, the `imagesize` property returns the scaled units corresponding to the actual pixel size of the image drawn on the `Picture` object. In the past this was true for `scalemode = 5` (virtual pixels) and `imagesize`. But now that scale 5 units are virtualized, the shape of the image property's return value will not match the `imagesize` property value unless you are running at 100% DPI. The only time the scale size matches the shape of the value returned by referencing the image property is when you are running with scale 6 units. The `imagescale` setting does not have any effect on the value returned by the image property because the image property is always related to the actual image pixels on the screen.

imagesize

Description:

Two-element numeric vector that specifies the height and width of the image property for a `Picture` control or the height and width of each image you want an `Imagelist` to hold for display.

Classes: `Imagelist` and `Picture`.

Remarks:

For a `Picture` object, the elements of this property specify the height and width of the image in the scaling units defined for the control. Initially, this size matches the client area of the control, which makes the entire image visible. If you reset `imagesize`, the image itself is restored to the default value.

For an `Imagelist`, when you are using bitmaps, the height of each bitmap you add must match the first element of this property; if it does not, the system triggers the `onResource` event handler of the system object, which may occur when you specify the bitmap, when you open the `imagelist`, or when you reset the `imagecount` or `imagesize` properties. When you specify `imagesize`, the system removes all images that are

currently in the imagelist. The system reinitializes the object, and then checks the `imagenames` property. If that property is not empty, the system repopulates the imagelist with the images specified in `imagenames`.

If you do not set `imagesize`, the first image you add to an imagelist establishes the size. If you want to use vertical slices of one bitmap for multiple images, you must set `imagesize` before you add the first bitmap. When you add a bitmap to the imagelist, the system calculates the number of images it can take horizontally and sets the index values for those images. If you subsequently add another bitmap, its first index is one higher than the index for the last image of the previous bitmap. Using a bitmap whose width is not an even multiple of the horizontal value of `imagesize` is not recommended; if you do, the system may or may not try to use the remainder as the first part of the next image.

If you add an icon or a cursor to an imagelist when `imagesize` is not specified, the system uses its default icon or cursor size to specify `imagesize`. If the (possibly multi-sized) icon or cursor has an image of this size, the system uses it. If the icon or cursor is not the default size, the system takes an image near the desired size and stretches or shrinks it to fit. If you add an icon or a cursor to an imagelist after `imagesize` is specified, the system stretches or shrinks the image to fit.

imagespace

Description:

Integer that specifies the number of pixels between the image and the caption of a button on which you are placing an image. The default value is 3 pixels.

Class: Button.

Remarks:

This property is useful only when you set both an image and a text caption for the button. The value of this property defines the number of pixels (not scaling units) between the image and the caption.

increment

Description:

Two-element vector that specifies the page increment and line increment for a Trackbar object.

Class: Trackbar.

Remarks:

The page increment is the number of values the slider moves when a user presses the PageDown or PageUp key or when the user clicks on the trackbar channel. The line increment is the number of values the slider moves when a user presses an arrow key. If you specify `-1` for the page increment, the system sets it automatically; this is the default value. The default value for line increment is `1`.

indent

Description:

Numeric scalar giving the distance a child is indented from its parent in a Tree object, in the current scaling units for the object.

Class: Tree.

Remarks:

The property returns an empty vector if the Tree is closed and the indent property has not been explicitly specified. Setting this to an empty vector restores the default indent.

infotipmask

Description:

Integer that specifies which places in the Tree and Listview "info tips" are triggered. For the Listviw, you can set the infotipmask property to a combination of any of the following values:

- 2 On icon
- 4 On label
- 8 On state icon

For the Tree, you can set the infotipmask property to a combination of any of the following values:

- 2 On icon
- 4 On label
- 8 On indent
- 16 On button
- 32 To right of label
- 64 On state icon

Classes: Listview and Tree.

Remarks:

The default is 4, on label. Use -1 to select all possible places.

instance

Description:

Integer that represents the instance index number. The system sets this property if you create an object with a bracketed number at the end of its name; for example:

```
'fmMain.fFrame.opOption[2]'
```

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, and Tree.

Remarks:

The instance property distinguishes multiple objects that share a common name. This property exists primarily for system-conversion compatibility.

interface

Description:

Integer that represents a pointer to an ActiveX control or server. You can use the pointer for low-level calls to the object. This value is not "typed"; the corresponding value in the obj property is typed.

Classes: ActiveControl and ActiveObject.

Remarks:

This pointer is transitory; it may be valid only for the duration of a callback. Thus, you should reference and use it in the same function; do not store it in a global variable and try to use it during a different callback. APL+Win must keep track of these pointers in order to release the underlying Windows resources when appropriate. Thus, each pointer is invalidated when a callback returns.

interval

Description:

Integer that defines the spacing (in milliseconds) of Timer events by a Timer object.

Class: Timer.

Remarks:

The default value is 1000 (1 second). If you set this property to any positive integral value, a Timer event occurs at this frequency whenever the Timer object is open and enabled. If you set the interval to zero, the system produces Timer events as often as Windows can provide them without impeding other applications.

If you encounter a problem because you set a timer with a very short interval, unselect the Enable Callbacks option on the Debug menu in the APL window. This will stop the callbacks coming into APL, giving you the opportunity to make any desired programming changes.

keys

Description:

Numeric vector that represents the characters for which you want the onKeyPress event handler to apply.

Characters are represented by their index in the ANSI character set. See the Atomic Vector and ANSI Sequence appendix in the *System Functions Manual* for the index values.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Picture, RichEdit, Spinner, Toolbox, Trackbar, and Tree.

Remarks:

The `keys` property tells APL which characters of input trigger `KeyPress` events. If `keys = ⍬`, the system executes the `onKeyPress` event handler (if you define one) for any character the control recognizes. If `keys` includes one or more ANSI character codes, the system executes the `onKeyPress` event handler only for those characters. The control handles all the other input characters in the default manner.

An Edit object normally receives `KeyPress` events. If the `keys` property is empty, the `onKeyPress` handler recognizes all normal inputs that generate characters (letters, digits, symbols, and punctuation marks) but not inputs that take actions or change state, such as Shift.

If you want to handle only specific characters in an Edit control, set the `keys` property for those characters. The user can enter other characters, but the system does not execute the `onKeyPress` event handler. If you want to limit entry to specific characters, you must suppress all other characters in your handler.

If you want to monitor certain keyboard events for other object classes, you should use the `onKeyDown` or `onKeyUp` event handlers. If you want to take an action on an object with one specific key, you can define a hidden menu as a child of the object, specify the key as a shortcut key for the menu, and define the action in the `onClick` event handler for the menu.

labeledithwnd (read-only)

Description:

Window handle of the system-generated edit control being used to edit the label of an item of a Listview or the node of a Tree. Returns zero if no label is being edited.

Classes: Listview and Tree.

Remarks:

You can use this value to subclass the edit control for low-level programming. It is most likely to be useful in `onBeginLabelEdit` and `onEndLabelEdit` event-handler functions.

limit

Description:

Integer scalar or vector that defines behavior for user-input fields.

Classes: Combo, DateTime, Edit, List, and RichEdit.

Remarks:

This property has slightly different meanings for different classes:

- Edit, RichEdit, or Combo
Integer scalar or one-element vector that specifies a width limitation. Zero or `⍬` means no limit (except

as Windows itself imposes); a user can type many characters into an Edit control because the text scrolls left and right. A positive integer specifies the maximum number of characters that a user can enter.

There is a special circumstance when specifying the limit for a multiline Edit control that allows the character generated by the Enter key as a value in the text (style 8192+4). Windows counts the Enter as two characters, a new line and a linefeed, when it determines whether the limit has been reached.

Therefore, if you set limit to 10 and enter eight characters and an Enter, your text property is only 9 characters long, but Windows does not allow you to enter another character. (You can place the cursor at the end of the text and reference the selection property to see its length.)

- **DateTime**

Integer singleton that specifies the maximum number of days allowed in the calendar control with multiselect (style 4) selected. With this style, the default value is 7. This property defaults to 1 unless the primary style is zero and 4 is selected. It is meaningless with styles 1, 2, or 3.

- **List**

A scalar or a vector with 0, 1, or 2 elements that specifies column width for horizontal scrolling. This can be a two-element vector only for compatibility with previous code; the second element is ignored in APL+Win. If limit is empty, there is no horizontal scrolling. If limit has a value in its first element:

and List has multiple columns (style includes 32), limit specifies the width of columns. The system places as many items as fit vertically in each column and wraps the next item to another column.

and the List has a horizontal scroll bar without multiple columns (style includes 64 but not 32), limit specifies the width of the scrollable column.

limitwhere

Description:

Zero-, two- or four element integer vector that restricts the size of a Form or an MDIForm when it is resized.

Classes: Form and MDIForm.

Remarks:

If you specify a two-element vector, this property determines the minimum height and width of the form's where property any time it is resized, either by the user or by an event handler. The third and fourth elements, if you specify them, determine a maximum height and width even if you maximize the form. The default is an empty vector, which means the form size is restricted only by Windows limits.

links

Description:

Two-column matrix with the first column being a link name and the second column being the actual name of an APL object that is a target or another link to that name.

A link name is an arbitrary name (a link) that applies to another object (the target) anywhere in the same application. The name is a "link" from the pseudo-parent to the target, or an alias for the target. Your program can use the name of a link in all the ways you use an ordinary object name to identify the object that is the target of the link, but it is as though the target is a direct child of the object for which the link is defined. For example, if this property is for a form named fmDisney:

```
'fmDisney' ⌵wi 'links'
'Porky' 'fmComic.selCharacter.pgAnimal.bnPig'
```

Then the statement below invokes the onClick event handler for the button.

```
'fmDisney.Porky' ⌵wi 'Click'
That's all, folks!
```

The link name (first column) must be a single-part name, as for any child. The target name can be an absolute name or a relative name; if it is a relative name, the base name is the name of the object that defines the link.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

If you are assigning only one link name, you can use a two-element vector, which the system treats as a one-row matrix. You can set the links property directly, or you can use the `SetLinks` method, on any object, with a two-column matrix (or a two-element vector for a single link) as the argument. You can delete a link, using `SetLinks`, by specifying the link name with an empty target (second column or element), or by resetting the links property without the row of the link you want to delete.

list

Description:

The text of the list of choices from which the user makes a selection in a Combo or List control object; the names of the bitmaps or icons you want in a Toolbox; or an array describing the items in a Listview or the nodes in a Tree.

Classes: Combo, List, Listview, Toolbox, and Tree.

Remarks:

- For a Combo or a List, the property can be a nested vector of character vectors, a character vector with `\t`-delimited strings, or a character matrix of choices in the list. When you reference it, it is returned as you specified it. If empty, the system returns `0 0 0 0` regardless of how it was set.
For a Combo, the property may also be a five-column matrix or a nested vector of vectors with some or all of them having the same structure as a row of the matrix. The extra columns are meaningful only when the style property for the Combo includes 4096; they are ignored for other styles. This extended Combo control allows you to have images and indented items in the list.
 - The first column is the character string that appears in the list.
 - The second column is the the number of indent steps; each step moves the text by 10 pixels.
 - The third column is the origin 1 index to the image in the imagelist specified in the imagelist property that you want to display when the item is not selected; if you specify zero, no image appears.
 - The fourth column is the index in the same imagelist for the image you want to display when the item is the one selected. You may specify -1, the default, to have the same image as unselected.
 - The fifth column is the index into the overlays property of the same imagelist for an overlay image; you can specify zero for no overlay.
- For a Toolbox, list is a nested matrix, or a nested vector that the system treats as a one-row matrix, that specifies the names of the bitmaps or icons you want in the toolbox. Each item of this nested matrix specifies the bitmaps or icons that occupy one cell of the toolbox. If you specify one name for each cell position, this icon or bitmap will always be the one displayed. If you want to be able to toggle among several bitmaps or icons displayed in a cell, you can specify a nested vector of names. For example, when the user presses the mouse button over a cell, you may want your application to display an icon that appears to be pushed in, and an icon that appears to be highlighted or selected when the user releases the mouse button.

- For a Listview, list is a nested matrix containing one row for each item in the listview. You can have a variable number of columns; these columns can contain the actual information or they can contain placeholders that allow you to construct the information at the time the entry is about to be displayed using the onGetVirtualData event handler. A row of the list is organized as follows:

RowID Images Cell[1] Cell[2] ... Cell[n]

RowID

Unique positive integer item identifier. You use this number to identify the item. If you specify zero, the system assigns the next available ID in sequence.

Images

This column can contain one, two, or three numbers. If you specify one number, it is the index to the small or large image for this item. This number is used as an index into the Imagelist objects you specify in both the imagelist and imagelistlarge properties.

The second number is the index to an overlay image in the same Imagelist object as the first number. The indices to overlay images are specified in the overlays property of the imagelist. You specify the index to the list in the overlays property. You can have up to fifteen overlays in one imagelist. Thus, if your overlay images were the eighth, ninth, tenth, and eleventh images in the imagelist, the overlays property would be the vector 8 9 10 11. If you wanted to use the third overlay with the first image in the imagelist, you would specify this column of the appropriate row of the list property as 1 3.

The third number is an index to a user-state image in the Imagelist object you specify in the imagelistuser property.

You can specify zero for either the second or the third number; you cannot use zero for the first number. If you specify -1 for an entry, the system triggers the onGetVirtualData event handler when it needs information about the image. If you specify -2 for the image, the system inserts the default value when you are adding or inserting a row, and does not change the value if you are changing other values in an existing row. The defaults are 1 0 0.

Cells

You can have one or more columns of display information. Each cell contains a character vector. One of the columns contains the primary cells. The values contained in this column display as the labels in all the views. You can display any of the cells in report mode. You designate the column of primary cells in the column index in the first row, fourth column of the columndisplay property.

You can also assign an empty numeric vector (∅) to a cell to mark it as virtual; this triggers the onGetVirtualData event handler when the system needs information to display.

You can assign the list property directly, or you can build it using the AddRows or InsertRows methods to create new items, or the SetRows, SetCells or SetImages methods to change values in existing rows.

If you want to work with virtual data, you can specify an n by 0 matrix (the system creates a matrix of n virtual rows with RowIDs of 1: n) or you can specify an n by 1 matrix with the RowIDs you want to use. The images column defaults to 1 0 0, and all cells are virtual until you specify them.

- For a Tree, list is an array describing the nodes in the tree. The array has one row with five columns for each node. When inserting new nodes by setting the list property or by using the InsertNodes method, you must locate a node after its parent and before its parent's next sibling.

The columns contain information about the node and how it is displayed:

- 1 The first column is the node identifier. The node identifier is a positive integer less than 2×31 . When specifying the list property, you may specify the identifier as 0 and have APL+Win assign identifiers. If you assign identifiers, they must be unique.
- 2 The second column is the relative depth of the node. These numbers must be a dense set of integers where each node has a depth one greater than its parent. By default, the first level nodes are depth 1, and their immediate children are depth 2. You can assign an arbitrary depth to the first node you

create, and use the next higher integer for its children. You can have any number of levels, but you cannot skip levels.

- 3 The third column is the text label. The text label is a character vector to be displayed for the node or a negative numeric identifier. Although you can specify longer labels, the control uses only 259 characters.

You can specify -1 as the text label; the `Tree` control requests the label via the `GetVirtualData` event when it is needed to display or sort the node. You must program your `onGetVirtualData` event handler to provide the information. If you specify -2 for the label, the system inserts the default value when you are adding or inserting a row, and does not change the value if you are changing other values in the row.

- 4 The fourth column holds the image indices. This is a four-element numeric vector where the first element is the index to the image you want to display to the left of the text label when the node is not selected. The second element is the index to the image you want to display to the left of the text label when the user selects the node. The third element is the index to an overlay image, and the fourth element is the index to a user-state image. The first and second indices may not be zero.

The value for a selected or nonselected image is the origin one index to the image in the `Imagelist` object whose name you designate in the `imagelist` property of the `Tree`. The overlay value is the index of an overlay image from the same `Imagelist`; however, you must specifically designate an overlay image as such in the `overlays` property of the `Imagelist` object.

In the `list` property of the tree, you use the index value from 1 to 15 of the `overlays` property of the `imagelist`, rather than the direct index into the `imagelist`. See the description of the `Imagelist` class for a description of overlay images.

The user-state image index identifies an image from the `Imagelist` object whose name you designate in the `imagelistuser` property of the `Tree`. A user-state image is displayed to the left of the nonselected or selected image. An application can display user-state images, for example checked and cleared check boxes, to indicate application-defined states.

The default values for the image indices are 1 1 0 0. You can specify -1 for a selected or nonselected image; the `Tree` control requests the image via the `GetVirtualData` event when it is needed to display the node. You can also specify -2 to retain the existing image index when the item already exists. You cannot specify zero for either the first or second index.

- 5 The fifth column contains a nested vector of character vectors describing the states of the node. The valid states are:

`'highlightbold'`, `'highlightcut'`, `'highlightdrop'`, `'highlightselect'`, `'expanded'`, `'expandedonce'`, `'haschildren'`, and `'forceparent'`.

You can precede each word by a minus sign (`' - '`) to turn the state off. The special value `'?forceparent'` means request the `forceparent` state via the `GetVirtualData` event when needed.

The four `highlight` states are explained under the properties of the same names. The `expanded` state means the immediate children of the node are available for display. If the node is collapsed, the children are not visible even when the node is displayed.

If a node is a parent, it returns the `haschildren` state; you cannot set this state directly. If you attempt to set this state, the setting is ignored, but the system does not signal an error.

If you set the `forceparent` state, the system treats the node as if it has children; for example, the node has a button if you set the `style` property for buttons. Clicking on such a node also triggers the `onExpanding` event handler. You can use this state to postpone inserting children until they are actually needed for display. When you specify `?forceparent`, the system triggers the `onGetVirtualData` event handler when the node is about to be displayed. The system uses the result of `onGetVirtualData` to determine if there are children.

If the `expandedonce` state is set, the system does not fire `onExpanding` and `onExpanded` event handlers for the node. The system sets the `expandedonce` state each time the node is expanded or collapsed.

If you want the event handlers to fire, you must specify `-expandedonce` to turn the state off every time the `onExpanded` handler is triggered.

margin

Description:

Four-element numeric vector that specifies the left, right, top, and bottom margins of the printer in the current scaling units for the object.

Class: Printer.

Remarks:

If you set the values for any of the margins to less than their initial values or to a value greater than is allowed by the printer driver, the system resets the values to the maximums or minimums allowed.

Caution: Setting the right margin is ineffective under most circumstances.

maskcolor

Description:

Three-element numeric vector or numeric scalar that defines the color that is drawn, by default, as transparent for bitmaps you add to the imagelist.

Class: Imagelist.

Remarks:

This property does not apply to images that already exist in the imagelist when you specify `maskcolor`. This value also does not apply when you add a bitmap that has a mask associated with it or when you explicitly specify a mask color for a bitmap.

You can specify this value as a composite color using three values of 0 to 255 for each of Red, Green, and Blue, or you can use a scalar formed by combining the three numbers using the Base Value primitive function: `256 ⊥ blue, green, red`. (Note the reversed order of the colors.) The default value is `192 192 192`, which is Windows default button-face gray. You can also use negative integers to specify the colors defined for certain Windows elements by the user. These, of course, will vary from user to user. See the description of the color property for a list of these values and elements.

mediahwnd (read-only)

Description:

Handle of the child control window for a media control.

Class: Media.

Remarks:

When Windows creates a media control, it actually creates two objects; one is the window for the control itself, and the second is a separate window for the player. The media control has an `hwnd` property; the child

control window for the player has the `mediahwnd` property. You can use this value for low-level programming.

mediatype

Description:

Two-element nested vector that contains the requested child control type and the actual child control type.

Class: Media.

Remarks:

When you specify `mediatype`, you specify only the first element; if you specify both elements, the system ignores the second. The default value is `'auto'`. When you specify the default, the file determines the actual child control type. If you wish to specify the type explicitly, you can use `'ani'`, `'avi'`, or `'mci'`; you can use this, for example, to force a simple .AVI file to be played by a Multimedia player, even though an Animation control is capable of playing it.

methods (read-only)

Description:

Nested vector of character vectors that lists the methods you can use with this object.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

This property provides a read-only list of the methods you can use for an object of this class; you can reference it directly.

minsize (read-only)

Description:

Two-element numeric vector showing the minimum value for the `size` property that allows a single month to be displayed completely.

Class: DateTime.

Remarks:

You cannot reference this property while the object is closed. If the primary style is non-zero, and the calendar is not dropped down, this property returns an empty vector.

missing

Description:

Returns a variant type 10 when referenced; this is labeled as "error."

Class: # (the system object).

Remarks:

You can use this value, for example, as a placeholder for an optional argument when invoking functions on an ActiveX control.

mode

Description:

Three-element vector that defines how the system displays graphics or text on a control or form. The three elements are *TransparencyMode*, *RasterOpMode*, and *PolyFillMode*:

- **TransparencyMode**
Determines whether Text operations use a background fill between adjacent characters or let the background show through. The default value (1) is opaque; the text appears on a solid background color. If you set the first element of mode to 0 (transparent), the text background color assumes the color of the control object behind it.
- **RasterOpMode**
Sets the mode in which the Draw method performs graphics operations. These settings use the same names that Microsoft uses for Windows SetROP2 or for Visual Basic's DrawMode property:

Mode	Name	C and APL Notation	Description
1	Black	0 0 0 0	Draws black pixels.
2	Not Merge Pen	~(Pen Screen) ~Pen∨Screen	Draws a combination of colors that are in either the inverse of the pen color or the inverse of the screen color.
3	Mask Not Pen	(~Pen)&Screen (~Pen)∧Screen	Draws a combination of colors that are in both the screen color and the inverse of the pen color.
4	Not Pen	~Pen ~Pen	Draws the inverse of the pen color.
5	Mask Pen Not	(~Screen)&Pen Pen∧~Screen	Draws a combination of colors that are in both the pen color and the inverse of the screen color.
6	Not	~Screen ~Screen	Draws the inverse of the screen color.
7	XOR Pen	(Pen∧Screen) Pen≠Screen	Draws a combination of colors that are either in the pen color or in the screen color, but not in both.
8	Not Mask Pen	~(Pen&Screen) ~Pen∧Screen	Draws a combination of colors that are in neither the pen color nor the screen color.
9	Mask Pen	(Pen&Screen) Pen∧Screen	Draws a combination of colors that are in both the pen color and the screen color.
10	Not XOR Pen	~(Pen∧Screen) ~Pen≠Screen	Draws the inverse of the XOR Pen color.
11	Nop	Screen Screen	Leaves all pixels unchanged.
12	Merge Not Pen	(~Pen) Screen (~Pen)∨Screen	Draws a combination of colors that are in either the screen color or the inverse of the pen color.

Mode	Name	C and APL Notation	Description
13	Copy Pen	Pen Pen	Draws the pen color (default).
14	Merge Pen Not	(~Screen) Pen PenV~Screen	Draws a combination of colors that are in either the pen color or the inverse of the screen color.
15	Merge Pen	Pen Screen PenVScreen	Draws a combination of colors that are in either the pen color or the screen color.
16	White	0xfffff00 255 255 255	Draws white pixels.

The table above shows the mode, its name, the notation in the C language and in APL, and a description of the effect. The symbols in the top line of each entry in the third column correspond to the C language notation for logical functions: & = AND; | = OR; ^ = XOR; ~ = NOT. The second line of the third column is the corresponding notation using APL syntax.

- **PolyFillMode**

Determines how the Draw method fills the interior of figures you draw using the *Poly* operation. The default value (0) is alternate; you can also specify (1) winding. By default, the system fills only those areas that can be reached from outside the figure by crossing an odd number of lines. Winding means the system fills all areas in the figure. (See the technical note, below.)

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, Richedit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

When you specify the mode property, you must specify one, two, or three integers. If you specify only one integer, it applies to *TransparencyMode*. If you specify two values, the system does not change *PolyFillMode*. To leave any of the modes unchanged, specify zilde (⊖). For example, to set *PolyFillMode* without changing *TransparencyMode* or *RasterOpMode*, you type:

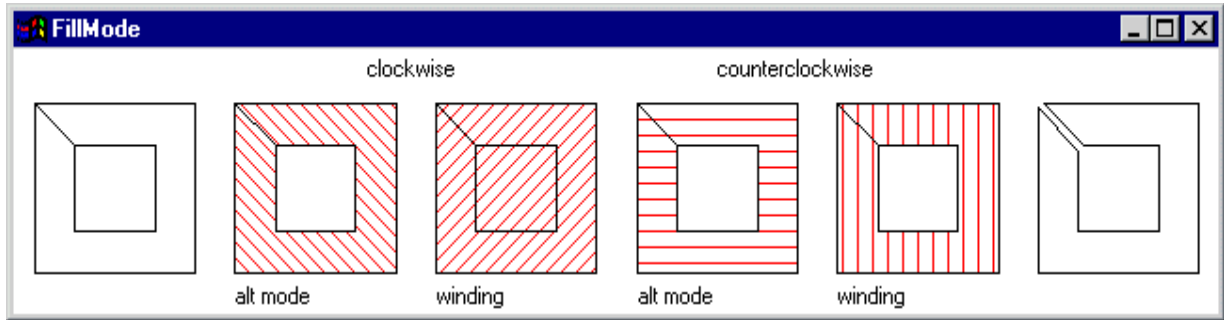
```
'fmMain.fDrawbox' ⊞wi 'Draw' ('Mode' ⊖ ⊖ 1)
```

Technical Note: In complex polygon figures where lines are coincident or cross, the concept of what is *in* the figure is not always evident. A point can be inside or outside a figure depending on which direction you draw the lines. The technical rule is that a point is inside the figure if you draw a ray from the point to infinity and the number of clockwise-directed line segments that the ray crosses does not equal the number of counterclockwise-directed line segments that the ray crosses. If you want to see the effect of this rule, execute the following example. (If your screen cannot accept a width of 480 using scale 3, set the scale property to 5.)

```
⊞wself ←'fmFillMode' ⊞wi 'New' 'Form' ('scale' 3) ('caption' 'FillMode')
⊞wi 'Set' ('size' 100 480) ('color' (255 255 255)) ('where' 0 0)
⊞wi '.12.New' 'Label' ('where' 88 88 15 70) ('caption' 'alt mode')
⊞wi '.13.New' 'Label' ('where' 88 168 15 70) ('caption' 'winding')
⊞wi '.14.New' 'Label' ('where' 88 248 15 70) ('caption' 'alt mode')
⊞wi '.15.New' 'Label' ('where' 88 328 15 70) ('caption' 'winding')
⊞wi '.16.New' 'Label' ('where' 2 140 15 120) ('caption' 'clockwise')
⊞wi '.17.New' 'Label' ('where' 2 280 15 180) ('caption' 'counter
Ⓜ clockwise')
clck←10 2p20 8 20 72 84 72 84 8 20 8 36 24 36 56 68 56 68 24 36 24
⊞wi 'Draw' ('Poly' clck)
⊞wi 'Draw' ('Brush' 4 (255)) ('Poly' (clck+10 2p0 80))
⊞wi 'Draw' ('Brush' 5 (255)) ('Mode' ⊖ ⊖ 1)('Poly' (clck+10 2p0 160))
ctrclck←clck ⊖ ctrclck[7 9;]←ctrclck[9 7;]
⊞wi 'Draw' ('Brush' 2 (255))('Mode' ⊖ ⊖ 0)('Poly'(ctrclck+10 2p0 240))
⊞wi 'Draw'('Brush' 3 (255))('Mode' ⊖ ⊖ 1)('Poly'(ctrclck+10 2p0 320))
```

The first three figures are drawn with all lines going clockwise. In the last two figures, the inner square is drawn counterclockwise. The second and fourth figures use alternate mode. The third and fifth figures use winding mode. A point in the very center of the smaller square is inside the polygon in third figure, but outside the polygon in the fifth figure. It is as if you drew the figure with the following statement (the sixth figure in the picture below).

```
□wi 'Draw' ('Poly' (10 20 20 410 20 472 84 472 84 408 22 408  
↳ 38 424 68 424 68 456 36 456 36 426))
```



modified

Description:

Three-state flag that indicates that an object or one of its descendants has changed since it was opened or since the flag was last reset.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, and Tree.

Remarks:

The value of this property is 0 if the object and all of its descendants are unmodified; 1 if the object has been explicitly modified; or -1 if a descendant of the object has been modified and the object itself has not been explicitly modified. If you modify an object, such as by changing its value, or if a user takes an action that changes the value, the system sets this property to 1 and sets the modified property of each parent up to the top-level object to -1 (unless the parent has been explicitly set, in which case its flag remains as 1).

If you reset this property to zero, the system resets any nonzero modified property of a descendant to zero; it also resets the parent if the parent's modified property is -1 and if this child is the only one whose modification caused the parent to be modified. If a sibling or a descendant of a sibling is also modified, or if the parent was explicitly modified, the parent remains modified. This effect ripples up through the parentage structure until it reaches the top-level object or an object it cannot modify. You can clear the flag either by setting this property or by using the Modify method. When the modified property changes, it triggers the onModified event handler.

The following events cause the system to set the modified property automatically. You can set the modified property of any control object under program control. Note that the onChange event for a Label occurs only as the result of a DDE transaction:

- onClick List, Check, Option, and Combo
- onChange Combo, Edit, and Label
- onSelect Listview

modifystop

Description:

Flag that indicates the ripple effect of a modification should stop with this control.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, and Tree.

Remarks:

If you set this property to 1, the ripple effect of modifying objects stops at this level. If you set modifystop to 1, when the user changes a descendant of this property, the system sets the modified property of this object to -1, but does not set the modified property of the parent of this object.

The default value for modifystop is zero for control objects, but 1 for Form, MDIForm, and Status. See the description of the modified property, above, and the description of the onModified event handler in the next chapter in this manual.

monthdelta

Description:

Scalar or one-element integer giving the number of months that a DateTime control scrolls when the user clicks the scroll button.

Class: DateTime.

Remarks:

The value defaults to the number of months that are visible at one time; for the visible calendar style, you can display more months by making the control larger.

msversion (read-only)

Description:

Integer scalar that returns a value corresponding to the version number that Microsoft assigns to its RichEdit control.

Class: RichEdit.

Remarks:

Microsoft provides at least three different versions of the RichEdit control. Version 3 is required for some features; for example, numeric or alphabetic bullet formats. If you have version 2, the control may store the formatting values but not display the effect; in this case, if you move the application to another machine that has a higher version, the effect does display. If a user has version 1, some property values may not be retained, and certain restrictions apply; for example, you are limited to a single Undo action.

The MSDN Library provides the following information: The original specification for RichEdit controls is Rich Edit 1.0; the current specification is Rich Edit 3.0. Each version of RichEdit is a superset of the preceding one, except that only Asian builds of Rich Edit 1.0 have a vertical text option. If you use the features requiring version 3, you should verify which version is installed before creating a RichEdit control.

The DLL for Rich Edit version 1.0 is named `Riched32.dll`. The DLL for versions 2.0 and 3.0 are both named `Riched20.dll`. Windows 2000/XP includes version 3.0 and newer with a version 1.0 emulator.

name

Description:

Character vector that specifies the last part of the name by which you identify the object.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

The name property includes everything after the final period. This differs from the `self` property, which is the fully-qualified name. The two are equivalent only for a top-level object.

Examples:

This example creates a Check control object named `fmMain.ckRange`; referencing `name` returns only the last name of the object.

```
'fmMain.ckRange' □wi 'New' 'Check'
fmMain.ckRange
'fmMain.ckRange' □wi 'name'
ckRange
```

newclasses

Description:

Empty character vector by default. You can assign the names of those built-in classes that you want to modify upon creation using the `onNew` event handler. See the Creating User-Defined Classes chapter in this manual for more information.

Class: # (the system object).

Remarks:

The `onNew` event handler is triggered for any built-in class named in the `newclasses` property, except that the event handler can be suppressed by putting an asterisk in front of the class name in the `□wi` statement. This allows you to choose to create either a normal instance of a built-in class or an instance modified to your specifications. You may also include your user-defined class names in the `newclasses` list.

noredraw

Description:

Boolean flag that allows you to suppress Paint events on an open and visible control. The default value, zero, allows normal operation.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

You may use this property to reduce flickering while updating a control. For example, if you want to add a number of items to a Listview control one at a time with the `InsertRows` method rather than restructuring the list property, you can set this property to 1 before starting the update and set it back to zero after completing all the insertions. You must then invoke the `Paint` method; the control is not automatically redrawn after this property is reset to zero.

Note that it is dangerous to leave this property set at 1 for very long. Many situations and events cause a `Paint` event to occur, some of which are outside the scope of APL. The `Draw` method, the `Paint` method, covering the control with another window and exposing it, moving its parent, or resizing it, all generate a `Paint` event normally. If you suppress the event for long, your application could suffer visual deterioration.

If you hide or close the control and then show it, the system does redraw it as part of creating the window. However, if you have moved it, the screen area previously occupied by the window may not be repainted. The system may also re-create an open control under certain circumstances, such as changing its border.

normsize (read-only)

Description:

Two-element read-only numeric vector that describes the calculated height and width required to show all the children of a command bar when it is docked. It is measured in the bar's scale units. The values do not include the area that contains the gripper and caption.

Class: `CommandBar`.

Remarks:

The values for this property change when the vertical/horizontal orientation changes. If the bar is not open, the system returns 0 0. One element (depending on orientation) of this property determines the default value for `dockideal`. When the bar is floating, it retains the values for its docked state.

null

Description:

Returns a variant type 1 when referenced.

Class: `#` (the system object).

Remarks:

You can use this to pass data to an ActiveX control or server. In some applications, for example, relational databases, this value signifies the existence of a field, but one without a value. Compare this with the empty property.

obj

Description:

Integer that represents a pointer to an ActiveX control or server. You can use the pointer for low-level calls to the object. This value is "typed," but contains the same raw value as the `interface` property.

Classes: `ActiveControl` and `ActiveObject`.

Remarks:

In some cases, you can use this pointer directly as an argument to another ActiveX action where you would have to use the `VT` function to "wrap" the normal integer returned by the `interface` property.

This pointer is transitory; it may be valid only for the duration of a callback. Thus, you should reference and use it in the same function; do not store it in a global variable and try to use it during a different callback. APL+Win must keep track of these pointers in order to release the underlying Windows resources when appropriate. Thus, each pointer is invalidated when a callback returns.

opaque

Description:

Integer scalar that defines the behavior of the opaque feature.

Classes:

`Check`, `CommandBar`, `Edit`, `Frame`, `Label`, `List`, `Listview`, `Media`, `Option`, `Picture`, `Status`, `Toolbox`, `Trackbar`, and `Tree`.

Remarks:**Values for opaque property**

Value	Description
Primary Value: Choose only one value	
0	Background is transparent
1	Background is opaque (default)

The opaque property can have a value of 0 or 1. The default is 1. A value of 0 causes the background of the object to be transparent such that the parent window behind it is visible through the background area of the object. For example, a `Check` object that's a child of a `Picture` object will display the image assigned to the picture as the background of the `Check` object. Similarly, when the opaque property is set to 0 the child object will, in general, ignore whatever background color is specified or inherited by the object. For example, when you set the background of a `Check` object to color (255 0 0) it will be painted with a red background when opaque is 1. But when opaque is 0 then the red background color will be ignored, since the background becomes transparent and displays the parent control as the background of the `Check` object.

In general, the opaque property is only really useful for objects like `Check`, `Option`, or `Label` that do not completely paint their client area. Objects such as `Button`, `Combo`, and `List` are not good candidates for setting the opaque property to a value of 0. Doing so can cause interesting but not necessarily useful visual effects. For example, setting opaque to 0 on a `List` object that's a child of a picture object will allow the underlying picture to be displayed below the list, but only the currently selected list item will be clearly visible. It is recommended that you use this property with caution.

opened (read-only)

Description:

Boolean flag that indicates whether an object is open or closed. The default value is 1, open.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, and Tree.

Remarks:

This is a read-only property you can use to determine whether an object is currently open or closed. You can use the Open and Close methods to change the state of an object. For visible controls and forms, the Show, Hide, and Wait methods implicitly open an object if it is not already open.

The state of being open means that an object is active; it is able to receive Windows messages and perform callbacks in response to events. Setting a property with `True` signals an event as a side-effect only on an open object. The opened state also determines whether an object has a Window handle in Windows (for visible classes and Imagelist), a device context (Printer class), or a timer id (Timer class). Forms and control objects also have enabled and visible properties, which are independent of opened.

order

Description:

Integer scalar or one-element vector that defines the sequential position of a control among its siblings. Siblings generally are controls other than menus and command bars that are first-level children of the object that contains them. Their order values are a sequence of positive integers starting with 1.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Frame, Label, List, Listview, Media, Menu, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Toolbox, Trackbar, and Tree.

Remarks:

For controls that get the focus, the order property defines where the focus moves when a user presses the Tab key. For objects that are grouped, the tab moves to one object in the group and the arrow keys move the focus among the members of the group. Pressing the Tab key when the focus is on any member of the group places the focus on the next ungrouped object or one member of the next group, whichever immediately follows the order value of the last member of this group. When a group consists of Option buttons, the focus moves to the one that is selected; when a group consists of Check boxes, the focus moves to the first object in order number regardless of how many are checked.

As you use the New method to create control objects on a form, the system assigns each control the next available order number. If you assign a noninteger value to the order property for an object, the system rennumbers all objects with order values higher than your assigned number. For example, if you assign the value 2.4 to an object, the system sets order for this object to 3, and adds 1 to the order value for the object that was previously 3 and each object with a higher order value. If you create two or more Check or Option objects sequentially, the system defines them as a group.

Typically, there is a nongrouped object that separates a group, such as a Label above or a Frame around the group. If you do not have a natural separator, you can create a hidden Label to separate distinct groups of Check or Option objects.

Menu items have separate order property sequences. The order of the top-level menu items is a separate sequence of integers distinct from the order values of other controls, such as buttons and frames, that are children of the same form. The order of the menu objects that are the choices of each drop-down menu are distinct from the top-level menu order values and from each other.

Command bars have separate sequences for each side of a container where they are docked and for those that are floating. If a command bar is styled as a client, it is part of the sequence with other controls. Command bars change groups of siblings when they dock on a different edge of a container or float. The siblings property for a command bar lists itself and all the other command bars that are part of the set of objects that share a sequence of values in their respective order properties.

You can explicitly determine the sequence of movement of the focus when a user tabs by arranging the order properties and setting the tabparent, tabgroup, and tabstop properties of various controls.

orientation

Description:

Character vector that tells how subsequent pages will be printed; the allowable values are 'portrait' (the default value) or 'landscape'.

Class: Printer.

Remarks:

You must set this property before you place anything on the page to which you want it to apply. When you set orientation, the Printer starts a new page.

origin

Description:

Two-element numeric vector that specifies the placement of the visible portion of the image in the client area of a Picture control.

Class: Picture.

Remarks:

The elements of this property define the row and column (in current scaling units) of the upper left corner of the portion of the image that is visible in the Picture control. Initially, both values are zero. If you define scroll bars for the Picture control, these two values change as the user scrolls.

overlays

Description:

Integer vector with a maximum length of fifteen elements. You specify the indices to those images in the imagelist you want to use as overlays.

Class: Imagelist.

Remarks:

When you want to use an overlaid image in a Tree or Listview object, you specify the index value between 1 and 15 in the overlays vector rather than the direct index into the imagelist.

padding

Description:

Two-element numeric vector specifying the vertical and horizontal padding (in pixels) around the image and label on each tab.

Class: Selector.

Remarks:

See also the `imagespace` property for a button.

page (read-only)

Description:

Integer that specifies which page of a print document you are currently working on.

Class: Printer.

Remarks:

When you first open the Printer, its `page` property is set to zero. As soon as you use the `Draw` method or the `Print` method for the first time, the system changes the page number to 1. When you use the `Eject` method or the `Clear` operation of the `Draw` method, the page is complete. The system increments the page setting by 1 when you use the `Draw` or `Print` method on a new page. When you `Close` the Printer, the system resets the page property to zero.

pages (read-only)

Description:

Nested vector of character vectors containing the last part of the name for each child of class `Page`.

Class: Selector.

Remarks:

The order of the `Pages` in this list is determined by the `order` property of the `Pages`. This is the same as the order in which the tabs on the `Selector` are displayed.

palette

Description:

Numeric vector or an N-by-3 array that describes the colors used by the application, or a singleton value that is a handle to a palette. You can set this property if the display driver supports Windows palette operations; if not, this property becomes read-only.

Classes: Printer and # (the system object).

Remarks:

You must set the `palette` property for the system object explicitly each time you start APL; its value is not saved. The system signals a `DOMAIN ERROR` if you attempt to assign the `palette` property when the device does not support custom palettes or if the number of colors you attempt to define exceeds the number of colors in the palette (256) minus the number of reserved colors (20).

When you set the `palette` property, the system redraws all existing forms and controls using the new color system. Any subsequent color definitions are relative to the new palette. For devices that use palettes, the color values you specify are not hardware values but are interpreted by the device relative to its palette definition. Therefore, when you request a color, the color is not used directly, but the system searches the palette for the color. If it finds a match, it uses that color. Otherwise, it uses the closest matching color available in the palette or creates a dithered pattern to simulate the color.

You specify color values as you do for the `color` property of an object. If `palette` is an array you specify each row as a composite color value with three values from 0 to 255 for each of red, green and blue. If `palette` is a vector, you specify each color as a scalar using the base value primitive function with the three values to the base 256: `256 ⊔ blue green red` (note the reversed order).

You can also set this property using a singleton value that is a handle to a defined palette.

paper

Description:

Integer designating the current paper size; one of the numbers in column 1 of the `papers` property.

Class: Printer.

Remarks:

The number to name correspondence may vary among printers.

papers (read-only)

Description:

Two-column matrix, consisting of paper size number and paper size name as recognized by the printer driver of the current default printer.

Class: Printer.

Remarks:

A paper size in this context can include envelopes as well as sheets of paper measured in inches or metric designations. This property shows the possibilities for the current printer. The paper size name is merely descriptive; you can reference or change the current setting using the `paper` property.

pending

Description:

Vector of nested vectors that is saved in the `def` property containing methods for an ActiveX object that are invoked when you create a new instance of the control using the `def` property.

Classes: `ActiveControl` and `ActiveObject`.

Remarks:

This property allows you to invoke methods that are part of an ActiveX object as part of its definition.

Example:

```
'fmCalc.ocxGrid' ⊔wi 'pending' (⊔('XColWidth' 1 30))
```

place

Description:

Two- or four-element vector (similar to the `where` property) that specifies the location and size you want a form to have when it is restored from a maximized or minimized state.

Classes: Form and MDIForm.

Remarks:

A form can exist in three states when it is open, enabled, and visible. If it is minimized, it exists only as an icon in the task bar where a user can click on it to show its contents; if it is maximized, it takes up the entire portion of the available screen (excluding task bar, tool bars, and status lines, etc.). The state in between minimized and maximized, when the form has some or all of its contents visible but it does not take up the entire screen, is called "restored." Usually, a user can control these states with buttons in the upper right corner of the form beside the "close" button that removes it from Windows.

When a form is minimized or maximized, you can set the `place` property to specify the location and size you want the form to have when it is restored. In addition to numeric values, you can use the equal character (`' = '`) to indicate the element should not change, or you can use the question mark (`' ? '`) to indicate the system should use the default setting. This is particularly useful for forms that are children of an MDI form. In this case, minimized forms show an icon at the bottom of the MDI form, and maximized forms take up the entire client area of the parent MDI form (as opposed to the entire screen). An MDI form can be maximized to take up the entire screen.

When a form is in a restored state, or when it is closed, `place` is identical to `where`; setting one changes the other. The interaction of `place` with the `size` and `extent` properties for a closed form is the same as it is for `where` on a closed form, except that when you reference the `place` property, it always returns a four-element vector. It may show the equal character or question mark for some of the elements.

Warning: In future systems, the `place` property may have more than four elements. If you reference it with the intention of using the values in your program, you should use a statement like

```
(r c h w) ← 4 ↑ 'fmMain' □wi 'place' rather than the simple statement
(r c h w) ← 'fmMain' □wi 'place'
```

Additional values would be catenated to the existing vector, so it is strongly suggested that you avoid statements like `(h w) ← -2 ↑ 'fmMain' □wi 'place'`

pointer

Description:

Integer scalar or one-element vector that specifies the appearance of the mouse pointer while it is over the object with this property, or a character vector that specifies a file with an image for the pointer.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

If you specify a pointer for a form, it applies across the entire form; if you return the form's `pointer` property to zero, each control's value then determines the appearance while the mouse is over that individual control.

If you set the pointer property for a `Frame` or a `Label`, that also activates the `onMouseXx` event handlers for that control. If the pointer property is zero and none of the `onMouseXx` event handlers are set, these objects are transparent to the mouse.

The table below shows the numeric values you can use for the pointer property. You can also use an arbitrary icon; to do this, you specify a `-1` in the pointer property followed by the icon's window handle. You can use the `Draw` method, `Icon` operation with a question mark to find the handle, or you may get an icon handle from another source.

Values for the pointer Property

Value	Description
0	Default (shape determined by control or Windows)
1	Arrow
2	Cross-hair
3	I-Beam
4	Box (Icon)
5	Size (four points North, South, East, West)
6	Size (NE SW)
7	Size (NS)
8	Size (NW SE)
9	Size (WE)
10	Up Arrow.
11	Hourglass (Wait)
12	International No (slashed circle)
13	Small hourglass (application starting)
14	Help (Question Mark)
15	Pen

In addition to numeric values, you can define the pointer by specifying a character vector that is the name of a file with one of the extensions `.ICO`, `.CUR`, or `.ANI`, or you can specify a resource using the greater than sign (`>`) notation; for example,

```
'fm1.edFinger' wi 'pointer' '>cur 1001 c:\windows\system\shell32.dll'
'fm1.ed60' wi 'pointer' '>ico rel60 c:\aplwin70\examples\rel70.dll'
```

Note that if you use an icon for a pointer, the location of the mouse click, for example in the `onMouseDown` event handler, occurs at the center of the icon. This may affect the pointer's usability.

position

Description:

Two-element numeric vector that specifies the location on the page where the next output generated by the `Print` method will occur.

Class: `Printer`.

Remarks:

The value of this property is measured in the units set by the `scale` property for the `Printer` object. It determines the distance (row and column) from the upper left corner of the logical page where the next text printed by the `Print` method will appear. You can set the `position` property to control this location. `Print`

causes a page eject whenever the first element of position plus the height of the character cell about to be printed extends beyond the height of the page.

poslargeimage

Description:

Three-column numeric matrix that specifies the positions of items in a Listview in largeimage viewmode. The first column contains the RowIDs of items; the second and third columns are row and column coordinates of the upper left corner of each item in the current scaling units for the listview.

Class: Listview.

Remarks:

If you create the listview with one of the other viewmodes, this property is empty initially. It obtains values when you set the viewmode appropriately, or you can set this property directly. You can reference and set values for selected items using the `GetPos` and `SetPos` methods. The values remain when you switch viewmodes.

If you set this property, you can choose to specify only a subset of the items. When you switch to the largeimage viewmode, the system places those items with values first; it then places the remaining items according to its default algorithms and sets the remaining values for this property. If you add an item when you are largeimage viewmode, the system assigns a value only for that viewmode.

possmallimage

Description:

Three-column numeric matrix that specifies the positions of items in a Listview in smallimage viewmode. The first column contains the RowIDs of items; the second and third columns are row and column coordinates of the upper left corner of each item in the current scaling units for the listview.

Class: Listview.

Remarks:

If you create the listview with one of the other viewmodes, this property is empty initially. It obtains values when you set the viewmode appropriately, or you can set this property directly. You can reference and set values for selected items using the `GetPos` and `SetPos` methods. The values remain when you switch viewmodes.

If you set this property, you can choose to specify only a subset of the items. When you switch to the smallimage viewmode, the system places those items with values first; it then places the remaining items according to its default algorithms and sets the remaining values for this property. If you add an item when you are smallimage viewmode, the system assigns a value only for that viewmode.

printername

Description:

Character vector that tells the name of the current printer. You can set it to any valid printer name; resetting this property to an empty character vector selects the default printer.

Class: Printer.

Remarks:

The list of names of local and attached printers is returned in the first column by the Printers method on the system object.

progid

Description:

Character vector that returns the programmatic ID of an ActiveX object.

Classes: ActiveControl and ActiveObject.

Remarks:

This is a three-part identifier delimited by periods. The first part is the manufacturer's name; the second part is a server name; and the third part is the ActiveX version number. This information is also returned as the third column of the xclasses array for ActiveX controls (ActiveControl classes) and the first column of the result of the XInfo method (ActiveControl and ActiveObject classes).

prompt

Description:

Arbitrary data associated with an object; typically a character vector that the system displays when the object is highlighted.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

You can use this property to store a message that the system displays when a user moves the focus to a control or highlights a menu item and triggers the onPrompt event handler of the system object. This effectively allows you to have dynamic online usage hints for any part of your application.

properties (read-only)

Description:

Nested vector of character vectors that lists all of the properties for this object.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

This property provides a read-only list of the properties you can use for an object of this class; you can reference it directly. You can determine which properties of an object have nondefault values defined by referencing its `state` property.

range

Description:

Integer scalar or vector that defines the lines that are visible for a list or an edit field or the lower and upper bounds for a spinner, trackbar, date/time or multimedia device.

Classes: Combo, DateTime, Edit, List, Media, RichEdit, Spinner, and Trackbar.

Remarks:

For a Combo, Edit, List, or RichEdit, this property is a two element numeric vector that describes the visible content of the control. For a Combo box, the first element is -1 ; for the others, the first element is the line number of the topmost item in the list. The second element is the number of visible lines in the list box. Except for a Combo, you can assign a scalar to scroll the list so the line number you assign is at the top. For a RichEdit control, the second element is read-only. For the others, if you assign both values, you effectively change the size of the control.

This property specifies the visible selections in a List or the number of visible items in the list portion of a Combo box. For an Edit or RichEdit control, it is meaningful when you specify multiline (one that has 4 added in its style property); then, it specifies the visible lines in an entry that exceeds the vertical size.

For a Spinner, this property is a two-element vector that specifies the lower and upper bounds for the spinner values. The default range is $0 \ 100$. The magnitude of the range cannot exceed 32767 . However, the upper limit is 32767 , and the lower limit is -32767 .

If you set the two elements of the range to the same value, that effectively disables the spinner, by inhibiting the `onSpin` and `onChange` event handlers. If you set the first element of the range to be greater than the second element, the actions of the arrows are reversed; that is, clicking on the up arrow decreases the value and clicking on the down arrow increases it.

For a Trackbar, this property can be a scalar or zero-, one- or two-element integer-valued numeric vector that specifies the minimum and maximum values for the trackbar. If you specify a two-element vector, the elements are the minimum and maximum values. If you specify a scalar or one-element vector, the system assumes a zero for the other value; you can specify a negative value for the minimum or a positive value for the maximum. If you specify an empty vector, the system restores the default values of $0 \ 100$.

For a Media control, this property is a read-only, two-element vector that specifies the lower and upper bounds for the file. The values express numbers of frames for `.AVI` and `.ANI` files. For other types of files, the units may vary; for example some `.WAV` files are measured in milliseconds. For some devices, you can change the format; this may result in an obscure value that represents packed data.

For a DateTime control, this is a two-element nested vector of three or seven-element vectors showing the minimum and maximum dates or times the control allows the user to select. You may set either or both elements; if an element is an empty vector, there is no limit for that end of the span other than the built-in limits of the control.

recursions

Description:

Three-element numeric vector that specifies the limit on the number of nested callbacks between APL and the APL Windows interface and the maximum and current number of recursions. The first element specifies the limit to the number of recursive calls, the second element contains the highest number of recursions since the start of the session or since you last set the limit, and the third element shows the current number. The system ignores the third element if you set it. The default limit is 50.

Class: # (the system object).

Remarks:

When your application code calls a function in the APL Windows Interface, for example `□wi`, the function can call your code back, for example, to run an event handler. The event handler can, in turn, call a function and begin an infinite recursion. The `recursions` property halts an infinite recursion.

The default `recursions` value should be adequate for most applications. If you ever reach the recursion limit, the most likely cause is a recursive event handler in your code; for example, an `onClose` event handler that invokes the `Close` method. Looking at `⌋si` should give you an idea of where the problem is occurring. If you want to simplify debugging, you can set `recursions` to a small number to identify the problem sooner. If you get an error while your code is behaving normally, you can increase the value.

If the value for `recursions` is too large for your system, you may run overflow a Windows stack or run out of memory before you get a `WSFULL`, so the error message you may receive is not predictable.

The current recursion count can be higher than the number of callouts (`<<`) and callbacks (`>>`) that appear on the stack. Nested executions are included in the recursion count but do not appear on the stack. (Nested executions can occur, for example, when you use the `New` method within `New`.) See the description of callouts and callbacks in the *Programming the Windows Interface: Basic* chapter for more information.

The current recursion count can also exceed the limit. The system signals an error only on a callout and not on a callback. If you exceed the limit on a callback, you encounter the error on the next callout. For the same reason, the maximum number shown in element two of the vector can also exceed the limit.

The maximum number can exceed the limit for another reason: When you set `recursions` to establish a limit, you can also set the maximum number or preserve the current value of the maximum number. The maximum number is not tied directly to the current limit. **Note:** If you want to reset the limit when you have reached it, you must strip a level in `□si` to be able to set the limit.

redoname (read-only)

Description:

Character vector, when a `RichEdit` control is open, that indicates the type of action that can be redone; when the `canredo` property is zero, this property is an empty vector.

Class: `RichEdit`.

Remarks:

After an editing action in a `RichEdit` control has been undone, by the user pressing the `Ctrl+Z` keystroke combination or the program invoking the `Undo` method, the action can usually be redone, by pressing `Ctrl+Y` or invoking the `Redo` method. This property gives an indication of the type of action that is first on the Redo stack. Possible values include: `typing`, `delete`, `cut`, `paste`, `dragdrop`, `unknown` or an

empty vector. The value is 'unknown' for character and paragraph formatting operations; it is empty if nothing can be redone.

restartArg

Description:

Character vector that specifies the value of the %ARG% variable that can be substituted into the restartCmd or restartMsg property strings.

Class: # (the system object).

Remarks:

In practice, this isn't very useful when specified statically in the INI file at application startup since it could be coded directly into the restartCmd property. The concept of a RestartArg property becomes effective when an application sets it dynamically while running to provide a process specific restart context (such as current transaction number). It cannot be set by changing the INI file while running because the restart mechanism does not read those values during a system crash. It only reads (and caches them) at application startup time. But you can use the RestartArg property of the system object like this:

```
'#' □ WI 'restartArg' 'context=993.221'
```

restartCmd

Description:

Character vector that specifies the command line for the recovery application to be executed when an APL system crash is detected; it includes the executable name and its arguments.

Class: # (the system object).

Remarks:

This execution occurs regardless of whether the crash is by accident (such as an assertion failure or access violation by APL+Win itself) or intentional (when the application executes the □IT 'crash' command to intentionally crash and restart itself). This allows applications to simply "crash and recover" using a standard mechanism. If you do not specify a command in restartCmd then APL+Win looks to restartMsg and displays that information in a message box to indicate system failure. If no value is specified in restartMsg, then APL+Win displays its own standard failure message box.

If you do not want any restart action to take place including no messages to the user, then you can assign none to restartCmd. This is often appropriate for APLW.WSEngine instances where it is presumed that controlling application can monitor failures and handle restarting of failed worker sessions. Using the none value does not suppress generation of .log and .dmp files when a crash occurs. It simply prevents display of a message box by APL+Win when it crashes and doesn't start a recovery application running.

restartDir

Description:

Character vector that specifies the current directory setting that is to be in effect when the recovery application is launched.

Class: # (the system object).

Remarks:

This defaults to the same directory as the executable specified in the `restartCmd` property.

restartMsg

Description:

Character vector that specifies the error message box to be displayed when the system crashes and application restart fails or when the APL+Win system is not configured for automatic restart.

Class: # (the system object).

Remarks:

This property specifies a template for the error message box to be displayed when the system crashes and application restart fails or when the system is not configured for automatic restart. The message strings is limited to 4000 characters. The default value is:

%APPNAME% is closing due to an unrecoverable error.

Please contact the application vendor for assistance
if this problem persists:

%CONTACT%

See error log file for details:

%LOGFILE%

restartRate

Description:

Character vector that specifies the number of crash restart cycles that are allowed over a given time span.

Class: # (the system object).

Remarks:

It is specified as the number of restarts per minutes (restarts/minutes). Specify 0 to suppress restarts completely. The default value is 3/5. For example:

```
'#' □ wi 'restartRate' ' 3 / 9 '
```

allows 3 restarts per 9 minute period.

roworigin (read-only)

Description:

Integer scalar that contains the row number of the topmost visible entry in a Listview object when you are in list view or report view; in the other views, it is an empty vector.

Class: Listview.

rtf

Description:

Character vector that contains the text and associated control codes for the RichEdit control. This is the primary text property for the RichEdit control. You can also use this to read from or write to a native .rtf file.

Class: RichEdit.

Remarks:

You can set the rtf property on a closed RichEdit control, but you cannot set the text property until the object is opened.

savecontent

Description:

Boolean flag that enables the saving of the complete internal state of ActiveX properties for an ActiveX control. The default value is 1.

Class: ActiveControl.

Remarks:

If you set this property to zero, an ActiveX control does not update the value of the content property when you close the control. This means the most recently set value persists until you explicitly set it again.

scale

Description:

Five-element, integer vector that defines scaling units for the sizes of objects and positions inside them.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

The first element of `scale` is *scalemode*. Its default value is 1. The possible values for *scalemode* are:

Values for scale Property *scalemode* Element

Value	Description
0	Proportional scaling (defined in terms of $1 \div scale$).
1	Characters (1 sys-font high; 1 sys-font wide).
2	Dialog units (1/8 sys-font high; 1/4 sys-font wide); the scaling units used by external dialog box editors.
3	Points (72 points per logical inch).
4	Twips (1440 twips per logical inch).
5	Virtual pixels. (See Notes 1 and 3.)
6	True pixels. (See Notes 2 and 3.)

Note 1: There is a new configuration setting [Config]VirtualPixels that controls whether

scalemode=5 units are virtualized for DPI scaling or unscaled pixels; 1 (default) means virtual pixels are scaled to the current DPI level and 0 means not scaled.

Note 2: This behaves like *scalemode*=5 used to (pixels) except *scalemode* 6 is real pixels on the hardware and not virtualized like *scalemode*=5 when dpi is not equal to 100.

Note 3: The Printer and EMF objects support *scalemodes* 5 and 6. But these modes are both TruePixels on the printer.

When referencing the *scale* property, the system always returns a five-element vector. The last four elements of *scale* indicate the position (row, column of the upper left corner) and the size (height and width) of the object's client area in scaled units. The client area is inside the object's border, if it has one. If there is no border, it is all of the object's area. If the object is a form, its border includes the title bar and menu line, if they are present.

Setting the *scale* property does not resize the control; rather, it specifies the logical coordinates that you use to measure the object's client area. You can use the *extent*, *size*, or *where* property to resize a control. Ordinarily, when you set *scale*, you specify only the first element; you can use a scalar or a one-element vector. When the first element is nonzero, the system changes the *size* property and the fourth and fifth elements of *scale* (which match *size*) to values that represent the current size of the object in the new scaling units. You can also use a five-element vector; if the first element is nonzero, the system ignores the last four values.

The *scale* property defines the units you specify when you use the *Draw* method to draw on an object. For containers, such as Forms and Frames, it also affects the coordinates you use to position other objects contained by the form or frame.

The *size* property specifies the size of an object in terms of its own scaling units, rather than the scaling units of the object that contains it. For child objects, the *where* and *extent* properties are scaled in the units of the parent object (the one that contains it). Note that the *where* and *extent* properties of a top-level object are always scaled in character units, since you cannot specify a different scaling unit for the parent of a top-level object. For example:

```
'fmRunAll.bnStart' □wi 'scale'
1 0 0 1.5 10
'fmRunAll.bnStart' □wi 'scale' 3
'fmRunAll.bnStart' □wi 'scale'
3 0 0 18 60
'fmRunAll.bnStart' □wi 'size'
18 60
'fmRunAll.bnStart' □wi 'extent'
1.5 10
```

The 0 0 origin refers to the upper left corner of the push button.

When you specify proportional scaling (*scalemode* 0), you can specify a two-element, four-element or five-element vector. If you specify two elements or four elements, the system assumes that *scalemode* is zero. If you specify two elements, they represent the *height* and *width* of the client area (the fourth and fifth elements); the system assumes an origin of (0 0). If you specify four elements, the *row* and *column* of the upper left corner of the object precede the *height* and *width*.

When you use proportional scaling, the coordinate system for positions inside the object is defined in terms of the object's current size instead of any fixed units. This is particularly useful for graphics, where it allows you to make a drawing that is proportional to the current size of the object that contains it. For example, setting *scale* to 0 0 0 100 100 provides units that are expressed as a percentage of the window size.

Therefore, the position (50 50) would always be the center of the object, even after changing its size. Note

that when *scalemode* is zero, you cannot change the size of the object using the *size* property; you must use *extent* or *where*.

You can also use proportional scaling to effectively change the origin. For example, if you want to specify coordinates with the origin at the lower left corner of a frame, you could specify *scale* with the statement:

```
'fmRunAll.fEasel' □wi 'scale' 0 600 0 -600 600
```

Then you could draw an X in the center of the frame with these statements.

```
'fmRunAll.fEasel' □wi 'Draw' ('Line' (2 2 ρ 200 200 400 400))  
'fmRunAll.fEasel' □wi 'Draw' ('Line' (2 2 ρ 400 200 200 400))
```

scrollaccel

Description:

Numeric two-element vector or two-column matrix that describes the schedule of speeds for automatic scrolling when an object is dragged over the control.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

The first column gives the time, in seconds, before the scroll acceleration in the second column takes effect. The second column is in scroll units per second. For example, suppose you set *scrollaccel* to `2 2 ρ 0.5 8 5 20`. A half second after the cursor moves within the scroll margin the control starts scrolling at 8 units per second. Five seconds after that (5.5 seconds after moving into the scroll margin) scrolling accelerates to 20 units per second.

scrollmargin

Description:

Four-element numeric vector that describes where automatic scrolling will occur during a drag operation.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

The four elements are the height or width of the top, left, bottom, and right margins, in current scaling units, in which automatic scrolling will occur. During dragging, when the cursor is within the specified distance of an edge, the control will automatically scroll (if it has a scroll bar) in that direction.

searchstring (read-only)

Description:

Character vector containing the current incremental search string. Listview and Tree objects use the

incremental search string to select an item based on characters typed by the user. This is an empty vector if no incremental search string is in progress or if the object is closed.

Classes: Listview and Tree.

Remarks:

Note that when a user types characters that cause the focus to move, it changes both the `value` property and the `highlightfocus` property (in a Listview) or the `highlightselect` property (in a Tree). In a Listview object, in report view, the search string works on the first column.

selalign

Description:

Character vector describing the alignment of the selected text in a RichEdit control. You must include 32 in the `style` property (inhibit automatic horizontal scrolling) for this property to be effective. There are two categories of values you can use. The valid values that have a visible effect on the screen are `'left'`, `'right'`, or `'center'`.

Other values are preserved with the paragraph formatting; these values are compatible with Microsoft's Text Object Model (TOM) interface, and they will have their effect if you copy the formatted text to an application that uses such an interface. Two of the values provide block margins; that is, text that is both left and right justified. The `'interletter'` value accomplishes this by adjusting the spacing between letters; the `'justify'` value increases space between words. Other valid values are: `'scaled'`, `'glyphs'`, and `'snapgrid'`; you can find documentation on these effects in your application that uses the TOM interface.

Class: RichEdit.

Remarks:

This is a paragraph formatting style.

selborder

Description:

A five-element numeric vector that describes the paragraph border codes.

Class: RichEdit.

Remarks:

This property currently has no useful effect in APL+Win. Values may translate to applications that display borders around paragraphs.

selbullet

Description:

A character vector indicating if the selected text in a RichEdit control is bulleted (“none” or “bullet”), or integer values that specify various sequential paragraph formats: zero is equivalent to “none”; 1 specifies the standard bullet (•) for each paragraph. Other values include: 2) arabic numerals; 3) lowercase letters;

4) uppercase letters; 5) lowercase Roman numerals; and 6) uppercase Roman numerals. The corresponding character vectors are: 'arabic', 'lcletter', 'ucletter', 'lcroman', and 'ucroman'.

Class: RichEdit.

Remarks:

When using sequential paragraph formats, you can specify a starting value for the first selected paragraph in the selnumstart property (all affected paragraphs must be selected) and you can vary the format using the selnumstyle property. The selbullet property is a paragraph formatting style.

Numeric values require RichEdit version 3 (see the description of msversion); for version 2, the values are preserved with the paragraph formatting but do not display (hence may work on another machine). For version 1, the values may not be recognized.

seleffects

Description:

A twelve-element numeric vector that specify effects for selected paragraphs in RichEdit. Each element is 1 to apply the effect; 0 to not have it, -1 to leave value unchanged; when you reference it, the values are for the first selected paragraph; effects represented:

[1] keep together, [2] keep with next, [3] page break before, [4] no line number, [5] no widow control, [6] do not hyphenate, [7] side-by-side, [8] right-to-left paragraph, [9] outline level, [10] collapsed, [11] box, [12] table;

Class: RichEdit.

Remarks:

This is a paragraph formatting style.

seleffectsall (read-only)

Description:

A twelve-element numeric vector that specify if all selected paragraphs in RichEdit have the same selecteffects formatting values. Each element is 1 if all settings are the same; 0 if different; when you reference it, the values are for all the selected paragraphs; effects represented:

[1] keep together, [2] keep with next, [3] page break before, [4] no line number, [5] no widow control, [6] do not hyphenate, [7] side-by-side, [8] right-to-left paragraph, [9] outline level, [10] collapsed, [11] box, [12] table;

Class: RichEdit.

Remarks:

This is a paragraph formatting style.

selcolor

Description:

Integer scalar or three-element integer vector specifying the color of the selected text in a RichEdit control. See the description of the `color` property for the methods for specifying colors.

Class: RichEdit.

Remarks:

Note that the color property works in the normal fashion for a RichEdit control. You can set the background for the control and the foreground color for any text for which `selcolor` is not set. The `selcolor` property is a character formatting style; you can apply it to individual characters or a contiguous block. If you select multiple characters and reference this property, the system returns the values for the first one.

selection

Description:

Two-element integer vector that shows the cursor position and the length of any highlighted text in an input field or the beginning and end points of a portion of the channel of a `Trackbar` that you want to highlight.

Classes: `Combo`, `Edit`, `RichEdit`, and `Trackbar`.

Remarks:

In an `Edit` or `RichEdit` control or the edit portion of a `Combo` control, the input field may be completely or partially highlighted. The first element of the vector is the number of characters in the field that come before the insertion point or beginning of select. The second element is the number of highlighted characters.

If you specify a scalar value, the system assumes `selection` length is zero and sets the caret (insertion point) to this position. You can specify `-1` for the first element to place the cursor at the end of the field. You can specify `-1` for the second element; that selects everything after the insertion point to the end of the field.

If you set the `text` property of an `Edit`, the system resets `selection` to 0. You may need to capture this value if you adjust the `text`, for example, in an `onKeyPress` event handler. Note that this property may give incorrect values if you use `style 32` (Inhibit automatic horizontal scrolling) and the system puts in soft line breaks.

For a `Trackbar`, this property specifies the beginning and end points of a portion of the channel that you want to appear in the Windows highlight (selected items) color. The default value is an empty vector; you must specify `32` in the `style` property to enable this property. If you set this style, Windows redraws the trackbar with a wider channel to display the selection range. The values you specify must be within the values of the `range` property.

self (read-only)

Description:

Character vector that gives the fully qualified name of the referenced object.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

You can test for the existence of an object by referencing the `self` property. If the object does not exist, `❏wi` returns an empty vector. If you reference another property for a nonexistent object, you get an error message. You can use the system variable `❏wself` for referencing the current object during callbacks.

selfont

Description:

Nested vector that specifies the typeface, size, and (partial) style of the font that you want the system to use for the selected text. You can specify font as a five-element vector similar to the font property plus an offset value (for subscripts and superscripts). However, you cannot specify formatting (bold, italic, underline, etc.) or quality in the third element.

Class: RichEdit.

Remarks:

This is a character formatting style; you can apply it to individual characters or a contiguous block. If you select multiple characters and reference this property, the system returns the values of the first one. The value for font consists of five elements explained below: Trailing elements that you do not want to change can be omitted. You can specify other elements you do not want to change as $\emptyset$.

- **fontName** is the logical name of the font. If you specify $\emptyset$, the system selects a font based on the *fontStyle* value.
- **fontSize** is the logical size of the font (in the current scale units as defined by the *scale* property for this object). You can specify the height or the height and average width. Note that the system interprets *fontSize* differently for this property than it does for the font property of an Edit control.
- **fontStyle** is a numeric scalar or one element vector that indicates how the font is to be styled. The value is the sum of one value for Pitch and one value for Family. To allow you to specify the font for a selection without changing formatting styles (bold, italic, etc.) the font and formatting styles have been separated into two properties, *selfont* and *selstyle*. You cannot set formatting styles in the *selfont* property. If you set one of the font property formatting styles (1, 2, 4, or 8) in this property, an error is signalled. The values you can use include:

Values for selfont Property *fontstyle* Element

Pitch value	Description	Family value	Description
0	Default pitch	0	Do not care
32	Fixed pitch	128	Roman
64	Variable pitch	256	Swiss
		384	Modern
		512	Script
		640	Decorative

- **fontCharset** is the name of the character set (selection and order of symbols). Choose one of the following: 'ansi', 'oem', 'symbol', 'default', or an integer.
- **offset** is the distance a character is placed above or below the baseline, measured in current scaling units for this object. If you specify a positive value, the text is drawn above the baseline (superscript), and if you specify a negative value, the text is drawn below the baseline (subscript).

See the description of the font property for more detail on the first four elements.

selfontall (read-only)

Description:

Four-element Boolean vector that shows whether all selected text in a RichEdit control has the same font formatting styles.

Class: RichEdit.

Remarks:

An element is 1 if the settings are the same, zero if the settings differ among selected characters. The four elements in order are: font, font size, offset, color. Note that *fontName* is the primary determinant of the first element; you can however change the font by changing some aspects of its *fontStyle* (the third element of font, not the same as *selstyle*).

selindents

Description:

Three-element numeric vector that specifies the left indent, bullet indent and right indent for the selected paragraphs.

Class: RichEdit.

Remarks:

This is a paragraph formatting style. A 3-element numeric vector of left indent, bullet indent and right indent, measured in current scaling units. The left indent, measured from the left margin, affects the entire paragraph, including any bullet or sequential paragraph format.

The bullet indent is measured from the left indent and affects the text of a paragraph in various ways. If you have no bullets, this value affects all lines after the first (creating a hanging indent). If bulleting or a sequential paragraph format is specified, this value determines the minimum distance for the text from the left indent (the start of the bullet, letter or number). If *selnumtab* is greater than this value, *selnumtab* determines the minimum for the first line and bullet indent the remaining lines.

Right indent is measured from the right margin of the control. This value and any effect on second or subsequent lines depend on the *style* property including 32 (suppress automatic horizontal scrolling).

sellinespacing

Description:

A singleton integer value or two-element numeric vector that specifies the spacing of lines within a paragraph.

Values:

0=Single space (default)

1=One-and-a-half space

2=Double space

Values + Measure:

3 n=Space value of n (current scaling units) minimum equal to single space

4 n=Space value of n (current scaling units) causing overlapping lines, maximum equal to single space

5 m=Space m relative to single spacing; e.g., 5 1.5=space-and-a-half

Note: Single values (1 or 2) may not translate to other applications.

Class: RichEdit.

Remarks:

This is a paragraph formatting style. This effect is visible only if you inhibit horizontal scrolling (style includes 32) to force line wraps.

selnumstart

Description:

An integer scalar specifying the starting value for numbering paragraphs in a RichEdit control when you specify sequential paragraph formatting in the `selbullet` property.

Class: RichEdit.

Remarks:

If you select letters with a `selbullet` value, setting this property to zero or 1 starts with "A" or "a"; if you set this property to zero, roman numerals start with a blank. This is a paragraph formatting style.

selnumstyle

Description:

A character vector showing the formatting style for numbered or lettered sequential paragraphs in a RichEdit control when you specify sequential paragraph formatting in the `selbullet` property; the default

value is: ' 1) ' ; other valid values are: ' (1) ' , ' 1 . ' , ' 1 ' , and ' ' (an empty vector). If sequential paragraph formatting is not specified in **selbullet**, this property is an empty vector.

Class: RichEdit.

Remarks:

This is a paragraph formatting style. You specify the format in this property using the numeral one. The formatting applies equally to letters or Roman numerals; thus you might see: (A) a. i) or an unadorned l in place of a bullet.

If you have **selbullet** set to a sequential paragraph numbering style, an empty vector in this property suppresses the numbering for the paragraphs to which it is applied, but does not stop the sequence for all the following paragraphs to which the **selbullet** value and one **selnumstart** value apply. This allows you to have a multi-paragraph exposition for one number or letter in your sequence and continue automatically with the next number or letter instead of having to start a new sequence with a hard-coded, non-standard starting value.

selnumtab

Description:

A numeric scalar specifying the minimum distance between the start of a bullet or the letter or number for sequential paragraphs and the start of the text.

Class: RichEdit.

Remarks:

This is a paragraph formatting style. This value affects only the first line of a bulleted or sequential paragraph. The second value of the **selindents** property determines subsequent lines. If both are set, the greater of the two applies to the first line. If **selbullet** is zero, this property has no effect. Note that this property specifies the minimum; a sequential number format, such as VIII, could push the text further.

selfpargall (read-only)

Description:

Nine-element Boolean vector that shows whether all selected paragraphs in a RichEdit control have the same paragraph formatting styles.

Class: RichEdit.

Remarks:

An element is 1 if the settings are the same, zero if the settings differ among selected paragraphs. The nine elements in order are: alignment, bulleting, left indent, bullet indent, right indent, tab settings, starting paragraph number, sequential numbering style, and minimum indent for first line of a bulleted paragraph.

selrtf

Description:

Character vector of the selected text in a RichEdit control with associated control codes.

Class: RichEdit.

Remarks:

You can use this to query or replace formatted text. See the description of the rtf property.

selshading

Description:

A numeric scalar that describes the paragraph shading code.

Class: RichEdit.

Remarks:

This property currently has no useful effect in APL+Win. Values may translate to applications that display shading around paragraphs.

selspaceafter

Description:

A numeric scalar that describes the distance in current scaling units of space after the selected paragraph.

Class: RichEdit.

Remarks:

This is a character formatting style.

selspacebefore

Description:

A numeric scalar that describes the distance in current scaling units of space before the selected paragraph.

Class: RichEdit.

Remarks:

This is a character formatting style.

selstyle

Description:

Five-element numeric vector indicating styling for the selected text. The elements represent bold, italic, underline, strikeout, and protected, in that order. If the element is 1, the style is applied to the selected text.

If it is zero, the text does not have that style. You can specify `-1` for an element to leave it unchanged when you specify values for other elements.

Class: RichEdit.

Remarks:

This is a character formatting style; you can apply it to individual characters or a contiguous block. If you select multiple characters and reference this property, the system returns the values of the first one. If you apply the protected style to text, it does not literally protect it. When a user attempts to edit such text, it triggers the `onProtected` event handler; you can prohibit the editing by setting `OnProtected` in the handler.

`selstyleall` (read-only)

Description:

Five-element Boolean vector that shows whether all selected text in a RichEdit control has the same `selstyle` formatting values.

Class: RichEdit.

Remarks:

An element is 1 if the settings are the same, zero if the settings differ among selected paragraphs. The five elements are in the same order as the `selstyle` property elements: bold, italic, underline, strikeout, protected.

`seltabs`

Description:

Numeric vector of up to 32 tab positions for the selected text in a RichEdit object, in the scaling units of the object. Unless you include `131072` in the `style` property, a user cannot insert a tab by pressing the Tab key; this moves the focus to the next control. A user can insert a tab by pressing Ctrl+Tab.

You can also specify a type of tab stop with the advanced typography style (`262144`) set. Without that setting, the `seltabs` property is a vector of values specifying the location of tab stops, measured from the left margin. With the style setting you can specify `seltabs` as a nested vector of two- or three-element vectors. The first element is the location setting, as with all styles. The second element of each nested vector specifies the type of tab stop: 0=left justified (normal); 1=centered; 2=right justified; 3=decimal.

Class: RichEdit.

Remarks:

This is a paragraph formatting style.

`selttext`

Description:

Character vector of the selected text in a RichEdit control without the associated codes. Compare with `selrtf`.

Classes: Edit and RichEdit.

Remarks:

You can query or replace the selected text using this property.

separator

Description:

Boolean flag that indicates whether a Menu item is a separator line or a regular menu item. The default value is 0, regular menu item.

Class: Menu.

Remarks:

You can use a separator line to illustrate a logical grouping of menu items. When this property is set, the item is simply a line on the menu, independent of any of its other properties. A menu item cannot serve as a separator if it is part of the top-level menu or if it has any children.

server (read-only)

Description:

Integer pointer to the APL IUnknown interface when you have invoked APL as an ActiveX server.

Class: # (the system object).

Remarks:

This property is read-only; when you reference it, you must use this value during the callback in which you acquire it; the system releases the pointer at the end of the callback. If you reference this property on the system object for an instance of APL that is not a server, or if the connection between the client and the APL server is not established, it returns zero. Thus, you probably must wait to use this until the client has performed some kind of initialization operation.

sessions (read-only)

Description:

Three-column nested matrix that contains information for open editor session windows.

Class: # (the system object).

Remarks:

This property has one row for each open editor session window:

Column 1: The first column is the object name. This may be blank if the object has not yet been saved with a name.

Column 2: The second column is the editor session type: 'F' = Function; 'V' = Character Vector; 'M' = Character Matrix; and 'N' = Numeric.

Column 3: The third column is the changed flag: 0 = not changed and 1 = unsaved changes.

If no editor session windows are open, the result is a 0 by 3 empty matrix.

Note: Additional columns may be added in a future release, so your application should be prepared to gracefully accept a result of more than three columns if you reference this property.

shortcut

Description:

Two-element vector that defines a keystroke combination for the menu item. The first element designates the base key; the second element the shift keys you combine with the base key. When a user presses this combination of keys, the system executes the `onClick` event handler if the menu is enabled.

Class: Menu.

Remarks:

This property specifies the keystroke combination you can use to invoke a menu selection without actually clicking on the item or highlighting the item and pressing Enter. When you set the `shortcut` property, it automatically modifies the menu's `caption` to reflect the shortcut key names.

You can also define a hidden Menu as the child of a control object. When you specify a shortcut key combination for such a Menu, and you define the `onClick` handler for the Menu to match that of the control object, the shortcut key effectively triggers the `onClick` event handler for that object.

You specify the first element of `shortcut` as a character or integer singleton value. A character singleton literally means that key value; that is 'A' denotes the A key. You use a numeric value to denote a Windows virtual key code that you cannot represent with a single letter. This virtual key code also occurs as the seventh element of `¶warg` for the `onKeyDown` and `onKeyUp` event handlers. The table below shows the virtual key codes for many of the noncharacter keys. See the Virtual Key Codes appendix in this manual for the complete list.

Selected Virtual Key Codes

Code	Label / Description	Code	Label / Description
8	Backspace	40	Down Arrow (↓)
9	Tab	44	Print Screen
13	Enter	45	Insert
19	Pause	46	Delete
20	Caps Lock	96–105	Numeric Keypad 0-9
27	Esc	106	Numeric Keypad *
33	Page Up	107	Numeric Keypad +
34	Page Down	109	Numeric Keypad -
35	End	110	Numeric Keypad . (decimal)
36	Home	111	Numeric Keypad /
37	Left Arrow (←)	112–135	F1-F24 (function keys)
38	Up Arrow (↑)	144	Num Lock
39	Right Arrow (→)	145	Scroll Lock

The second element of the `shortcut` vector denotes the shift keys that you combine with the base key. It is the sum of any of the values shown below.

Shift Key Values

Value	Description
0	Base key alone
1	Shift
2	Ctrl (the default)

Value	Description
4	Alt

showtext

Description:

Boolean flag that you can use to show (1) or hide (0) captions on command buttons.

Class: CommandBar.

Remarks:

The default value is 1, but under certain circumstances, you may have to set this property to zero and reset it to 1 to cause command button captions to appear; for example after setting captions on buttons that already existed on an open form.

siblings (read-only)

Description:

Nested vector of character vectors that lists all the command bars that are part of the set of command bars that share a sequence of values in their respective `order` properties.

Class: CommandBar.

Remarks:

Command bars, unlike other controls, can move from one set of siblings to another. Command bars form dynamically changing sets of siblings as they dock and float. A command bar can belong to one of six different sets of controls that each have their own `order` values. Bars docked at the top, bottom, left side and right side of the container are each a separate set of controls for purposes of `order`. Floating bars have their own set of `order` values. If you make a bar a client of the container (by setting the first element of `dock` to `-1`), it becomes part of the set of all "normal" controls that are children of the container.

Command bars are the only objects that can change the set of siblings to which they belong without being destroyed and recreated.

The `siblings` property returns a list of all the command bars that are docked on the same side of the container for a docked command bar, and a list of all the floating command bars if the bar on which you query this property is floating. The value returned is meaningless and may change in the future for a command bar that is a client of its parent container.

For controls other than command bars, you can find the corresponding list by querying the children of its parent, except that first-level menus that are children of a form (or MDI form) constitute a different set for `order` than non-menu children of the same form (or MDI form):

```
control [wi] '..children'
```

size

Description:

Two-element vector that defines the height and width of the object's client area in scaling units that its scale property defines. This property is read-only for Page, Printer, and Status objects and the system object.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

The client area is the area inside an object's border, if it has a border. The border of a form includes the title bar and menu line. If there is no border, the client area is the same as the size of the object. If you add items to a control, such as a scroll bar in an edit control or a status bar on a form, the client area is reduced. The most common use for size may be on a form, because the client area differs from the area of the entire object.

The size property is related to the extent and where properties. See the description of the where property for the relationships among extent, size, and where. For a definition of scaling units, see the description of the scale property.

size for the System object is read-only; it returns the size of the Windows desktop (the screen) in character scaling units. size for a Printer object is read-only; it returns the size of the page.

sliderlen

Description:

Scalar that specifies the width of the slider in a vertical trackbar or the height of the slider in a horizontal trackbar. The default value is -1, which means the system sets the size automatically.

Class: Trackbar.

Remarks:

You can determine the actual size by referencing the sliderwhere property. If you specify a positive value, you define the base-to-point length of the slider in current scaling units. If you set this property to zero, the slider is hidden. The user cannot drag a hidden slider, and there is no visual indication of the current value; however, the user can change the value from the keyboard with the arrow keys and the PageUp and PageDown keys.

sliderwhere (read-only)

Description:

Four-element vector that specifies the position and size of the slider in a Trackbar object. The elements, in order, are row, column, height, and width (similar to the where property).

Class: Trackbar.

sortorder

Description:

Integer vector of RowIDs of the list property of a Listview object permuted in the order the rows are sorted.

Class: Listview.

Remarks:

You can use this property to quickly reorder the rows of the Listview, for example in the onColClick event handler, to change the display in report view.

sourceformats

Description:

Zero or more character vectors that specify the format for dragging items from this control. The valid values are 'ListviewItem' and 'Text' for a Listview object and 'TreeItem' and 'Text' for a Tree object.

Classes: Listview and Tree.

Remarks:

When a user initiates a drag from a Listview or a Tree, the onDragStart event handler is fired. If `owres` is not zero on return from onDragStart, and if sourceformats is not empty, dragging begins; otherwise the attempt to drag is ignored. This property describes the data formats the control is willing to provide to the object receiving the drop. The object receiving the drop may be the drag source, another APL+Win WI control, or a window in another application. See the description of Drag-and-Drop in the Programming the Windows Interface: Intermediate chapter in this manual.

state (read-only)

Description:

Nested matrix of properties and their values for all properties that have nondefault values for this object. This matrix does not contain the read-only properties, such as properties, opened or state itself.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, and Tree.

Remarks:

This property is a snapshot of the specification of an object. When you reference it, you see all the properties that have nondefault values. You cannot change this property directly, but the system updates it every time you change a property setting for an object.

status

Description:
Seven-column nested matrix that contains all the display information for the status bar.

Class: Status.

Remarks:
This property has one row for each pane; you can use the columns of this property in different ways:

Column 1: The first column is the non-zero integer number of the pane. Overlay panes have negative numbers; standard panes have positive numbers. If you use zero, the system assigns a unique positive number. You must have a pane numbered 1; this is the primary standard pane. It does not, however, have to appear first on the status bar. The panes appear in the order in which they exist in the matrix; if you have multiple overlay panes defined with text, the system displays the first one in the matrix.

Columns 2/3: The second and third columns determine the text and the width of the pane. You have two methods for using this pair of columns.

- Flag method (Standard panes only):** This method requires that you place text in the third column. The second column is a flag. If you set the flag to 1, the text in column 3 is displayed. If you set the second column to zero, the text is removed, but the blank pane remains. If you want to remove the pane, you must remove the text from column 3. You can use this method most effectively if you want to toggle information on and off without changing text or pane attributes frequently.
- Text method:** The second column contains the text you want to display. You always use this method for overlay panes; since they take up the entire status bar, the system ignores column 3. For standard panes, the third column determines the minimum width for the pane. If the text in column 2 is longer than the minimum, the system expands the pane to hold the text. If you specify a number, the system uses this as a minimum in current scaling units. If you specify a character vector, the system calculates the space it would require to hold that vector and treats it as the minimum value. You can use this method effectively if you want to change the text in standard panes frequently, and you want to avoid the flickering that would result from changing text lengths. Simply assign the longest message to column 3.

Column 4: The fourth column determines the border for each individual pane. (The border property determines the border for the status bar itself.)

Values for the Borders of Status Bar Panes (status Property Column 4)

Value	Description
Primary Value: Choose only one value for each pane	
0	No border.
1	Inset border.
2	Outset border
Add-on Values: Choose any one of the values for a given pane.	
8	Expand width of pane fully. (See note 1.)
16	Show Caps Lock state. (See note 2.)
32	Show Num Lock state. (See note 2.)
64	Show Scroll lock state. (See note 2.)

Note 1: For standard panes, if you add 8 to the border value, the system expands the width of the pane so that it takes all the available space on the status bar; that is, the space not actually required for other visible panes. If you specify this value for more than one pane, the excess space is divided evenly among these panes. This is the default value for the main pane only.

Note 2. You must use the **Flag** method for these panes. Define a standard pane; place a zero in column 2, your text, for example 'CAPS', in column 3; and the add-on value in column 4. When the user presses the Caps Lock key, it toggles the value in column 2 and the text in the pane on your status control. You can have only one of these in a pane. For example, to have the Nums Lock indicator: 'fmMain.stBar' □wi 'SetStatus' 1 0 'NUM' 33

Columns 5/6: The fifth column is the origin 1 index to the image in the imagelist specified in the imagelist property that you want to display at the left of the pane when the pane is visible. The sixth column is the index into the overlays property of the same imagelist for an overlay image. You can specify zero in either column for no image.

Column 7: The seventh column is initially an empty character vector; you can assign text for a tooltip that is specific to the pane. This pane-specific tooltip displays, whether or not the focus is on the form, when the user moves the cursor over the pane. If any visible pane has content in column 7, it disables the tooltip property for the control.

The default value for the **status** property is a two-row, seven-column matrix; it assumes the main standard pane and one overlay pane. The default value is shown below. Columns 2 and 7 contain empty character vectors.

```
1  ''  0  8  0  0  ''  
-1 ''  0  0  0  0  ''
```

When you assign the **status** property, you should allow for all the panes you will want to use. You can control the display of information by modifying the rows using the **SetStatus** method. If you want to add rows to the matrix, you must reassign the **status** property.

Note: Additional columns may be added in a future release, so your application should be prepared to gracefully accept a result of more than seven columns if you reference this property.

style

Description:

Integer vector or scalar that defines the type of object within its class. This property has a different meaning for each class.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Toolbox, Trackbar and Tree.

Remarks:

The **style** property is an integer vector or scalar that contains values from the following lists. **style** can be an integer singleton that is the sum of the selected values. The initial value of **style** for any newly created control object is 0. See the descriptions of each class for greater detail. The tables below show the possible values for the **style** property for different classes. You can choose any one of the primary values, which are listed first for each class. You can add values or combinations of values from the add-on values.

Values for the Button Class style Property

Value	Description
Primary Value: Choose only one value	
0	Draw button in style of operating system. Automatically justify text, and align image to left of text.
1	Default Button for entire form.
2	Cancel button (closes form if no onClick handler exists for this button).
Add-on Values: Choose none, any or all of these values	
4	Button does not get the focus. (Cannot be used with 1 or 2)
8	Button toggles between normal and selected. (Cannot be used with 1 or 2)
16	Do not grey image when button is disabled.
32	Use foreground color of color property for text; ignore background color.
Drawing Style: Choose one of these values	
64	Force Windows 3.1-style buttons. (Obsolete)
128	Force Windows 95 style button. (Obsolete)
Justification Values: Choose one of these values	
256	Left justify text and image.
512	Right justify text and image.
768	Center justify text and image.
Alignment Values: Choose one of these values	
4096	Align image to right of text.
8192	Align image above text.
12288	Align image below text.

Values for the Check Class style Property

Value	Description
0	Check box on left of text.
1	Check box on far right; text left-justified.
Add-on Value: Choose with either primary value.	
2	Three-state check box; unchecked, checked, indeterminate.

Values for the Combo Class style Property

Value	Description
Primary Value: Choose only one value	
0	Drop-down list.
1	Simple (always down).
2	Drop-down list with display field.
Add-on Values: Choose none, any, or all of the values	
8	Inhibit automatic scrolling in the edit field.
16	Vertical scroll bar in list.
32	Allow the display of a vertical fraction of a line.
128	Hide scroll bar if not necessary (short list).
256	Update edit window before running the onClick event handler.
2048	Allow APL characters from keyboard in edit portion of combo.
4096	Extended Combo control (images and indents).

Values for the CommandBar Class style Property

Value	Description
Primary Value: Choose only one value	
0	Default command bar.
1	Place button captions at side of image instead of below.
2	Fixed-position corner display bar.
Add-on Values: Choose these values with appropriate restrictions	
4	Reduce space surrounding image or control.
8	Remove Gripper from bar.
16	Fixed wrap positions.
32	No caption.
64	Fixed height bar.
128	Floating bar cannot be closed by menu or close button.
256	Do not show menu or close button on floating bar.
512	Command buttons look like normal buttons; raised, no hot images.

Values for the CommandButton Class style Property

Value	Description
0	Normal button.
1	Separator.
2	Toggle button.
4	Part of a group of radio buttons; must be used with 2.
8	Drop-down button.
16	Automatically resize button to fit text.
32	Don't underline the character following an ampersand (&) in caption.

Values for the DateTime Class style Property

Value	Description
Primary Value: Choose only one value	
0	Calendar control.
1	Date Selector with short date format.
2	Date Selector with long date format.
3	Time Selector.
Add-On Values: Choose any of the values with the appropriate Primary value	
4	Allow a range of consecutive dates to be picked for calendar control.
8	Display week (of the year) number to the left of the calendar.
16	Provide up/down buttons instead of the drop-down calendar.
32	Provide check box in date/time field to disable entry and allow blank for value.
64	Do not circle today's date on calendar.
128	Do not display today's date below calendar.
256	Right align the drop down calendar.
512	Clicking calendar with mouse does not move focus to DateTime control.

Values for the Edit Class style Property

Value	Description
0	One-line; left-justified text (the default).
1	Centered text (only if 4 is also coded on some systems).
2	Right justified text (only if 4 is also coded on some systems).
4	Multiline.
8	Inhibit automatic vertical scrolling (only useful if 4 also coded).
16	Vertical scroll bar (only useful if 4 also coded).
32	Inhibit automatic horizontal scrolling (causes word wrapping with 4).
64	Horizontal scroll bar.
128	Password (displays chars as *).
256	Lowercase (cannot be used with 512).
512	Uppercase (cannot be used with 256).
1024	Keep selection highlighted when focus moves away from field.
2048	APL entry field (you must also set the font property).
4096	Read only.
8192	Allow Enter character (only if 4 is also coded).
16384	Do not select text when tabbing back to control. (Often used with 4 and 1024.)
524288	Read only with the Paste menu in the context menu disabled (only with 4096).

Values for the Form Class style Property

Value	Description
1	Signal Wait event before Showing form.
2	Independent of any other form that is Waiting.
4	Hide form before Waiting.
8	Do not disable for Wait on a secondary form.
16	Always on top. (Ineffective for an MDI form.)
32	Force icon in taskbar even if form is a tool palette (border 512).

Values for the Frame Class style Property

Value	Visual Effect without 3D	Visual Effect with 3D
0	Frame appears as black line; see note.	Frame appears etched into surface; see note.
1	Black line rectangular frame	Inset 3D rectangle
2	Gray line rectangular frame	Flush 3D (etched) rectangle
3	White line rectangular frame	Outset 3D rectangle
4	Solid black rectangle	Inset 3D rectangle
5	Solid gray rectangle	Flush 3D (etched) rectangle
6	Solid white rectangle	Outset 3D rectangle
7	Transparent	Transparent

Note: style 0 is the standard frame with a caption. Other styles do not display a caption.

Values for the Imagelist Class style Property

Value	Description
0	Masked imagelist (allows one color value to be transparent pixels).
1	Non-masked imagelist.

Values for the Label Class style Property

Value	Description
Primary Value: Choose only one value	
0	Left justified.
1	Centered.
2	Right justified.
Add-on Values: Choose none, any, or all of the values	
4	Inhibit the effect of an access key (do not underline character following &).
8	Inhibit word wrap (only with 0).
32	Display ellipsis at the end of the visible caption; not with 64.
64	Display ellipsis in the middle of the visible caption that has the form of a path; not with 32.

Values for the List Class style Property

Value	Description
Primary Value: Choose only one value	
0	Single selection.
1	Multiple selections.
Add-on Values: Choose none, any, or all of the values	
2	Allow the display of a vertical fraction of a line.
16	Vertical scroll bar.
32	Multiple columns (must also set limit).
64	Horizontal scroll bar (must also set limit).
128	Hide scroll bar if not necessary (short list).
256	Select multiple items independently (only with 1).

Values for the Listview Class style Property

Value	Description: Choose none, any, or all of the values
1	Highlight entire row; not just image and label.
2	Stop highlighting at end of last column (only with 1).
4	Highlight selected items even when control loses focus.
8	Disable multiple selections; single selection only.
16	Re-arrange images automatically in both image views.
32	Do not wrap long labels in largeimage view.
64	Disable scrolling. (This setting is ignored in report view.)
128	Do not show headers in report view.
256	Disable column header buttons and onColClick in report view (not with 128).

Value	Description: Choose none, any, or all of the values
512	Enable editing of column one text.
1024	Enable user to reorder column headers in report view.
2048	Enable limited automatic drag-and-drop.
4096	Display gridlines between cells in report view.
8192	Item is selected if cursor hovers over it.
16384	Trigger onActivate when item is clicked.
32768	Trigger onActivate on second click.
65536	Enable automatic check boxes as user state images.

Values for the MDIForm Class style Property

Value	Description
1	Signal Wait event before Showing form.
2	Independent of any other form that is Waiting.
4	Hide form before Waiting.
8	Do not disable for Wait on a secondary form.
16	Automatic scroll bars when child forms extend outside the visible client area.

Values for the Media Class style Property

Value	Description
Primary Value: Choose only one value	
0	Position the child control window in the upper left corner of the Media window.
1	Resize the Media window to match the size of the child control window.
2	Center the child control window in the Media window.
3	For an MCI player, stretch the child control window to fill the Media window. For other child controls, this is the same as value 0.
Add-on Values: Choose none, any or all of these values	
4	Draw the file transparently.
8	Do not automatically play the file when it is opened.
16	For an MCI player, include a toolbar in the child control window.
32	For an MCI player, display a context menu in response to a right mouse click in the child window; if used with 16, place a menu button on the toolbar.
64	For an MCI player with menu, do not include an Open menu item. Use with 32.
128	For an MCI player, allow recording. Use with 32.
256	For an MCI player, inhibit the error message box.

Values for the Menu Class style Property

Value	Description
0	Visible check by program only.
1	User toggles visible check.
2	Window menu; automatically shows children of an MDI form.
4	Radio style; only one menu of a radio group can be selected at a time.
8	Menu item appears in bold; (only one per group of siblings).

Values for the Option Class style Property

Value	Description
0	Radio button on left of text.
1	Radio button on right of text.

Values for the Picture Class style Property

Value	Description
Primary Values: Choose only one value	
0	Initial bitmap is unscaled and centered in the image.
2	Initial bitmap is tiled to fill the image.
4	Initial bitmap is scaled to fit the image.
Add-on Values: Choose none, one or both values	
16	Vertical scroll bar.
64	Horizontal scroll bar.

The default value of zero assumes that the image property is larger than the bitmap. If the bitmap is larger than image, and you do not increase the size of image, the image contains only the upper left portion of the bitmap. In this case, the rest of the bitmap is not accessible even with scrolling.

Values for the Progress Class style Property

Value	Description
0	Default bar; oriented horizontally, filled with blocks.
1	Smooth fill. The "fill" color that shows the progress is solid.
2	Vertical bar; filled with blocks; progress starts from the bottom.
3	Vertical bar with solid fill coloring like a thermometer.

Values for the RichEdit Class style Property

Value	Description
0	One-line; left-justified text (the default).
4	Multiline.
8	Inhibit automatic vertical scrolling (only with 4).
16	Vertical scroll bar (only if 4 also coded).
32	Inhibit automatic horizontal scrolling (causes word wrapping with 4).
64	Horizontal scroll bar.

(continued)

Values for the RichEdit Class style Property (cont.)

Value	Description
1024	Keep selection highlighted when focus moves away from field.
2048	APL entry field.
4096	Read only.
8192	Allow Enter character (only if 4 is also coded).
16384	Do not select text when tabbing back to control. (Often used with 4 and 1024.)
32768	Disable automatic word selection when highlighting with mouse (that is, allows partial words at the beginning or end of mouse drag).
65536	Disable, rather than remove, unneeded scroll bars (with 16 and/or 64).
131072	Allow Tab character as part of the text.

Values for the Scroll Class style Property

Value	Description
Primary Value: Choose only one value	
0	Vertical.
1	Horizontal.
2	Size box (grayed area where scroll bars meet).
Add-on Values: Choose any of the values	
0	No alignment.
4	Bottom/right alignment.
8	Top/left alignment.

Values for the Selector Class style Property

Value	Description
Primary Value: Choose only one value	
0	Tabs.
1	Buttons.
2	NoTabs.
Add-On Values: Choose any of the values in meaningful combinations	
8	Multiline: Use multiple rows to display all tabs at once.
16	RaggedRight: Do not increase size of tabs to fill entire width of control (only with 8).
32	FixedWidth: Make all tabs the same width.
64	ForceImageLeft: Left justify the image, but leave the text centered (only with 32).
128	ForceLabelLeft: Left justify the image and the label (only with 32).
256	Limit focus: Tabs accept focus from tab key, but not from mouse click.
512	Immediate focus: Tabs accept focus on first mouse click; (not with 256).
1024	Tabs on side (left): Tabs are oriented vertically on left edge with caption running up.
2048	Tabs on bottom or right: Tabs on bottom of selector or, with 1024, on right edge.
4096	Keep lines intact: Forces multiline display; keeps lines in the same order.
16384	Flat buttons: Gives the flat button appearance, like command buttons (only with 1).
32768	No separators: Do not put separators between flat buttons (only with 1 and 16384).
65536	Hot track: Change tab appearance when mouse cursor is moved over it.

Values for the Spinner Class style Property

Value	Description
Primary Value: Choose only one value	
0	Vertical spinner.
1	Horizontal spinner.
Alignment Value: Choose only one value.	
0	Right alignment with buddy control.
4	Left alignment with buddy control.
8	No alignment.
Add-On Values: Choose any of the values with the appropriate buddy class	
16	Update buddy control automatically.
32	Use thousands separator (only with 16). System setting; in US, commas.
64	Keyboard arrow keys do not change spinner.

Values for the Toolbox Class style Property

Value	Description
Primary Value: Choose only one value	
0	Items in list property refer to icons.
1	Items in list property refer to bitmaps.
Add-on Values: Choose any of the values.	
0	No overlap or spacing between icons or bitmaps.
16	One pixel spacing between icons or bitmaps.
32	One pixel overlap of icons or bitmaps.

Values for the Trackbar Class style Property

Value	Description
Primary Value: Choose only one value	
0	Vertical.
1	Horizontal.
Tick Placement Value: Choose only one value	
0	Ticks on bottom or right side.
2	Ticks on top or left side.
4	Ticks on both sides.
Add-on Values: Choose none, any, or all of the values	
8	Hide tick marks but display slider.
16	Track slider movement. (See note.)
32	Enable selection range.

Note: By default, one Change event occurs when the user releases the mouse button to stop scrolling. If you add in 16, a user who drags the slider generates a string of Change events.

Values for the Tree Class style Property

Value	Description
0	Lines connecting children; buttons on parents below the top level (default value).
1	Does not have buttons.
2	Does not have lines connecting children.
4	Has buttons for top-level parents (if 1 is not included) and has lines connecting children to top-level nodes (if 2 is not included).
8	Always highlight selected node, even when focus is lost.
16	Enable editing of labels.
32	Enable automatic check boxes as user state images.

suppress

Description:

Three-state flag that indicates whether to suppress event handlers that would occur as the result of program action setting a property, but still allow the same event handlers to occur as a result of user action or invoking the event handler by using a method.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, and Tree.

Remarks:

The system automatically triggers certain event handlers when a program action sets a property of an object; it triggers the same event handlers as it does when a user action changes the value of that property. One example of this behavior is triggering the `onChange` event handler of an Edit control when the program sets the `text` property as well as when the user types a character. Another example is triggering the `onModified` event handler of the parent of an object that the user changes.

You can distinguish between the implicit trigger of the program setting a property and the explicit trigger of a user action or a method that invokes the event handler. You can suppress the implicit triggers by setting the `suppress` property of an object to 1. Setting this flag does not suppress event handlers triggered by user action or by a method.

Remarks:

The default value for this property is 0 for top-level objects, and -1 for children. The value -1 tells the system to take the same value for this object as its parent. The value 0 says do not suppress any event handlers. If you set the value to 1, the system does not trigger event handlers as a result of a program action that sets a property. It still triggers event handlers as a result of a user action or a program invoking a method.

For example, the system triggers an `onResize` event handler for a Form whenever the `size` property changes. If you set the `suppress` flag to 1, the system does execute the `onResize` event handler if the user drags a margin or if you invoke the `Resize` method, but does not trigger the handler when you set the size to a new value by using a program statement.

tabgroup

Description:

Boolean flag that indicates an object either is not a member of a group or is the first member of a group. If tabgroup is zero, the object is a member of a group. To determine the members of a group, place (figuratively) the objects in sequence by the value of their order property. Each object that has a tabgroup value of 1 followed by an object with a tabgroup of zero is the first member of a group. The group includes all those in immediate succession that have a tabgroup value of zero. If an object has a tabgroup value of 1 and its immediate successor also has a tabgroup of 1, the first object is not a member of a group.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Frame, Label, List, Listview, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

If you create a group, a user can move among members of the group by using the arrow keys. By default, the system makes a group of option buttons or check boxes whose order properties are in sequence. You can make any object a member of a group; however, if you make an edit control or a combo box part of a group, a user can arrow to it, but cannot arrow out of it. The arrow keys have a different behavior within an edit, combo, or list field.

Warning: This flag is not persistent; it is not saved in the def or the state property. You must redefine it each time the object is opened.

tabparent

Description:

Boolean flag that indicates there is a child of this object to which the user can tab.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Frame, Label, List, Listview, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

If you set the tabparent property of an object, for example a frame, to 1, a user can tab from outside the frame to a child of the frame that has a tab stop and from there to the object that is next in the taborder. If the frame's tabparent value is zero, a user cannot tab to the object from outside the frame, but can tab from an object inside the frame to one outside the frame.

tabrows (read-only)

Description:

Scalar integer indicating the number of rows of tabs or buttons in the selector.

Class: Selector.

Remarks:

This property is useful only when Multiline style (8) is selected. If there are no pages, or if NoTabs style (2) is selected, tabrows is 0; otherwise, if multiline style is not selected, tabrows is 1.

tabstop

Description:

Boolean flag that specifies whether a user can tab to this object.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Frame, Label, List, Listview, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

By default, objects that can get the focus to allow user input have `tabstop` 1, except that only the first of a group of check boxes or the selected option of a group of options has a tab stop. You can set this property to zero to remove an object from the tabbing sequence; the value of its `order` property does not change.

You can also set `tabstop` to 1 for an object that cannot get the focus, such as a label. When you do this, the system inserts the object in the tabbing sequence and a user must tab twice to get past it. The focus property of the system object will report that this object has the focus when a user tabs to it; however, there is no visual indication that the object has the focus and there are no events or behaviors that allow a user to do anything with the object.

You can set multiple check boxes in one group as tab stops without changing their being part of a group through which a user can cycle using the arrow keys. You can also set an option button that is not selected as a tab stop. However, that setting is transitory, because the system resets the `tabstop` property of each option in the group, just as it does the `value` property, each time a new option is selected.

Warning: This flag is not persistent; it is not saved in the `def` or the `state` property. You must redefine it each time the object is opened.

targetformats

Description:

Zero or more character vectors that specify the format for dragging items to this control. The valid values are 'FileName', 'FileDrop', 'Text', 'ListviewItem', and 'TreeItem'.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

When an object is being dragged and the cursor moves into the window representing an APL control with a non-empty `targetformats` property, the control queries the drag source to determine its source formats. If none of the target formats is available from the source, no drop target events, such as `DragEnter` and `DragOver`, are triggered for the control. If a source format matches a target format, the drop events are fired. If the item is dropped on the control, the first format in `targetformats` that is available from the drag source is used as the format for data passed to the control in the `onDrop` event handler. See the description of Drag-and-Drop in the Programming the Windows Interface: Intermediate chapter in this manual

text

Description:

Character vector that holds the text displayed in an Edit or RichEdit control, or in the edit box of a Combo or a DateTime control that is not the visible calendar. This property is read-only for a DateTime control.

Classes: Combo, DateTime, Edit, and RichEdit.

Remarks:

The text property specifies the text that appears in the edit window of an Edit or Combo control. The system changes this value whenever a user types characters in the window. Setting the text property using the `PutText` function triggers the `onChange` event handler for either Edit class, unless the `suppress` property is set to 1. Setting the text property also triggers the `onChange` event handler for a Combo object unless the `suppress` property is set to 1 or its `style` property is set to 2 (drop-down list with display field). In that case, it triggers the `onClick` event handler. A style 2 Combo never gets the `Change` event, since there is no text entry.

The text property of a RichEdit control contains only the raw text. Its `rtf` property contains the text and the control codes that determine the formatting. You cannot set the text property of a RichEdit control when the object is closed; you can set its `rtf` property.

For a DateTime control with a drop-down calendar, a time picker, or one with a customized display, this property holds the contents of the edit box, which you can reference but not set. You can set the `value` property of the control to change this indirectly. If the control is the visible calendar, this property is an empty vector; if the object is closed, the system signals an error if you attempt to reference it.

theme

Description:

The theme property controls the visual styling theme used for drawing the control. The default value is an empty character vector (' '), which uses the new visual styling for the control in Windows XP and later. To suppress the new visual styling for the control, set the theme property to "~".

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

themeactive (read-only)

Description:

Returns the boolean value of the Microsoft `IsThemeActive()` API.

Class: # (the system object).

Remarks:

A 1 means a visual style for the current application is active.

themed (read-only)

Description:

Returns the boolean value of the Microsoft `IsAppThemed()` API.

Class: # (the system object).

Remarks:

A 1 means that a visual style can be turned off in the Windows Control Panel, so that an application can support visual styles but not have a visual style applied at a given time.

thememode

Description:

Controls how visual styles are applied to the application windows. This property corresponds to the Microsoft GetThemeAppProperty and SetThemeAppProperty APIs. The default value is 3 for Windows XP and later.

Class: # (the system object).

Remarks:

The value of the thememode property is the sum of the following values or zero:

Values for the System Class thememode Property

Value	Description
1	Specifies that the non-client areas of your application windows have visual styles applied. Turning off this value will give you the Windows classic look in the borders of your top-level windows. (See note 1.)
2	Specifies that controls in your application have visual styles applied.
4	Specifies that all web content displayed in an application is rendered using visual styles.

Note 1: Setting this value to 1 gives you the visual styling of the non-client window frames around the forms but the non-visual styling for the child controls created on the forms after having set the property. This is more or less the same visual effect you have in past versions of APL+Win.

The visual styling used for a window depends on the value of this property that was in effect at the time the window was created. After a window is created, changing this property does not affect how it is displayed.

tickinterval

Description:

Scalar that specifies the spacing of the tick marks along the range between the minimum and maximum values in a Trackbar object. The default value is 0; which means only the tick marks at the minimum and maximum display.

Class: Trackbar.

Remarks:

By default, the system places tick marks at the minimum and maximum unless you specify 8 in the style property to hide all ticks. You can specify any positive value up to the length defined by the range property. If you specify 1, the system sets a tick for each integer value. If you specify 2, the system places a tick at every second value, and so on.

If you want to assign the tick mark positions yourself, set this value to zero. The default ticks at the minimum and maximum appear. You can assign the tick positions you want the trackbar to display between

the minimum and maximum in the ticks property. Whenever you set the tickinterval property, the system resets the ticks property to an empty vector.

tickpos (read-only)

Description:

Two-column matrix that contains the list of the currently displayed tick marks for a trackbar. The first column contains the integer values corresponding to each tick mark. The second column contains the physical position in the currently selected scaling units.

Class: Trackbar.

Remarks:

This matrix includes a row each for the ticks that mark the minimum and maximum values of the range. If you specify the ticks manually, this matrix is in the order you specify, not necessarily in ascending order.

ticks

Description:

Integer vector that contains the tick mark list for a trackbar if you set the ticks explicitly; otherwise, it is empty.

Class: Trackbar.

Remarks:

If you set the ticks property, the system resets tickinterval to zero. If you specify ticks outside the range, they do not display.

today

Description:

For the visible calendar this property is a three-element vector showing the day marked as "today" on the calendar. For the other styles, this property is initially empty; you can set it to a three-element numeric date vector.

Class: DateTime.

Remarks:

By default, the date considered "today" is shown at the bottom of the calendar and circled in red when its month is visible.

tooltip

Description:

Character vector that holds the text you want to display in the popup window when the mouse arrow hovers over the object.

Classes:

Button, Check, Combo, CommandButton, DateTime, Edit, Frame, Label, List, Listview, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

Setting the tooltip property to a character vector causes the tooltip to appear automatically, as long as tooltips are enabled. Setting the tooltip property to an empty vector clears the tooltip, so no tooltip window appears for the control.

For a Toolbox, if you specify a simple character vector, the same tooltip appears for any cell. You can also specify the tooltip as a nested array or as a character matrix. In this case, the first row of the matrix or the first vector in the nested array appears for the first tool button, the second node for the second tool button, and so on. If you have fewer rows or nodes than cells, nothing appears for the last tool buttons.

For a Page object, the tooltip appears only when the mouse is over the tab or button that is used to select the page.

For a Status object, the tooltip is suppressed when any visible pane has a pane-specific tooltip; that is, the seventh column of the status property is not empty. For a Listview object, tooltips may be preceded or preempted by label tips (if the first column has more text than will display) or info tips (item-specific text).

tooltipenabled

Description:

Boolean flag that specifies whether tooltips are enabled for the application or session.

Class: Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

By default, tooltips are enabled. You can disable them by setting this property to zero. You can suppress the tooltip for an individual control by setting its tooltip property to an empty vector (' ').

You could allow the user to control whether tooltips display by toggling a menu node that sets the tooltipenabled property.

tooltipstyle

Description:

Integer that specifies the behavior of a tooltip set for a control.

Class: Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:**Values for the tooltipstyle Property**

Value	Description
0	No special behavior (default).
1	Always display tooltip when mouse is over the control regardless if the form not activated.
2	Suppress special handling of ampersand (&) in the tooltip property; without this value a single ampersand will appear as an underscore in the tooltip.
64	Display tooltip in a balloon style.

Note: This value can be overridden by setting [Config]ToolTipStyle=*value* in the APLW.INI file.

tooltiptime

Description:

A zero- to two-element vector that specifies the display time and delay time of tooltips. When you set the value, trailing elements can be omitted without changing their values. The two elements are:

[1] Autopop delay time (milliseconds)

The amount of time the tooltip window remains visible if the pointer is stationary within a control's bounding rectangle. Use -1 to inherit.

[2] Initial delay time (milliseconds)

The amount of time the pointer must remain stationary within a control's bounding rectangle before the tooltip window appears. Use -1 to inherit.

Class: Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

When setting the property any elided trailing elements act as if specified as -1.

When referencing the property any -1 trailing elements are elided. Therefore an empty vector is returned for any object on which none of these times are set.

Note: All objects inherit from their parent, grandparent, etc, back to the system object. If the delay times are not set for the system object they use defaults based on the double-click time unless overridden by the following INI file settings:

[Config]ToolTipTimePop
[Config]ToolTipTimeInit

If not set the defaults for non-unfold times are based on the system's double-click time as follows:

ToolTipTimePop delay time = doubleclicktime × 10
ToolTipTimeInit delay time = doubleclicktime

If system double-click time is 500 ms, the autopop and initial delay times are 500 ms and 5000 ms, respectively.

The maximum delay time settings for any element of tooltip time is 32767 seconds.

tooltipwidth

Description:

Numeric singleton that specifies the width of the popup window in which the tooltip appears.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

By default, the popup window adjusts to the length of the tooltip, and it can extend across the entire screen. You can specify the maximum width in character units using this property. The window then adjusts vertically. However, note that a tooltip does not break words. If the longest word in the text of the tooltip is larger than the tooltip width, the popup window adjusts to the length of that word.

An object inherits this property from its parent if you do not explicitly set it. Thus, you can, for example, set a value for the system object that applies to every object in your application.

translate

Description:

Integer that specifies the translation of text characters between APL and Windows.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Selector, Status, Trackbar, and Tree.

Remarks:

The translate property controls translation between APL's character set (⎱av) and the ANSI character set that is normal for Windows. The table below shows the possible values.

Values for the translate Property

Value	Description
0	No translation
1	Translate characters between APL and ANSI to preserve meaning.
2	Do not add line feeds; do not translate characters.

Value	Description
3	Do not add line feeds; translate text characters to preserve meaning.
6	Add and remove line feeds; do not translate other characters
7	Add and remove line feeds; translate other characters between APL and ANSI.

APL and ANSI contain different sets of characters and have different positions for some of the same characters. For example, the É (capital E with acute accent) is `⎕av[⎕io+144]`, but is at position 201 in ANSI. If `translate` is set to 1, the system automatically translates `⎕av[⎕io+144]` into ANSI 201 so it appears as an É to Windows, and vice versa. If you set `translate` to zero, you must use `⎕av[⎕io+201]` to produce É. The system performs translation on setting and referencing any text-valued property, in data transferred via the clipboard or DDE, and the `Text` operation of the `Draw` method. Because the ANSI character set contains characters that are not in `⎕av`, and vice versa, several `⎕av` characters may map to the same ANSI character, and vice versa. The characters that were added to `⎕av` in version 1.2 of this product satisfy most needs users have with European alphabets. This renders the `translate` property zero setting unnecessary in most instances.

There are additional complications when text has multiple lines, for instance in a `Edit` control. APL traditionally represented a new line (hard break) by a single character, `⎕tcn1`, where Windows uses two characters, a carriage return and a line feed. The APL Windows interface attempted to adjust for this difference by adding line feeds when APL sent a `⎕tcn1` character to Windows and removing `⎕tc1f` characters when receiving data from Windows; however, this has caused anomalies, particularly with the `selection` property in an edit control that has automatic line wrapping (style 32).

APL+Win provides options to the `translate` property to ameliorate these problems. If you add 2 to the `translate` property, APL does not add or remove line feed characters. If you set `translate` to 2, no characters are translated. If the user inserts a line break (with style 8192, which allows the Enter key) both Windows and APL use two characters. You may have to adjust the `limit` property if you use this style. Similarly, if you use style 3, APL translates all text (displayable) characters, but does not add or subtract line feeds. The system calculates the `selection` property consistently, but you may have to adjust the `limit` property to account for two-character line breaks.

If you set the `text` property under program control with `translate` set to 2 or 3, you must put in both a `⎕tcn1` and a `⎕tc1f` for a new line.

If you add both 2 and 4 to the `translate` property, APL+Win adds the line feed character after a `⎕tcn1` when going from APL to Windows and removes it when returning from Windows to APL. Additionally, it handles the `selection` property properly in the presence of soft line breaks. With `translate` set to 6, the system does not translate displayable characters; with `translate` set to 7, the system does translate characters between APL and ANSI.

We recommend that you use one of the larger `translate` values for new code. Existing code will continue to run unchanged using `translate` 0 or 1. If you want to match the Windows style of using two characters for hard line breaks, you can use `translate` 3 (or 2). If you want to continue to use only `⎕tcn1` for a hard break, you can use `translate` 7 (or 6). In particular, if you intend to make use of the `RichEdit` control, which depends heavily on the `selection` property, you should switch from the default `translate` value of 1 to `translate` 7.

undolimit

Description:

Integer scalar that specifies the maximum number of actions stored in the undo queue for a RichEdit control.

Class: RichEdit.

Remarks:

Setting this property to zero disables the undo facility. A version 1 RichEdit can undo only one action.

undoname (read-only)

Description:

Character vector, when the control is open, that indicates the type of action that can be undone; when the canundo property is zero, this property is an empty vector.

Class: RichEdit.

Remarks:

After an editing action in a RichEdit control has been taken, the action can usually be undone and the control returned to its previous state, by the user pressing the Ctrl+Z keystroke combination or the program invoking the Undo method. This property gives an indication of the type of action that is first on the Undo stack. Possible values include: `typing`, `delete`, `cut`, `paste`, `dragdrop`, `unknown` or an empty vector. The value is 'unknown' for character and paragraph formatting operations; it is empty if nothing can be undone.

unicodebstr

Description:

Integer scalar that allows passing Unicode character arrays to and from BSTR (string) arguments of ActiveX controls and objects.

Class: ActiveControl and ActiveObject.

Remarks:

In order to allow passing Unicode character arrays as arguments to ActiveX controls with `Unicode`, a new property has been added to all ActiveX controls. If the `unicodebstr` property is set to 1, any BSTR argument passed to the control can be a Unicode character array, and any BSTR argument returned from the control might be a Unicode character array. Also, the translation used for eight-bit characters will be different. Eight-bit characters will not be translated using the underlying Windows ANSI-to-Unicode API. Instead, they will be translated from `Char` to Unicode (or Unicode to `Char`).

As an example, `Char[io + 234]`, which represents `nand` (★) in APL, is sent as Unicode U+2372 (the Unicode code point for APL `nand`) if `unicodebstr` is 1, but as Unicode U+00EA (the unicode code point for lowercase e circumflex, ê) if `unicodebstr` is 0. Likewise, if an ActiveX control sends a Unicode U+00EA and `unicodebstr` is 0, it will be received in APL as `Char[io + 234]`, but if `unicodebstr` is 1 it will be received in APL as either `Char[io + 136]` (the position in `Char` for ê) or `Unicode 234`.

units (read-only)

Description:

Six-by-two integer matrix that defines scaling units in terms of pixels per scaling unit.

Classes: Printer and # (the system object).

Remarks:

This is a read-only property. The rows of the matrix correspond to the *scalemode* element of the *scale* property. For the system object (and therefore applicable to all visible controls), the values are:

Row Designations for the units Property

Row	Description
1	Characters (1 sys-font high; 1 sys-font wide).
2	Dialog units (1/8 sys-font high; 1/4 sys-font wide); the scaling units used by external dialog box editors.
3	Points (72 points per logical inch).
4	Twips (1440 twips per logical inch).
5	Virtual pixels.
6	True pixels.

The first column of units is the height of scaling element in pixels and the second column is the width of the scaling element in pixels. You can use this property to convert from scaled units to pixels by multiplying by the appropriate row of units, and you can convert from pixels to scaled units by dividing by the units row.

A typical units matrix might look like the matrix below; the numbers may vary, depending on your hardware configuration.

16	8
2	2
1.333333333	1.333333333
0.066666666667	0.066666666667
1	1
1	1

You can calculate the size of the Windows screen in pixels as:

```
('#' □wi 'size') × 2ρ '#' □wi 'units'
```

For a Printer, this property works the same way as it does for the system object, but the values will differ because the number of pixels per unit is different for a printer than it is for the screen.

value

Description:

Integer scalar for all control object classes except List, Scroll, and DateTime, which are integer vectors;

Toolbox, which is an integer matrix; and Form, which can be an arbitrary array of any type and shape. This property designates the current value of the object, with different meanings for each class.

Classes:

Button, Check, Combo, CommandButton, DateTime, Form, List, Listview, MDIForm, Media, Menu, Option, Progress, Scroll, Selector, Spinner, Toolbox, Trackbar, and Tree.

Remarks:

The value property designates a current value or setting for an object. See the descriptions of each class for greater detail. The table below shows the descriptions of the value property for classes for which value is a scalar.

Values when the value Property is a Scalar

Class	Values
Button	Up = 0; Down= 1; Up/Down flash = -1. When set nonzero, causes a Click event. For a toggle button (style 8) setting value to zero also causes a Click event.
Check	Unchecked= 0; Checked= 1; Indeterminate=2 (for 3-state check box, style 2 only). When set, causes a Click event.
Combo	No current selection= 0; If >0 the index of the current choice in the drop-down list.
Command Button	For a toggle button: default/unselected=0; selected=1. When set, Windows provides built-in effects that interact with the various imagelists that are set; see the table below.
Media	Current location in file or source. Empty vector when file type is 'avi'.
Menu	Unchecked= 0; Checked= 1. When set, causes a Click event.
Option	Unchecked= 0; Checked= 1. When set nonzero, causes a Click event.
Selector	No current selection= 0; If >0 the origin 1 index of the currently selected page. This is equivalent to the order property of the Page.
Spinner	Any integer value, the current value of the spinner.
Trackbar	Any integer value, the current position of the slider.
Tree	No current selection= 0; If >0 the numeric identifier of the currently selected node.

For a command button, Windows provides some built-in effects for various values, including showing the button pressed down and different background colors or overlays. In addition, the image that appears may change, depending on which imagelists are set on the parent command bar. You can set combinations of the values below through 15 with varying effects. In particular, values 8 through 15 may depend on system settings as well as what images are set.

Values for the CommandButton value Property

Value	Description
0	Normal – flat button; normal image.
1	Selected – sunken appearance; image from imagelisthot.
2	Pressed – sunken appearance; normal image. (See note.)
4	Indeterminate – flat appearance but image may be masked, greyed, or invisible; image from imagelistdisabled, if set.
8	Marked – flat button, but image may be overlaid or otherwise accented.

Note: Although this is designated as pressed, it is not the appearance the button has when the user holds it down with the mouse. When the mouse is over the button, it has the hot image.

Classes for which value is a vector or matrix:

- **List**

Index of current choice or choices; empty if there are no currently selected items. For a multiple-selection List, this property is a 1-element vector if one choice is selected; a longer vector if more are selected. It is a scalar for a single-selection (style 0) list. Triggers a Click event when any value is set.

- **Listview**

An integer vector of the items that are currently selected.

- **Scroll**

A five-element vector that indicates a scroll state. When you set the value, trailing elements can be omitted without changing their values. The five elements are:

- Current position (default 0)
- Maximum position (default 100)
- Page size increment (default 1)
- Minimum position (default 0)
- Enable tracking of user moving scroll box (thumb) by sending Change events as it moves.

- **Progress**

A four-element integer vector that indicates the progress state; the elements must be non-negative and less than 65536. When you set value, trailing elements can be omitted without changing their values. However, the maximum cannot be less than the minimum. The four elements are:

- Current position (default 0)
- Maximum position (default 100)
- Step increment (default 10)
- Minimum position (default 0)

- **DateTime**

When you do not have style 4, allowing a range of dates, this property is a simple numeric vector. When you reference the value property, it is either a three-element or a seven-element numeric vector. If you are using the calendar style, it is a three-element vector showing the year, month, and date. If you are using either of the drop-down date styles or the time style, value is a seven-element vector, such as that provided by `DateTime`: year, month, date, hours, minutes, seconds, milliseconds.

When you set the value property for any style, you can use a seven-element vector. You can also use a six-element vector, ignoring the milliseconds. When you do this, the milliseconds value is set to zero. For any of the date styles, you can specify a three-element vector, designating year, month, and date. When you do this, the time elements are left unchanged.

For the time style only, you can specify a three- or four-element time, designating hour, minute, second and, optionally, millisecond. The date portion of the value property remains unchanged, either the last value provided by the program or the default current date.

When you have 4 included in the style, which is valid only with the style 0 calendar control, value is a two-element nested vector showing the first and last dates of the range selected. If only one date is selected, the two vectors are the same. If you have the multiple date style selected, you can set either one date or the first and last dates of a span. You cannot have a range of times for a style 3 control or a range of dates for styles 1 or 2.

- **Toolbox**

A matrix that contains one number for each cell in the Toolbox. value determines which bitmap or icon the system displays for each cell. By default, each element of value is 0, which causes the system to display the first icon or bitmap for each cell. Setting the value causes the system to display the corresponding bitmap or icon in the list property.

- **Form and MDIForm**

This property usually has no meaning for Form; you can assign an arbitrary value. On exit from the Wait method, `Wait` returns value as an explicit result.

version (read-only)

Description:

Character vector that holds the version of the system or an ActiveX object.

Classes: ActiveControl, ActiveObject, and # (the system object).

Remarks:

The version property of the system object provides the version number, date, time, and file name of the DLL that contains the APL Windows interface facility. It complements the system function `sysver`, which returns the version number of the APL interpreter. The version property of an ActiveX object holds the ActiveX version of that control.

viewalign

Description:

Character vector containing one of the three alignments for largeimage or smallimage view in a Listview control: 'none' 'top' or 'left'. The default is no alignment.

Class: Listview.

Remarks:

In largeimage and smallimage views, if you assign 'top', Windows aligns the first images with the top of the client area of the control and arranges them in columns; if there are more than will fit, the user must scroll vertically. In largeimage and smallimage views, if you assign 'left', Windows aligns the first images with the left side of the client area of the control and arranges them in rows; if there are more than will fit, the user must scroll horizontally.

viewmode

Description:

Character vector containing one of the four views: 'largeimage', 'smallimage', 'list', or 'report'. The default value is 'report'.

Class: Listview.

visible

Description:

Flag that controls whether an object is hidden, visible, maximized, or minimized.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

The values the visible property vary for different classes:

- Menu Class
 - 0 Hidden
 - 1 Displayed on the menu bar or a menu
- Control Object Classes: (other than Menu)
 - 1 Hidden
 - 1 Shown normally
- Form and MDIForm
 - 3 Maximized but hidden
 - 2 Minimized and hidden
 - 1 Hidden
 - 1 Shown normally (restored)
 - 2 Minimized (reduced to an icon)
 - 3 Maximized

For a form, you can set this property value directly with the `❏wi` function; however, you usually change this value indirectly by using the `Hide`, `Open`, `Show`, and `Wait` methods. The user can cause a change to visible through interaction with the Form, for example by minimizing or maximizing it.

The `Open`, `Show` and `Wait` methods cause the system to change the value of this property from 0 to 1 and from a negative number to its absolute value. The `Hide` method leaves zero and negative numbers unchanged and negates a positive number. The only way for the program to maximize or minimize a form is for you to set the visible property value:

```
'formname.objectname' ❏wi 'visible' visiblevalue
```

Because of a restriction that Windows imposes, when the program hides a form, you cannot change its minimized/normal/maximized state. When you make a form visible, however, you can change it from one state to another. If you have a maximized form (`visible = 3`), and you assign zero or any negative number to the visible property, the value of visible becomes `-3`. When the form is hidden, you can assign any of 1, 2, or 3 when you make it visible again. Note also that you cannot change the position or size (its `where`, `size` or `extent` properties) of an object that is minimized or maximized; however you can change these values effectively by setting the `place` property.

where

Description:

Two- or four-element vector that specifies the position and, optionally, the size of an object. Position is the row and column of the upper left corner of the object relative to the client area of the parent, and size is the height and width, all in the scaling units of the parent object. This property is read-only for `Page`, `Status`, and `CommandButton` objects. For a `CommandBar`, you should attempt to set this property only if it is set to be a client of its container (`dock` property `-1`). To position a floating command bar, you can use the `floatwhere` property. If you want to position a docked bar, use the `dock` and `order` properties, augmented by the previously set `docklength`, if necessary.

Classes:

`Button`, `Check`, `Combo`, `CommandBar`, `CommandButton`, `DateTime`, `Edit`, `Form`, `Frame`, `Label`, `List`, `Listview`, `MDIForm`, `Media`, `Option`, `Page`, `Picture`, `Progress`, `RichEdit`, `Scroll`, `Selector`, `Spinner`, `Status`, `Toolbox`, `Trackbar`, and `Tree`.

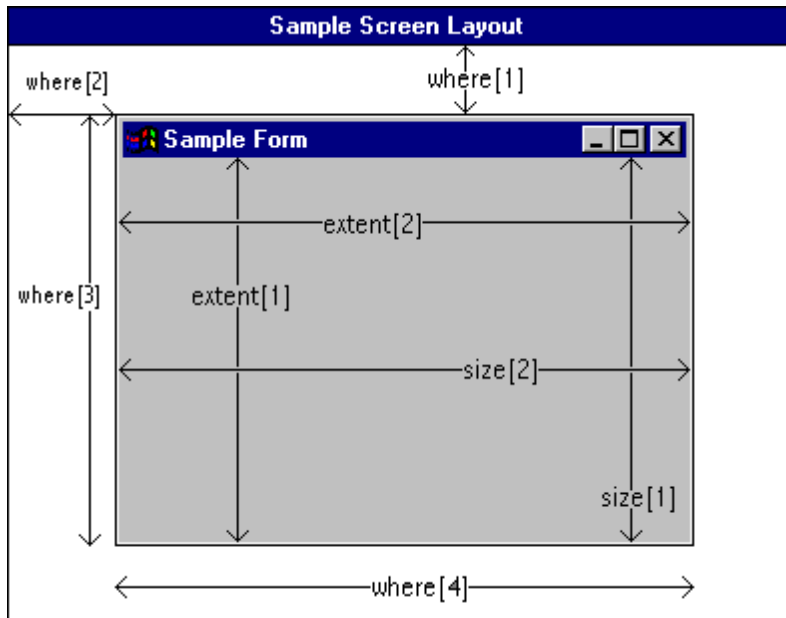
Remarks:

The `where` property and the `extent` and `size` properties are related for all control object classes; if you set one property, it causes an automatic change in the other. `where` specifies the position and size of an object's

outer dimensions. `size` and `extent` specify the size of an object's client area or inside dimension (that is, inside the border, title bar, and menu bar; see the figure below).

For controls without a border, the `size` and `extent` properties indicate the same area as the third and fourth elements of `where`. If you add items to a control, such as a scroll bar in an edit control or a status bar on a form, the client area is reduced.

The extent, size, and where properties



The `size` and `extent` properties can specify the client area in different units, depending on the value of the `scale` property. The `extent` property always uses the same units as `where`, which are the scaling units of the object that contains this object. The `size` property uses the scaling units defined for this object.

If the object is open and you change the value of one of these three properties, the system automatically updates the other two. When the object is closed, if you specify `where` as a two-element vector, it leaves the current size of the object unchanged. If you specify `where` as a four-element vector, it resets `size` and `extent` to be empty vectors. If you specify `size` or `extent` as a two-element vector, the system displays only the first two elements of the `where` property, the coordinates of its location. When you reopen the object, the properties are synchronized and as described above.

width

Description:

Integer value that represents the `width` of a command button in the parent command bar's scale units (since `CommandButton` does not have a `scale` property). The default value is zero, which causes the button to take the width of the images from the imagelist named in the bar's `imagelist` property.

Class: `CommandButton`.

Remarks:

The height of the button is the largest of the three image sizes from the command bar's imagelists, if all three

are defined. A command button does not have a `size` property; its `where` property is read only. Captions on a command button can affect height or width.

workarea (read-only)

Description:

Four-element numeric vector, like a `where` property, showing the location and size of the screen's work area, which is the area excluding an always-on-top taskbar or other application bar that docks and is not using the autohide feature. The property is scaled in character units and its origin is relative to the upper left corner of the screen.

Class: `#` (the system object).

Remarks:

When the user has a taskbar, the effective size of the screen is more accurately represented by the third and fourth elements of `workarea` than the `size` property of the system object. And, if the taskbar is positioned at the top of the screen or on the left, the effective origin for positioning APL objects is more accurately represented by the first two elements of `workarea` rather than by `0 0`.

If you position your top level forms carefully, you may want to use this property. When the user moves the taskbar, it triggers a `SystemChange` event with a `SettingChange` message.

wrap

Description:

Integer value that specifies whether the spinner wraps around when the upper or lower limit is reached or any child object of the command bar causes the command bar to wrap to a new line.

Classes: `CommandButton` and `Spinner`.

Remarks:

Values for a wrap Property

Value	Description
Primary Value: Choose only one value	
-1	Explicitly not wrapped (never wrapped)
0	Implicitly not wrapped
1	Implicitly wrapped
2	Explicitly wrapped (always wrapped)

For a spinner, this value specifies that the `value` property wraps around as it changes; that is, returns to its minimum when the value reaches the upper limit or to its maximum if the value decreases to its lower limit. This property can only be set to zero or 1. The default value of zero specifies no wrapping. You can set this property to 1 to allow wrapping.

For children of a command bar, this flag specifies that you want its parent command bar to wrap to a new line after the child object that has this property set to 1. This takes effect only when the `style` of the command bar includes 16, fixed wrap positions.

The setting is effective both for docked and floating command bars. Note that when you float any bar with multiple children, the underlying Windows system may display it on multiple layers. Using `style 16` on the bar forces the wrap at the location you specify with this property. When the bar is floating, and its `style` does not include 16, you can reference this property to determine where the system has caused the bar to wrap, but setting this property to 1 for any button on the bar is ignored.

Do not confuse the effect of setting the `wrap` property on a command button with the effect of setting the `dockbreak` property on a command bar. The former causes the next child of the bar to wrap to a new line within the same bar; the latter causes the entire bar to move to a new band on the side where it is docked.

xclasses

Description:

Four-column matrix that returns a list of the ActiveControl classes registered with Windows on your system:

- The first column contains the plain text name of the ActiveX control; this is the same value that appears in the `description` property of an instance of that ActiveControl.
- The second column contains the alphanumeric string beginning with a brace that identifies the class of the ActiveX control; this is the same value that begins the `class` property of an instance of that ActiveControl.
- The third column contains the program identification of the ActiveX control; this is the same value that appears in the `progid` property of an instance of that ActiveControl.
- The fourth column contains the version number of the ActiveX control; this is the same value that appears in the `version` property of an instance of that ActiveControl.

Class: # (the system object).

zoom

Description:

Numeric singleton that specifies a magnification or reduction of the contents of a RichEdit control.

Class: RichEdit.

Remarks:

This property allows you to magnify or reduce the display of text in a RichEdit. To enlarge the display you can enter any number greater than 1 up to, but not including 64. To reduce the display, you can enter a fractional number; theoretically you can go to $1/64$, but small numbers may have a visible effect only when the font size is very large. For any font, there may be threshold effects. Changing font size is not effective until you reset zoom to 1. This facility requires a version 3 RichEdit.

Δudp (user-defined data properties)

Description:

For any object, including the system object, that has a `data` property, you can add additional data properties. Simply name the property using the delta symbol (Δ) as the first character and follow the rules for naming APL variables.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List,

Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

If you reference a user-defined data property to which you have not assigned a value, the system returns a scalar integer zero. You can assign any value.

Data properties that have a non-zero value appear at the end of the list if you reference the properties property for the object. The values you assign are also stored in the state property, which means they are saved as part of the def property. If you assign a scalar zero to the property, it disappears from the properties list and the state property. If you want to maintain the presence of this property for a given object with the value zero, you can assign a vector zero.

Among the many possible uses for these properties, you may want to use them to replace those functions you are now performing with the WData utility function.

ActiveX Array Properties

ActiveX classes can have one or more array properties. These arrays may have multiple index notation. The following rules apply to ActiveX array properties:

- For a single index notation for one value, you can use the empty brackets, or you can just name the property without any additional annotation. Thus, you can reference a single value with either of these statements (where *col* is the number of the column):

```
width ← 'fmNew.ocxGrid' □wi 'xColWidth[]' col
width ← 'fmNew.ocxGrid' □wi 'xColWidth' col
```
- You can set a single value by adding a second argument (where *col* is the number of the column and *size* is the value you want to assign):

```
'fmNew.ocxGrid' □wi 'xColWidth[]' col size
'fmNew.ocxGrid' □wi 'xColWidth' col size
```
- If you want to reference or set multiple values with a single index, use the brackets after the name of the property:

```
width ← 'fmNew.ocxGrid' □wi 'xColWidth[]' (col1 col2 col3)
'fmNew.ocxGrid' □wi 'xColWidth[]' (col1 col2 col3) (size1 size2 size3)
```
- You can assign the same value to multiple cells by specifying only a scalar for *size*; otherwise the number of values must match the number of cells.

Using Multiple Indexes

If you have a property that needs multiple indices to reference one value, for example, if you supply both a row and a column for a grid cell, you supply the index values in a vector or a matrix. To reference one value, you would use a vector. To set one value, you would specify a value:

```
cell ← 'fmNew.ocxGrid' □wi 'xTextRC[]' (col row)
'fmNew.ocxGrid' □wi 'xTextRC[]' (col row) ('12345.678')
```

If you identify multiple values in a matrix, each row must contain one set of index values; the number of columns must match the number of indices. The statement below would return three cells:

```
x←(1 2 3,[1.5] 4 5 6)
cells ← 'fmNew.ocxGrid' □wi 'xTextRC[]' x
```

However, if you specify two vectors of values, the index values are catenated in outer-product fashion. Thus, the following statement returns a square of nine cells.

```
cells ← 'fmNew.ocxGrid' □wi 'xTextRC[]' (1 2 3) (4 5 6)
```

The result would be a 3-by-3 matrix holding the values of cells (1,4) (1,5) (1,6) (2,4) (2,5) (2,6) (3,4) (3,5) and (3,6).

If you want to set values, the values follow the indices. If you wanted to set values in the square, you could follow the arguments by a single value or by nine values. If you want scalar extension to apply to a vector you must enclose it. Thus, the first statement would place one letter in each cell, while the second would place the nine-letter string in each cell:

```
'fmNew.ocxGrid' □wi 'xTextRC[]' ((1 2 3) (4 5 6)) ('abcdefghi')
'fmNew.ocxGrid' □wi 'xTextRC[]' ((1 2 3) (4 5 6)) (<'abcdefghi')
```

Chapter 3: Windows Interface: Event Handlers

Introduction

Event-handler properties of objects allow you to associate APL expressions with various occurrences for an object. Events provide a means for the system to notify your application regarding actions that affect it, such as user modification of the text in an Edit control or clicking the mouse on a button. If you set the `onClick` event-handler property of an object, the system executes that handler whenever a Click event occurs for the object. If the object is a button, the Click event occurs when the user clicks the button with the mouse or moves the focus to the button and presses Enter or the spacebar.

In addition to actions you specify, Windows performs certain actions by default. When a user clicks and then drags the mouse, for example in an edit control, the text over which the mouse arrow passes is highlighted. If the user then presses Delete or Backspace, the highlighted text disappears.

The combination of default system behavior and event handlers controls the behavior of your application. When a user clicks on a menu item on the menu bar of a form, the menu for that item drops down. Your handlers and the menu options you provide determine what happens when the menu drops down and control what happens when a user clicks on an option and the menu closes up.

For example, when a user selects a certain menu option, you may want to gray out another menu, activate some disabled items, change the value property of an object, place the focus in an input field, and display a message. You accomplish all this with an expression for the `onClick` event handler that contains the series of commands to make these things happen. You can cancel an event handler by specifying an empty vector as the expression or by inserting the lamp (A) symbol as the first element.

Event-handler properties are characteristics of objects. As with regular properties, they apply to classes. You can use the same event handler for different instances of a class or you can write a special handler for each object. Just as you do not have to assign a value to each regular property for a given object, you do not have to write a handler for each event-handler property. When an event with no handler occurs, no action is taken other than the default system behavior.

For example, if you have a Combo control, you may not want to specify an `onChange` event handler that the system executes every time the user types a character to change the text in the edit portion of the control. The system still scrolls the list portion of the control if the entered text matches one of the entries. When the user actually selects an item, you can use the `onClick` event handler, or if the user moves the focus to another object, you can have the `onCloseUp` or the `onUnfocus` event handler perform a relevant action on the complete input.

In addition to user actions causing events, you can trigger events by invoking a method or by taking other actions in your program. For example, if you close a form using the `Close` method, the same `onClose` event handler runs as when the user clicks the button on the menu bar that closes the form.

Some program actions can trigger events indirectly, but you can control whether they do or not. For example, if you set the value property of an option control (radio button) to 1, meaning it is the one selected, the system normally triggers the option's `onClick` event handler, just as it does when the user clicks the option, moves the focus to it by using an arrow key, or presses the spacebar when it already has the focus. You can suppress the trigger of the program setting the value property by setting the option's `suppress` property to 1. This allows you to reset a default configuration, for example, without running an event

handler that you want to occur only when the user actively selects that option. See the description of the `suppress` property for more information. Throughout this chapter, program actions that trigger events conditionally will be noted with "unless suppressed."

In order to understand event-handler properties, you must be aware of the following global system variables:

- `□wself` is a character vector that contains the name of the current object when a callback occurs. The default value of this variable is empty when no callbacks are pending.
- `□wevent` is the name of the event whose handler the system is executing. If a `□wcall`-defined filter generates the callback, `□wevent` is an empty character vector. If you invoke the `Exec` method using `□wi`, `□wevent` has the value `'Exec '`. When no callbacks are pending, `□wevent` is an empty vector.
- `□warg` contains the argument to the event handler, when the event requires parameters to more fully define it. For example, a `Change` event on a `Scroll` bar uses `□warg` to define the current position of the slider (thumb; scroll box) when the user moves or finishes moving it.
- `□wres` is the result value for the event callback. You use this variable as a switch or a flag to modify or suppress the default system behavior. The APL system processes the callback with whatever actions you specify, but using this flag allows you to control whether Windows performs the default behavior associated with the event, ignores the event, or processes the event in a nonstandard manner.

If you set one of these variables during a callback, the value persists only until the end of the callback. If you set it in immediate execution mode when no callbacks are on the stack, the setting persists as with any other variable.

You cannot define event handlers for objects that do not have the corresponding event property. For example, you cannot define an `onClick` event handler for a `Label` object, because `Label` objects do not generate `Click` events. You also cannot define the events that trigger the handlers. It is possible, however, to simulate the effect of a user-defined event by using the `Send` method.

The event-handler properties are listed alphabetically. They include the following:

<code>onAction</code>	<code>onDragOver</code>	<code>onHide</code>	<code>onProtected</code>
<code>onActivate</code>	<code>onDragStart</code>	<code>onHotChange</code>	<code>onReopen</code>
<code>onBeginLabelEdit</code>	<code>onDrop</code>	<code>onKeyDown</code>	<code>onReqResize</code>
<code>onChange</code>	<code>onDropDown</code>	<code>onKeyPress</code>	<code>onResize</code>
<code>onClick</code>	<code>onEditEnd</code>	<code>onKeyUp</code>	<code>onResource</code>
<code>onClose</code>	<code>onEditNL</code>	<code>onLimit</code>	<code>onScroll</code>
<code>onCloseUp</code>	<code>onEditLoad</code>	<code>onMemory</code>	<code>onSelChange</code>
<code>onColClick</code>	<code>onEditSaved</code>	<code>onModified</code>	<code>onSelect</code>
<code>onColResize</code>	<code>onEditStart</code>	<code>onMouse</code>	<code>onSend</code>
<code>onContextMenu</code>	<code>onEndLabelEdit</code>	<code>onMouseDouble</code>	<code>onShow</code>
<code>onDbtClick</code>	<code>onError</code>	<code>onMouseDown</code>	<code>onSort</code>
<code>onDdeConnect</code>	<code>onExit</code>	<code>onMouseDrag</code>	<code>onSpin</code>
<code>onDdeDisconnect</code>	<code>onExitError</code>	<code>onMouseEnter</code>	<code>onStart</code>
<code>onDdeExecute</code>	<code>onExpanded</code>	<code>onMouseLeave</code>	<code>onStop</code>
<code>onDelete</code>	<code>onExpanding</code>	<code>onMouseMove</code>	<code>onSystemChange</code>
<code>onDestroy</code>	<code>onFile</code>	<code>onMouseUp</code>	<code>onTimer</code>
<code>onDockChange</code>	<code>onFocus</code>	<code>onMove</code>	<code>onTooltip</code>
<code>onDragContinue</code>	<code>onFormatQuery</code>	<code>onNew</code>	<code>onUnfocus</code>
<code>onDragEnd</code>	<code>onFormatString</code>	<code>onOpen</code>	<code>onUpdate</code>
<code>onDragEnter</code>	<code>onGetDayState</code>	<code>onPage</code>	<code>onUserKeyDown</code>
<code>onDragFeedback</code>	<code>onGetVirtualData</code>	<code>onPaint</code>	<code>onValidate</code>
<code>onDragLeave</code>	<code>onHelp</code>	<code>onPrompt</code>	<code>onWait</code>

onAction

Trigger:

After an object is created, every `⌵wi` statement that applies to it triggers the Action event. You can specify any special behavior for an object in the function that is called by `onAction`; this event handler is designed to be used in conjunction with `onNew` for creating user-defined classes.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

When the application program sets or references a property, or invokes a method, a statement invokes `⌵wi` on the user-defined object. This triggers the Action event, and the `onAction` event handler responds. The result is determined by the code in the event handler. During execution of an `onAction` event handler, `⌵wself` contains the name of the object and `⌵wevent` is always: 'Action'. The variable `⌵warg` contains the normalized right argument to `⌵wi`, and `⌵wres` is initialized as: `0 0 ρ 0`.

Note that `⌵warg` is always normalized as a nested *vector* whose first item is a character *vector*.

This is true even for cases such as `⌵wi 'DoIt'` (where `⌵warg` would be `1ρ<'DoIt'`) and `⌵wi 'N'` (where `⌵warg` would be `1ρ<1ρ'N'`).

The body of an `onAction` handler function will typically be structured as a `:select` statement with a `:case` clause for each action you want to handle. You should include a default `:else` clause (`⌵wres ← ⌵wi '★'`) that passes all other actions to the system for their normal built-in handling. It is important to have a statement of this kind to handle default processing; otherwise you must account for every conceivable possibility in your `onAction` handler function. With this default, you need define only the special behavior; the properties and methods that normally belong to the object, for example where, can simply be passed through for normal processing.

Handling events on an instance of a user-defined class presents a different problem than handling properties and methods. Events can be triggered other than by a program statement, frequently by a user action. In order for a user-defined object to respond, the system must, in essence, transfer the event trigger to the user-defined object so that the appropriate object responds. The technique used is to invoke a function in which `⌵wself`, `⌵wevent`, `⌵warg`, and `⌵wres` are localized to values you specify. When the function runs, it appears as though it is an event handler with the chosen name occurring for the user-defined object. Two methods, `Event` and `Defer`, invoke such a function.

As the class developer, you invoke one of these two methods in response to the actual trigger (that is, the specific user action when it will occur) as part of defining the behavior of your special class. When you, or an application programmer, uses your special class in an application, the specific behavior that occurs in response to the event is defined as an argument to the method. In using these methods while defining your class, you specify the event name, which can be the name of an event that is recognized by APL+Win or a name that you create. You provide a user-defined property to hold the definition of the event handler that is specified as part of the application to respond to the event.

Note that a diamond character in the `onAction` string is simply treated as another APL statement that is executed as part of the current `onAction` event handler statement.

Note that the value of an `onAction` event handler can be two or more strings separated by `⌵TCNL` character where in order to execute the next `onAction` statement following the `⌵TCNL` character, the current `onAction` event handler must invoke the '★' action-name prefix that takes the following format:

obj □wi '★'

without any arguments. There is no limit on the number of □TCNL-delimited expressions statements in a compound onAction string.

See the Creating User-Defined Classes chapter in this manual for more information.

onActivate

Trigger:

When a Listview item is activated by clicking on its image or primary label. This event handler responds when style includes either 16384 or 32768. If you have only the larger of the two styles specified, this event handler does not fire when a user clicks to select a new item. It fires when the user clicks a second time or clicks on an item that was selected by a means other than a click, for example, by arrowing to the item. If you have the smaller of the two style values specified, this event is triggered by a click on the item, whether or not it was already selected.

Class: Listview.

Remarks:

□warg contains the row ID. This event handler differs from onClick in that it fires only when the user clicks on the image or primary label. It differs from onSelect in that it fires when a user clicks on an item that is already selected.

onBeginLabelEdit

Trigger:

When the user single clicks on the label of an item of a Listview or the node of a Tree that already has the focus, and the style property is set to allow editing labels (512 for Listview, 16 for Tree); or you invoke the BeginLabelEdit method.

Classes: Listview and Tree.

Remarks:

When label editing is initiated by the BeginLabelEdit method, the system moves the focus to the specified item, making it visible if necessary, and highlights it appropriately; for either the method or user action, the system encloses the text of the label in a box that allows character input (a system-generated edit control). □warg contains the identifier of the item being edited. Initially □wres contains a 1; you can set □wres=0 to prevent the editing session.

onChange

Trigger:

When a change occurs for the object. The Change event can occur as the result of user action with the keyboard or mouse or by program action.

Classes:

Combo, DateTime, Edit, Label, RichEdit, Scroll, Selector, Spinner, Trackbar, and Tree.

Remarks:

The Change event occurs for each change to the object, for example, with each character a user types into an Edit control. You use this event handler to perform an action with every change. If you want to validate input after the user has completed entry, you can use, for example, the onExit event-handler property for the individual object or the onClick event-handler property for the default button on the form.

- **Combo, Edit, and RichEdit**
For a Combo (except style 2) or either Edit object, a Change event occurs as a result of any change made by the user to the object's text property. However, it does not fire for a Combo when the text property changes as a result of a user clicking an item in the list portion of the control. (If you want to take an action based on a user selecting an item from the list, use the onClick event handler.) For a Change event on any of these objects, the value of `¶warg` is `0`. Unless suppressed, a Change event also occurs as a result of any change to the text property under program control, such as by:
`'fmMain.edBirthCity' ¶wi 'text' 'Not Available'`
- **DateTime**
For a DateTime object, a Change event occurs for any style whenever the user changes the value, even if the new value is not selected; `¶warg` contains the new value. You may have to handle multiple occurrences of this event for each change. In particular, for a visible calendar, you may get two Change events each time the user changes the date without selecting it, for example, by scrolling the display of months. In addition, you may get periodic spurious Change events for a visible calendar-style control.
- **Label**
For a Label object, a Change event occurs only in a DDE transaction, when a remote application sends a data item to the control; the value of `¶warg` is `0`.
- **Scroll**
For a Scroll object, a Change event occurs when the user moves the scroll bar with the mouse or the keyboard, which changes the value of the slider position. Unless suppressed, it also occurs when the Scroll object's value property changes under program control, such as by:
`'fmMain.sbHorz' ¶wi 'value' 100`
For a Change event on a Scroll object, `¶warg` contains an integer scalar that is the current position of the slider on the scroll bar. For example,
`'fmMain.sbVert' ¶wi 'onChange' 'ScrollWindow'`

```

    vScrollWindow
[1] 'fmMain.ed' ¶wi 'range' ¶warg
v

```
- **Selector**
For a Selector object, a Change event occurs when a new page is selected. The Change event occurs after the current page is changed, but before the Show event occurs for the newly selected page. When the event occurs, `¶warg` contains a two-element vector with the origin 1 index of the new and previous pages (in that order).
- **Spinner**
The onChange event handler occurs after the value changes. `¶warg` is an integer vector that contains the new value.

- **Trackbar**
The `onChange` event handler occurs after the value changes. `¶warg` is an integer vector that contains the new value. A change in the trackbar value does not trigger the `onModified` event handler or set the `modified` property.
- **Tree**
The `onChange` event handler fires whenever the selected node changes; `onChange` occurs after `onClick` but before `onDbClick`. `¶warg` is a two-element vector that contains the node identifiers of the newly selected node and the previously selected node.

onClick

Trigger:

When a mouse click occurs for the object. The Click event can occur either by user action or, for some classes, by using `¶wi` to invoke the `Click` method or set the `value` property.

Classes:

Button, Check, Combo, CommandButton, List, Listview, Menu, Option, Status, and Tree.

Remarks:

You use an `onClick` event handler to perform the tasks the user requests by clicking the left (primary) mouse button on a control or by selecting it in another manner such as by pressing the Enter key when the focus is on a button. You can also trigger the `onClick` event handler under program control for Button or Menu objects by using the `Click` method, and, for all the relevant classes except CommandButton, Listview, and Status, by setting the `value` property. For a Check box, setting `value` always triggers the Click event; for the other six classes, you can suppress this behavior by setting the `suppress` property to 1. For a style 2 Combo box only, setting the text to a string that matches an item in the list also triggers `onClick`.

If you are using event handlers for Mouse and Click events, the order of the events is `MouseDown`, `MouseUp`, and then `Click`. You should avoid unintentionally handling the same event more than once.

- **Button, CommandButton, and Check**
The button or check box was clicked. The value of `¶warg` is $\emptyset$.
- **Combo and List**
The user selected an item in the list using either the mouse or cursor-movement keys (Up arrow, Down arrow, Home, or End). The value of `¶warg` is $\emptyset$.
- **Menu**
The user selected the menu item. `¶warg` contains 0 if the menu item is not checked; 1 if it is checked.
- **Option**
The user selected this radio button option. (Others in the group are automatically unselected.) The value of `¶warg` is $\emptyset$.
- **Tree and Status**
The user selected a tree item or clicked on the status bar using the mouse. `¶warg` contains a six-element vector containing the identifier of the node or pane that was clicked, a nested character vector that shows what part of the node or status bar was clicked, the shift key state, and the row and column of the mouse in current scaling units. The sixth element is 1 or 2 showing the left or right mouse button depressed.
- **Listview**
The user clicked on the Listview. `¶warg` contains a seven-element vector containing information about the location of the click. The first element is the RowID, and the second is the column ID. The third element is a character vector, one of the applicable location values provided by the Windows `Listview_HitTest` call. It can be one of the following: `image`, `label`, `imageuser`, `above`, `below`, `toleft`, `toright`, `aboveleft`, `aboveright`, `belowleft`, `belowright`, `nowhere`, or an empty

vector. The fourth element shows the shift key state: the sum of 1 for the Shift key, 2 for Ctrl, and 4 for Alt. The fifth and sixth elements show the row and column location of the mouse in current scaling units. The seventh element is 1 or 2 showing the left or right mouse button depressed.

onClose

Trigger:

When something attempts to close any object. The Close event can occur by a Windows system action, by user input action, or by invoking the Close method using `Obj`.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

If the object is a form, the event is an indication that Windows has been asked to close the form. The event handler can allow or disallow closing the form; it can also perform whatever other actions are appropriate before the form is closed. This event handler returns a result in `Obj`. Returning a value of `Obj` other than 0 allows the system to close the object; returning `Obj ← 0` inhibits closing.

Note that this behavior does not apply to a child object when its parent is closed; if the parent object closes, the child object automatically closes. The value of `Obj` is 0 except when you invoke the Close method. In this case, `Obj` contains the values to the right of 'Close' in the `Obj` statement.

- **Form and MDIForm**

If the object to be closed is a form, you can trigger this event handler several ways:

Clicking on a cancel push button. (A cancel button is a Button whose style property is 2).

Double-clicking or picking the close option from the form's System Menu or pressing Alt+F4.

Clicking the close button in the header of a form.

Explicitly invoking the Close method using `'object' Obj 'Close'`

You can prevent the form from being closed by setting `Obj ← 0`.

- **#**

This event occurs for the system object if the user attempts to shut down Windows or terminate the application using the Task Manager, the system command `Shutdown`, or by closing the session manager.

This is the appropriate time to prompt the user to save or discard any work in progress (and save it if the user opts to). If the user wants to cancel the shutdown, you set `Obj ← 0`.

You should not actually shut down the application at this time. Wait for the Destroy event telling you that all other applications have agreed to shut down and that Windows is actually terminating.

Example:

In the case of a search and replace form, you can use `onClose` to hide the form rather than actually closing it. This keeps the most recent data ready to use if the user selects the replace function again.

onCloseUp

Trigger:

Whenever the list portion of a Combo box or the drop-down calendar of a DateTime control is hidden.

Classes: Combo and DateTime.

Remarks:

The CloseUp event occurs when the user takes an action that frees the list or calendar, such as selecting an item or date or clicking elsewhere on the form. It is the opposite of a DropDown event. The value of `□warg` is 0.

onColClick

Trigger:

The user clicks on a column header of a Listview object in report view, unless its style property includes 128 or 256 (which hides the headers or disables the column header buttons, respectively).

Class: Listview.

Remarks:

You can use onColClick to allow the user to rearrange the data; for example, you could sort by the cell values in the column the user clicks. `□warg` contains the index number of the column header that was clicked and the shift key state (the sum of 1 for the Shift key, 2 for the Ctrl key, and 4 for the Alt key).

Example:

```
▽ ColClick;col;cells;sort;actual
[1] col←□warg[1]+2
[2] actual ← ('f.lv' □wi 'list')[;1]
[3] :if col=3
[4]     'f.lv' □wi 'sortorder' actual[4actual])
[5] :else
[6]     cells ← ▸('f.lv' □wi 'list')[;col]
[7]     sort ← □av ▲ cells
[8]     'f.lv' □wi 'sortorder' (actual[sort])
[9] :end
▽
```

onColResize

Trigger:

The user resizes a column of a Listview object in report view by dragging the margin between columns in the header (excluding style 128, which hides the headers).

Class: Listview.

Remarks:

`□warg` contains the index number of the column and the new width in current scaling units.

onComAction

Trigger:

When a command is executed in the ActiveX client session.

Class: System.

Remarks:

The ComAction event occurs in the APL+Win ActiveX server and is useful in debugging and/or error tracing purposes in the APL ActiveX server session. `□WARG` contains a nested vector of the right argument of the `□WFI` command that was executed in the client session.

onContextMenu

Trigger:

When a user clicks the right mouse button on an object.

Classes: Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar and Tree.

Remarks:

The ContextMenu event can occur only by user action. The `□warg` parameter indicates the mouse pointer coordinates of the event.

onCrypto

Trigger:

When the EncryptLog and DecrypFile methods are operating.

Class: # (the system object).

Remarks:

The `□warg` parameter contains two elements specifying the number of bytes processed and the number of total bytes to be processed. `□wres` can be set to 0 to cancel the operation. The values are floating point integers even if small enough to be integers.

onDbClick

Trigger:

When a user double-clicks the mouse on an object.

Classes: Combo, List, Listview, Status, and Tree.

Remarks:

The DbClick event can occur only by user action. For a Combo object, this event can occur only if the style property is 1, which specifies a simple Combo box (where the list is always down).

For Listview, Status, and Tree objects, `□warg` has the same values as it does for onClick. If both the onClick and onDbClick event-handler properties are set, when a user double-clicks on a list box or listview item or

the node of a tree, the system executes the `onClick` event handler first followed by the `onDbClick` event handler. When a user double-clicks, the `DoubleClick` event occurs after the `MouseDown` event, but before the (second) `MouseUp` event. You can set `Onwres←-1` or `Onwres←-2` in the `onMouseDown` event handler to suppress the `onDbClick` event handler. For a `Status` object, the system may execute the `onClick` event handler a second time, after the `onDbClick` event handler.

onDdeConnect

Trigger:

The establishment of a DDE conversation.

Classes: Edit, Form (server only), and Label.

Remarks:

The `DdeConnect` event occurs for an object when a DDE link is established with it. A Form that has posted a `ddeTopic` receives the `DdeConnect` event each time a client establishes a link. `Onwarg` is empty. A Label or Edit control that attempts to initiate a conversation as a DDE client receives a `DdeConnect` event when the connection is successful, except when you establish the connection in the same statement that creates the control object.

Example:

```
'fmServer' Onwi 'onDdeConnect' 'PostItems'
```

onDdeDisconnect

Trigger:

The disconnection of a DDE conversation.

Classes: Edit, Form (server only), and Label.

Remarks:

The `DdeDisconnect` event occurs when a DDE link is broken by the client. Actions that cause the server to disappear or disrupt its registration with Windows as a server break the DDE link but may not trigger the `onDdeDisconnect` event handler.

Example:

```
'fmClient.edXLdata' Onwi 'onDdeDisconnect' 'LinkBroken'
```

onDdeExecute

Trigger:

A client performs a DDE *Execute* command.

Class: Form (server only)

Remarks:

A form that is active as a DDE server receives the `DdeExecute` event when the client performs the DDE operation *Execute*, which is a request that the server execute a command. The value of `Onwarg` is the text of

the command, which may be any character string. If the client is an APL+Win application, it sends a command to the server using the DdeExecute method, supplying the command string as its argument.

Example:

```
'fmServer' □wi 'onDdeExecute' 'ServerExec □warg'
```

onDelete

Trigger:

When an object is deleted from APL.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, and Tree.

Remarks:

When the Delete event occurs, the object still exists, but has been closed. While it is usually not very useful to set properties or invoke methods on an object at this point, it is legal (as long as it is legal to use the property or method on a closed object). A typical onDelete handler will perform some cleanup such as releasing an HDC that was created using □wcall and assigned to a Printer's hdc property.

Visible objects require a "window" in the Windows operating system; if you can see the object on your screen, it has a window. Most of these objects (excepting Menu and CommandButton) have a Destroy event that occurs when the window is deleted from Windows memory. This occurs when the object is closed or deleted, a parent is closed or deleted, or you use the Reset method on the system object.

However, when you close an object without deleting it, the object still exists in APL. Thus, you can reopen an object without losing values and settings, etc., although it requires a new window in Windows when you do so. So, if you close an object, you can trigger an onClose event handler and an onDestroy event handler. If you later delete the closed object, you can trigger the onDelete event handler. You may have processing, such as releasing resources identified by a user-defined property, which exists only in APL, that you can profitably perform in the onDelete handler.

If an open object is deleted, the onDelete event handler fires after the onDestroy event handler for the object and any children. The onDelete event handler also occurs for objects that do not have a window, such as a Printer object, a Timer, or an Imagelist.

onDestroy

Trigger:

Your object is gone from Windows.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

The Destroy event occurs after the Close event has completed processing (and returned a nonzero value in

`Wres` indicating that it is OK to close the object). It is your best opportunity to release any Windows resources you acquired during the lifetime of an object.

You can place resource freeing logic in your `onClose` event handler, but that logic is sometimes not executed. For example, if you erase an open object by using the `Reset` method, no `Close` event occurs; a `Destroy` event does occur, however.

onDockChange

Trigger:

Whenever the dock property for a command bar changes. (The dock property has six possible states: floating, docked on one of the four sides of a container, or the bar is a client of the container.) The `onDockChange` event handler fires after at least one `onLayout` event handler has occurred.

Class: `CommandBar`.

Remarks:

During the event handler, the bar is hidden while it transitions. `Warg` is a two-element integer vector indicating the new and old values of the dock property.

onDragContinue

Trigger:

During a drag-and-drop operation, whenever the keyboard or mouse state changes or whenever a small time interval expires.

Classes: `Listview` and `Tree`.

Remarks:

This event handler occurs for the source of a drag-and-drop operation until the item is dropped or the operation is canceled. When the user drags the mouse, the system generates a string of events that allows you to monitor and control the action. You can use this event-handler property to determine when to drop the item by setting `Wres`.

`Warg` is a three-element vector:

- The first element is a Boolean indication whether the Esc key was pressed. If the user presses the Esc key, the system cancels the operation by default.
- The second element is the mouse button state; the sum of 1, 2, and 4 for the left, right and middle mouse buttons pressed. The user can press and release a mouse button other than the one that was held down to start the drag without changing the status of the drag-and-drop operation, but if the user releases the button that started the drag, the system drops the dragged item by default.
- The third element is the shift key state; the sum of 1, 2, and 4 for Shift, Ctrl and Alt keys pressed.

`Wres` is a character vector containing one of the following values: `'continue'`, `'drop'`, `'cancel'` or `'error'`. By default, the user actions determine what value the system reports. You can set `Wres` to one of the character vectors specified above to control the drag-and-drop operation by your own criteria.

onDragEnd

Trigger:

During a drag-and-drop operation, when the item is dropped or the operation is canceled.

Classes: Listview and Tree.

Remarks:

This event handler occurs for the source of a drag-and-drop operation when the item is dropped or the operation is canceled. It follows the last DragContinue event.

`⎕warg` is a two-element vector:

- The first element contains the same value as the first element of `⎕warg` for the DragStart event.
- The second element is a character vector that contains the drop effect returned by the target's `onDrop` handler, if the target is an APL object, or the drop effect reported by the system if the source was outside APL. If the drop was canceled, the value is 'none'.

`⎕wres` returns a value, but you cannot set it to control the drop.

onDragEnter

Trigger:

During a drag-and-drop operation, when the item is dragged over an object that is a potential target.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

This event handler occurs for the (potential) target of a drag-and-drop operation. A potential target has its `targetformats` property set to include a format provided by the item being dragged.

`EventArgs` is an eight-element vector:

- The first element is class-specific information describing the current location of the mouse in the target window for those classes that support the `HitTest` method; otherwise it is zero. For `Listview`, the first element is a three-element nested vector containing the item id under the cursor, a categorization of where on the item the cursor is (see `HitTest`), and the column id under the cursor (meaningful only for report view). For other classes that support `HitTest`, this is a two-element nested vector containing the name or id of the element under the cursor and a categorization of where on the control the cursor is (see the description of the `HitTest` method).
- The second element is the vertical position of the mouse in the control in current scaling units.
- The third element is the horizontal position of the mouse in the control in current scaling units.
- The fourth element is the state of the mouse buttons; the sum of 1, 2, and 4 for left mouse button, right mouse button, and center mouse button pressed.
- The fifth element is the shift key state; the sum of 1, 2, and 4 for Shift, Ctrl, and Alt keys pressed.
- The sixth element is the elapsed scroll time in seconds, if the drag causes the target to scroll.
- The seventh element is a character vector describing the direction of scroll.
- The eighth element is a character vector telling the first format in `targetformats` that the source is willing to provide.

`DropEffect` contains a drop effect; the value is based on the state of the Control key. APL drop targets return 'copy' if the Control key is down and 'move' if it is not. Note that these values reflect *de facto* Windows programming conventions. They do not determine the actual nature of the drop.

onDragFeedback

Trigger:

During a drag-and-drop operation, the `DragFeedback` event occurs for the source. In particular, it follows a `DragEnter` event or a `DragOver` event on a drop target.

Classes: `Listview` and `Tree`.

Remarks:

This event handler occurs for the source of a drag-and-drop operation. It captures values returned by the target that allow you to coordinate the drag-and-drop. `EventArgs` is a one- or two-element vector:

- The first element is the drop effect; it can be one of the following values: 'none', 'copy', 'move', or 'link'. APL drop targets return 'copy' if the Control key is down and 'move' if it is not.

- The second element is a character vector that contains 'scroll' if it is present. The target object returns this value if the dragging causes the target object to scroll. Windows changes the default cursor when a target object is scrolling. You could use this value, for example, as a trigger to display a special cursor when the user is dragging from an APL+Win WI control.

`□wres` returns one of the following character vectors: 'defaultcursor', 'continue', or 'error'. Initially it contains 'defaultcursor'.

onDragLeave

Trigger:

During a drag-and-drop operation, when the item is dragged off an object that is a potential target.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

This event handler occurs for the (potential) target of a drag-and-drop operation. A potential target has its `targetformats` property set to include the format of the item being dragged.

`□warg` is empty; `□wres` is initially zero; any value you set is ignored.

onDragOver

Trigger:

During a drag-and-drop operation that is already over a potential target, the mouse moved, the shift key state changed, or a small time interval elapsed.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

This event handler occurs for the (potential) target of a drag-and-drop operation. A potential target has its `targetformats` property set to include a format provided by the item being dragged. `□warg` is a seven-element vector.

These elements are the same as the first seven elements of `□warg` for the `onDragEnter` event handler:

- The first element is class-specific information describing the current location of the mouse in the target window for those classes that support the `HitTest` method; otherwise it is zero. For `Listview`, the first element is a three-element nested vector containing the item id under the cursor, a categorization of where on the item the cursor is (see `HitTest`), and the column id under the cursor (meaningful only for report view). For other classes that support `HitTest`, this is a two-element nested vector containing the name or id of the element under the cursor and a categorization of where on the control the cursor is (see the description of the `HitTest` method).
- The second element is the vertical position of the mouse in the control in current scaling units.
- The third element is the horizontal position of the mouse in the control in current scaling units.
- The fourth element is the state of the mouse buttons; the sum of 1, 2, and 4 for left mouse button, right mouse button, and center mouse button pressed.

- The fifth element is the shift key state; the sum of 1, 2, and 4 for Shift, Ctrl, and Alt keys pressed.
- The sixth element is the elapsed scroll time in seconds, if the drag causes the target to scroll.
- The seventh element is a character vector describing the direction of scroll.

`⌵wres` is a one- or two-element vector. The first (or only) element contains a drop effect; the value is based on the state of the Control key. APL drop targets return 'copy' if the Control key is down and 'move' if it is not. The second element is a character vector that contains 'scroll' if it is present. You can set `⌵wres` to 'none' in this handler to display the International No symbol as the cursor indicating your program does not want to allow a drop (despite the fact that the `targetformats` property matches the format of the item being dragged). Note that this setting does not actually prevent the drop, which is determined by the `onDrop` event handler.

onDragStart

Trigger:

During a drag-and-drop operation, when the user clicks on an item and drags the mouse.

Classes: Listview and Tree.

Remarks:

This event handler occurs for the source when the user initiates a drag-and-drop operation. `⌵warg` is a two-element vector:

- The first element is the identifier of the item being dragged; for a Tree, this identifies one node, but its descendants are included in the information being dragged, if the `sourceformats` property includes TreeItem. For a Listview, this includes the item where the mouse press occurred, but multiple items can be contained in the information being dragged, if the `sourceformats` property includes ListviewItem.
- The second element identifies the mouse button that was pressed when the drag started:
left button – 1; right button – 2.

`⌵wres` initially contains a 1; you can set it to zero to cancel the drag operation.

onDrop

Trigger:

The source dropped the item, typically by virtue of the user releasing the mouse button.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

This event handler occurs for the target of a drag-and-drop operation. `□warg` is a nine-element vector. The first eight elements are the same as the values for `□warg` for the `onDragEnter` event handler:

- The first element is class-specific information describing the current location of the mouse in the target window for those classes that support the `HitTest` method; otherwise it is zero. For `Listview`, the first element is a three-element nested vector containing the item id under the cursor, a categorization of where on the item the cursor is (see `HitTest`), and the column id under the cursor (meaningful only for report view). For other classes that support `HitTest`, this is a two-element nested vector containing the name or id of the element under the cursor and a categorization of where on the control the cursor is (see the description of the `HitTest` method).
- The second element is the vertical position of the mouse in the control in current scaling units.
- The third element is the horizontal position of the mouse in the control in current scaling units.
- The fourth element is the state of the mouse buttons; the sum of 1, 2, and 4 for left mouse button, right mouse button, and center mouse button pressed.
- The fifth element is the shift key state; the sum of 1, 2, and 4 for Shift, Ctrl, and Alt keys pressed.
- The sixth element is the elapsed scroll time in seconds, if the drag causes the target to scroll.
- The seventh element is a character vector describing the direction of scroll.
- The eighth element is a character vector telling the first format in `targetformats` that the source is willing to provide.
- The ninth element is a nested vector containing the data being dropped; the eighth element determines the format of these data.

`□wres` contains a drop effect; the value is based on the state of the Control key. APL drop targets return 'copy' if the Control key is down and 'move' if it is not. Note that these values reflect *de facto* Windows programming conventions. They do not determine the actual nature of the drop. The application is responsible for performing an appropriate action, such as inserting a new node or moving a node in a `Tree` object.

onDropDown

Trigger:

When the user presses the drop-down button on a Combo object, a DateTime object with a drop-down calendar, a drop-down (style 8) CommandButton, or selects a pull-down menu from the menu bar.

Classes: Combo, CommandButton, DateTime, Form, and MDIForm.

Remarks:

The DropDown event can occur only by user action:

- **Combo**
For a Combo, this event occurs after the user clicks the arrow and before the list drops down. This gives you an opportunity to update the list before the user sees it.
- **CommandButton**
For a CommandButton, this event occurs only for the style 8 button that has an associated drop-down arrow on the right side of the button. This event is independent and mutually exclusive from the Click event that is triggered when the user clicks the main part on the left side of the button.
- **DateTime**
For a DateTime, this event occurs only for those styles that have a drop-down calendar.
- **Form or MDIForm**
For a Form, this event occurs when the user first selects a Menu item on the menu bar that has children and therefore causes something to drop down, or when the user presses the alt key and causes the focus to move to the menu bar. It does not fire for lower levels of a cascading menu. This event handler allows you to update the state of menu items that depend on the current application context before the user sees the choices. For example, you might enable the Paste menu item if there is text on the clipboard.

Example:

```
'fmMain' □wi 'onDropDown' 'CheckPaste'  
  ▽CheckPaste  
[1] :if □wcall 'IsClipboardFormatAvailable' 'CF_TEXT'  
[2]   'fmMain.Edit.Paste' □wi 'enabled' 1  
[3] :else  
[4]   'fmMain.Edit.Paste' □wi 'enabled' 0  
[5] :end  
  ▽
```


onEditEnd

Trigger:

When exiting from an edit session for a existing object.

Class: # (the system object).

Remarks:

This event-handler property allows you to take an action when exiting from an edit session for an existing object. For this event handler, the value of `□warg` is the name of the object.

This may or may not be preceded by an `onEditSaved` event (depending upon how you exit the edit session).

onEditNL

Trigger:

When the Open Objects or Fetch Objects dialog box is being initialized before opening.

Class: # (the system object).

Remarks:

You can respond by returning a list of names that are to appear in the dialog box. Be careful about removing names that were in `□WRES` when the event started since that will make names in the workspace unavailable for editing via these dialogs. You should normally only add to the list of `□WRES` names. Don't worry about sorting the list. That is done for you automatically upon return from the event.

If you use this event to load the names of all available functions from your functions on file database, your repository will appear to be a seamless extension of the active workspace. In order to respond quickly each time one of these dialogs is being opened, you might need to cache a consolidated name list rather than rebuilding a name list by scanning the database each time.

`□warg` is a numeric scalar or vector specifying what kind of names are needed to be returned in `□WRES`. This can be variables only (2), functions only (3), or functions and variables (2 3). `□wres` is a character matrix of functions and variables names. When the event is fired this list contains the result of `□nl` for the `□warg` argument (functions, variables or both that appear in the workspace). You add additional names to this list to have them appear in the dialog list.

onEditLoad

Trigger:

When you start an edit session on an existing object or try to fetch an object into an edit or APL session.

Class: # (the system object).

Remarks:

This event-handler property allows you to take an action when starting an edit session on an existing object or fetching an object into an edit or APL session. Your typical response to this event would be to load the function named by `□warg` into the workspace so it can be edited or fetched. For this event handler, the value of `□warg` is the name of the object.

onEditSaved

Trigger:

When you save an edit session.

Class: # (the system object).

Remarks:

This event-handler property allows you to take an action when saving the edit session. For this event handler, the value of `⌵warg` is the name of the object.

onEditStart

Trigger:

When you start an edit session on an existing object.

Class: # (the system object).

Remarks:

This event-handler property allows you to take an action when an edit session is about to start. This is distinct from `onEditLoad` which is called before `onEditStart`. It tells you an edit session is about to be started and gives you a chance to control how it starts. For example, you can morph the requested data type into another based a naming convention or syntax of the line used to start editing. You can also initialize an edit session with a template such as copyright notice and standardized argument and result comment blocks (see the `⌵wres[3]` item below) and also suppress the "Unable to edit object" system message with setting `⌵wres[1]←-1`.

`⌵warg` is a two element vector: *name inWS*

name Name of session being started.

inWS True (1) if object exists in workspace; false (0) if new object.

`⌵wres` is a three element vector that you can modify: *type row init*

type Type of edit session to be started:

- 0 Not editable (because rank>2 or not simple type).
- 1 Function
- 2 Numeric array
- 3 Character vector
- 4 Character matrix

If you want to prevent an object from being edited, you can change the type to zero (0).

row Which row number to start on in edit session. Functions encode row and column as:

`65536+column row`. If you insert lines via the *init* result item you may want to adjust the setting on *row* to position the caret somewhere that you desire it should be placed.

init Initialization string for new session (such as "left FOO right;x;y;z"). You can modify this string to change the value used to initialize new edit session for objects that are being created (i.e., when *inWS*=0). However, your settings are ignored for existing objects and when starting a Numeric session.

This includes everything the user typed on the edit line. For example, if the user entered the following at the keyboard:

```
)EDIT x FOO r
```

The system would set *init* equal to "x FOO r". The *init* string contains whatever was typed following the)EDIT command, the ∇ del, or in the *Open Objects* dialog box. You can modify the *init* string to contain any initial value you would like to load into the edit session.

The *init* string can include multiple lines so you can make function templates. For example, you could create a standard comment at the top of new functions that includes copyright, argument and result comments (pre-initialized with the argument and result names parsed from the *init* string). If you do so, then you might want to adjust $\square wres[2]$ to specify the proper starting line number within the body of the function template you have created.

onEndLabelEdit

Trigger:

When the user ends a session of editing the label of the item of a Listview or the node of a Tree by moving the focus or pressing the Enter key, or when you invoke the CancelLabelEdit method.

Classes: Listview and Tree.

Remarks:

$\square warg$ contains the identifier of the item that was available for editing. $\square wres$ contains the new label if any characters were entered or a zero if the editing was canceled without any entry, for example by the user clicking elsewhere on the form. You can set $\square wres$ to a character vector, and that value becomes the new label. If you set $\square wres \leftarrow 0$ (or any numeric value), the editing is canceled and the label retains the value it had before the editing began. Virtual labels are not changed regardless of the value in $\square wres$.

onError

Trigger:

Whenever an error occurs while printing, for example, if the system runs out of memory; or when an error is signaled by the MCI player.

Classes: Printer and Media.

Remarks:

For a Printer, the value of $\square warg$ is the error number from the Win32 API call that failed.

For a Media control, this event handler applies to an MCI player only. $\square warg$ contains a two-element nested vector: the first element is the integer error code; the second element is a character vector containing the error message.

onExit

Trigger:

When a control is about to lose focus. Unlike the onUnfocus event handler, onExit is not triggered by transitions to another form, the APL session, or another application.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Frame, Label, List, Listview, Media, Option, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

`□warg` contains the name of the control where focus will move if no action is taken by the onExit handler. The default value of `□wres` is a character null.

You can use an onExit event handler to determine the validity of user input, for example by examining the text property of an Edit control. If the onExit handler determines that the input is invalid, it should set `□wres` with an appropriate error code or error message and return. Setting `□wres` to any value other than a character null triggers the ExitError event. The error code or error message set in `□wres` will be available within the onExitError event handler as `□warg`.

Do not attempt to set the focus within an onExit event handler. Doing so may cause recursion and can lead to system failure. If you wish to redirect focus, do so in the onExitError event handler. Do not attempt to display a message box in the onExit event handler. Doing so may cause screen flicker. If you wish to display a message, use the onExitError event handler. If the field that fails validation is a Combo or Edit control, the system automatically selects all the text in the edit box upon return from the onExit event handler.

onExitError

Trigger:

When an onExit event handler sets `□wres` to any value other than a character null and returns.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

`□warg` contains the value set by an onExit event handler in its `□wres`, typically an error code or error message. The default value of `□wres` is the name of the control that failed onExit validation.

An onExitError event handler can be defined at any level. For example, if you have a Form which contains a Frame with several Edit controls, you can define your onExitError event handler on any or all of the Edit controls, the Frame, the Form, or the system object. If an onExitError event handler is defined for more than one of these objects, the system executes the one at the parentage level closest to the control that triggered the error. Typically, you define a generalized onExitError event handler for the system object and simply display the message contained in `□warg`, using the Wmsg function found in the Windows workspace.

The onExitError event handler attempts to move the focus to the object specified in `□wres`. This allows you to safely redirect the focus. If the object is invalid or does not exist, `□wres` is ignored. Since the default

value of `pwres` is the name of the control that failed its `onExit` validation, by default the focus returns to that control.

onExpanded

Trigger:

When a node of a `Tree` object has expanded or collapsed and the `expandedonce` state is not set.

Class: `Tree`.

Remarks:

You can use the `onExpanded` event handler to perform actions after the node is expanded (or collapsed). This is the appropriate place to reset the `expandedonce` state. `pwarg` contains two elements: the first element is the node identifier; the second element is 1 if the node has expanded or zero if it has collapsed. The `expandedonce` state is part of the fifth column of the `list` property of a `Tree`. The system sets the `expandedonce` state for a node each time the node is expanded or collapsed; if you want the `onExpanding` and `onExpanded` event handlers to fire, you must specify `-expandedonce` to turn the state off every time the `onExpanded` handler is triggered.

onExpanding

Trigger:

When a node of a `Tree` object is expanding or is collapsing and the `expandedonce` state is not set.

Class: `Tree`.

Remarks:

You can use the `onExpanding` event handler to perform actions before the node is expanded (or collapsed), for example to add children, or you can prevent the node from expanding by setting `pwres←0`. `pwarg` contains two elements: the first element is the node identifier; the second element is 1 if the node is expanding or zero if it is collapsing.

onFile

Trigger:

When the current file was closed or a new file was opened by means other than setting the `file` property; for example, by using the `Open` command on the context menu.

Class: `Media`.

Remarks:

`pwarg` contains a character vector that is the name of the new file, if any. This event handler applies to an MCI player only.

onFocus

Trigger:

When an object gets the input focus.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, List, Listview, MDIForm, Media, Option, RichEdit, Scroll, Selector, Spinner, Trackbar, and Tree.

Remarks:

The Focus event can occur by Windows system action, by user action, by invoking the Focus method using `□wi`, or by setting the focus property of the system object using `□wi`.

onFormatQuery

Trigger:

Whenever the format property for a `DateTime` control is set and there is at least one callback field defined therein or when the control is opened and those conditions apply. (You may see the `FormatQuery` event twice for each callback field that is defined when the control is opened.) If no callback fields are defined in the format property, the event does not fire. You should be sure to define the `onFormatQuery` event handler before you set the format property with any callback fields.

Class: `DateTime`.

Remarks:

You use the `onFormatQuery` event handler to set the (maximum) size of any callback fields. The `FormatQuery` event occurs for each callback field defined in the format property; `□warg` is a character vector that contains the identifier of the field, that is 'X' or 'XX', etc. (Note that a single X is still a character vector, so if you use `:case` statements in your event handler, you must match to `(1ρ'X')` or `(, 'X')`. You set the size as a two-element vector in `□wres`, showing height and width in the current scaling units of the control. Practically, however, only the value of the second element is useful. The display should be no higher than your control.

onFormatString

Trigger:

Just before every `Paint` event for a `DateTime` control, the `FormatString` event occurs once for each callback field defined in the format property of the control. Thus, it occurs when the `value` property changes, either by user action or under program control; when the user arrows from field to field within the control; and when the control gets or loses the focus, is moved, is resized, or is covered or uncovered. It also occurs when you query the `text` property of the control, although this does not trigger a `Paint` event. Your event handler should be prepared to specify the contents of every callback field under any set of circumstances.

Class: `DateTime`.

Remarks:

You use the `onFormatString` event handler to specify the contents of any callback fields. When the event is triggered, `□warg` is a two-element, nested vector; the first element is a character vector holding the field identifier, for example 'XXX'. The second element is a seven element vector holding the current `value` property for the control. You set the contents of the callback field as a character vector in `□wres`. Any date

and time fields that are defined in the format property get their values from the value property. Note that you cannot set value in this event handler, because that would cause recursions.

onGetDayState

Trigger:

When a program action changes the display of the month or months in a `Date`Time control that has the visible (not drop-down) calendar, or when a user causes the display of the months to scroll.

Class: `Date`Time.

Remarks:

You can use the `onGetDayState` event handler to display dates in bold. Specify the dates you want to have in bold in a three-column matrix in `□wres`. The system initializes `□wres` to a three-column, empty, numeric matrix so you can concatenate (⌘) the dates you want in bold. Alternatively, you can simply reset `□wres` to a three-column matrix of dates.

To use this event handler, you need to recognize that the calendar display shows parts of two more months than those whose names appear in the title bar. The control displays up to a week of the end of the month preceding the first titled month and up to two weeks of the beginning of the month following the last titled month in the display. You can set some or all of these dates in bold, as well as any of those in the months that are displayed in full.

When the event handler is triggered, `□warg` contains a two-element, nested numeric vector. The first element is a three-element vector that contains the date of the first of the month for which the user can see only the last few days. That is, if the first (or only) full month on the calendar display is January 2003, then the first element of `□warg` is 2002 12 1. The second element is a singleton representing the number of months or parts of months that appear in the display. That is, if there are three full months displayed, the second element is 5.

onGetInfoTip

Trigger:

When the system attempts to display a value tip in a `Listview` and `Tree`.

Classes: `Listview` and `Tree`.

Remarks:

The `onGetInfoTip` event occurs only by user action when the user hovers the mouse over an item in the `Listview` and `Tree`. You can use the `onGetInfoTip` to request enhanced value tip information for items in a `Listview` and `Tree`.

When the event handler is triggered, `□warg` is a three-element nested vector for the `Listview`. The first and second elements are the `RowID` and column number for which the information is needed. The third element is a scalar defining the information needed; it can be one of the following values: 2 = on icon; 4 = on label; and 8 = on state icon.

When the event handler is triggered, `□warg` is a two-element nested vector for the `Tree`. The first element is the `NodeID`, for which information is needed. The second element is a scalar defining the information needed; it can be one of the following values: 2 = on icon; 4 = on label; 8 = on indent; 16 = on button; 32 = to right of label; and 64 = on state icon.

The value of `¶wres` on return from the handler determines the text to be displayed in the tooltip window. You can use `¶wres` to display a custom value tip or suppress a value tip with setting it to `' '`.

When an `onGetInfoTip` handler is defined, the `onGetVirtualData` handler will not trigger for the "info tip" data. But when the `onGetInfoTip` handler is not defined, the `onGetVirtualData` handler is triggered as before but it does so without giving meaningful `columnID` and where information; it only triggers when the mouse pointer is above the label and only for the first column.

onGetVirtualData

Trigger:

When the system attempts to expand a node of a `Tree` or display items in a `Listview` and encounters placeholders in the list property of the `Tree` or `Listview` rather than actual data. For `Listview`, a placeholder is `-1` for an image (column 2), or an empty numeric vector ($\emptyset$) for a cell (column 3 and above). You can also specify an `n-by-0` or an `n-by-1` matrix (with row IDs assigned) to denote virtual data. For `Tree`, a placeholder is `-1` for a label (column 3) or the unselected or selected image (column 4, elements 1 and 2).

For a `Listview`, this event is also triggered when the user hovers the mouse over an item (only the first column of the display in report view), which allows the program to provide an "info tip" for that item.

Classes: `Listview` and `Tree`.

Remarks:

You can use the `onGetVirtualData` event handler to populate part of a `Listview` or `Tree` object based on what the user selects, rather than building the entire object and holding it in memory. Your event handler must supply the needed information at the time the user clicks a node of a `Tree` or an item in a `Listview`.

- **Tree**

For a `Tree`, when the `onGetVirtualData` event handler is triggered, `¶warg` contains two elements, the node identifier and an enclosed list of one or more of the following character vectors: `'children'`, `'image'`, `'imageselect'`, and `'label'`.

The vectors in the second element of `¶warg` identify the information the system needs for the identified node. You must assign an array to `¶wres` with one element for each category of information specified in the enclosed list in the second element of `¶warg`. The values to assign are, respectively, the number of children, the index to a nonselected image, the index to a selected image, and the label. When the event handler returns, if `¶wres` contains an improper value or no value for the expected element, the system ignores that element and does not generate an error.

- **Listview**

For a `Listview`, when the `onGetVirtualData` event handler is triggered, `¶warg` is a three-element nested vector. The first element is a character vector defining the information needed; it can be one of the following values: `'cell'`, `'image'`, `'imageoverlay'`, `'imageuser'`, or `'tip'`.

The second and third elements are the `RowID` and the index of the column for which information is needed. Image data always have column index of 1.

The vector in the first element of `¶warg` identifies the information the system needs for the identified node. You must assign a value to `¶wres` for the item in `¶warg`. When the event handler returns, if `¶wres` contains an improper value or no value for the expected element, the system generates a reasonable default value and does not generate an error.

Example:

The following example shows a typical event handler for `onGetVirtualData` for a `Tree` object. The `:else` clause in lines 11 and 12 allow for future extensions to virtual data handling without breaking this code. Note that a handler for a `Listview` would be somewhat different, since you handle only one type of information at a time.


```

    ▽ GetVirtualData;type
[1] :for type :in (1+□io) > □warg
[2]   :select type
[3]   :case 'children'
[4]     □wres ← □wres, . . . A Number of children
[5]   :case 'image'
[6]     □wres ← □wres, . . . A Index for normal image
[7]   :case 'imageselect'
[8]     □wres ← □wres, . . . A Index for selected image
[9]   :case 'label'
[10]    □wres ← □wres, . . . A Enclosed label
[11]   :else
[12]     □wres ← □wres, 0      A Safe programming practice
[13]   :endselect
[14] :endfor
    ▽

```

onHelp

Trigger:

When a user presses the F1 key; when the user invokes context sensitive help by clicking the "?" button (if it exists) on the form's menu bar and then clicks a control; when you invoke the `ContextHelpMode` method and the user clicks a control; or when you invoke the `Help` method. This event handler can occur for the target, that is, a menu item that is highlighted or the control that has the focus when F1 is pressed, is clicked with the "?" cursor, or is the object of the `Help` method; however, if the event handler is not defined for the target control, `onHelp` occurs on the first control up the parentage chain that does have a handler defined. This allows you to have a global handler on the system object, or regional handlers on objects that have children.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

In order to have a context help button on the menu bar, a form's border property must contain 256 (the "?") and 16 (system menu and Close button), and it may not contain 32 or 64 (minimize/maximize buttons) or 512 (the form is a tool palette). You are more likely to use this combination on a subordinate form than on the main form of an application, which a user more often wants to minimize or maximize. The `onHelp` event handler fires when the user presses F1, regardless of the nature of the menu bar.

When the user clicks the context help button, the cursor changes to a normal pointer augmented by a large question mark. This does not change the pointer property, however; Windows has acquired the cursor. When the user next clicks on a control, the `Help` event fires. Since Windows has acquired control of the cursor, the first mouse action invokes only the `Help` event; that action does not cause, for example, a `Click` event on a button or a `MouseDown` on an edit control. This suppression is true only if the user clicks a control; if the first click is on a top-level form but not on a control, the `Help` event does not fire and `MouseDown` does fire.

You can initiate context help under program control by using the `ContextHelpMode` method on a form or MDIForm. This generates the pointer-plus-question mark that the user can then direct to a specific control. You can invoke the `Help` event with a top-level form as the target control only by invoking the `Help` method on the form itself. In the context of a parent MDIForm, context help does fire for child forms.

When the `Help` event occurs, `□wself` contains the name of the object that has the handler defined. The information about the target, the control or menu item for which you provide help, is contained in `□warg`.

`□warg` is a seven-element vector:

- The first element is the full name of the target control.
- The second element is the value, a single integer, of the `helpcontext` property of the target control.
- The third element is the value of the `help` property of the target control.
- The fourth element is a two-element numeric vector showing the location of the mouse click in coordinates and scale of the target control.
- The fifth element is the mouse button state; the sum of 1, 2, and 4 for left, right and center button down.
- The sixth element is the shift key state; the sum of 1, 2, and 4 for Shift, Ctrl and Alt keys pressed.
- The seventh element is 1 if the target control is a window or 2 if it is a menu.

This facility provides great flexibility in how you provide help to your users. You can write specific handlers for individual controls or one top-level handler that applies to all the children of the object that provides the context, or a combination of specific and general handlers. There are no content or usage restrictions on the `help` property, although it is intended to be used in conjunction with this handler. If you have online help files for your application, the `contexthelp` property, which is restricted to a single integer, allows you to store an index to the appropriate help topic for any control for which you provide help. You can invoke the Windows help facility using `□wcall` and the Windows API, with a statement resembling the following:

```
□wcall 'WinHelp' hwnd 'CONTEXT.HLP' 'HELP_CONTEXTPOPUP' topic
```

As an additional option, during this event handler, `□wres` is initially set to 1. If you set it to zero in the event handler, the system searches the remainder of the parentage chain for another `onHelp` event handler.

When you have the "?" button on a form, and a user invokes context help and then clicks a control, it is pretty obvious that it is a request for specific help. Whether you have the "?" button or not, when the user presses the F1 key, the request may not be so obvious. You can assume the user is requesting help on the control that has the focus or the highlighted menu item, or you can follow the less-well-thought-out practice of the major Microsoft applications. Their current behavior is that unshifted F1 invokes the online help (as if the user had clicked the Help menu and selected Contents and Index). Pressing Shift+F1 invokes context sensitive help only if a menu is pulled down; otherwise it merely puts the application into context help mode. You can adhere to this convention by checking the elements of `□warg`. If the fifth element is zero (no mouse button pressed) and the sixth element is odd (Shift key down), and the seventh element is 1 (control rather than menu), you can invoke the `ContextHelpMode` method (on the form or `MDIForm`) rather than providing help on the target control. The default system behavior does not distinguish between Shift+F1 and unshifted F1.

onHide

Trigger:

Just after an object is removed from the screen.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

The Hide event occurs only in response to a program action, either setting the visible property to zero or a negative number or invoking the Hide method. If the latter, the system sets the visible property of a control object to -1, of a menu to 0, or of a form (or MDIForm) to the negative magnitude of its current value.

When you invoke the Hide method, `¶warg` contains the values to the right of 'Hide' in the `¶wi` statement. If the onHide event-handler function stores a value in `¶wres`, the system returns this value as the explicit result of `¶wi`. The default result of `¶wi` is 0 000. If you set visible, the value of `¶warg` is 0.

onHotChange

Trigger:

When the user moves the mouse cursor over or off of a control or command button on a command bar.

Class: CommandBar.

Remarks:

`¶warg` is a two element nested vector that shows the name of the child that is currently under the cursor and the name of the child that the cursor just left. Either the first item or the second item may be an empty character vector.

onKeyDown

Trigger:

When a key is pressed while the focus is on an object.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, List, Listview, MDIForm, Media, Option, Page, RichEdit, Scroll, Selector, Spinner, Trackbar, and Tree.

Remarks:

The KeyDown event occurs only by user action. This event handler is intended to handle actions that are associated with specific keys on the keyboard. The system executes the handler for all the keys that are normally associated with the class of the object in Windows. In an Edit control, for example, almost every key you press triggers the onKeyDown event handler. An exception is the Esc key, which triggers only the onKeyUp event handler. The `¶warg` parameter indicates the key that was pressed and its shift state.

Elements of `¶warg` for KeyDown and KeyUp Events

Element	Description
1	Key scan code (OEM-dependent); see note.
2	Repeat count.
3	Shift state; the sum of 1 = Shift, 2 = Ctrl, 4 = Alt.

Element	Description
4	Toggle state of the CapsLock key; on (1) or off (0).
5	Extended key (such as an arrow key); yes (1) .
6	Key is being repeated; yes (1).
7	Windows virtual key code; see note.

Note: The key scan code (`␣warg[1]`) is not a `␣av` index but an untranslated keyboard scan code that can vary depending on the hardware you use. The virtual key code (`␣warg[7]`) is a hardware-independent key code. The table of virtual key codes appears in an appendix in this manual.

Other control objects respond to a different set of keys; for example, the arrow keys do not trigger the `onKeyDown` event handler for a `Button` object. If you write an `onKeyDown` event handler for a specific key, the system may also take the default action for that key. For example, you can specify that when a user presses the A key in a specific `Edit` control, the system sets the `value` property for a related `Check` box to 1, causing that box to be checked. The letter A also appears in the `Edit` control, unless you intercept it with an `onKeyPress` event handler. No keys trigger `onKeyDown` for a `Page`, since it does not get the focus.

The value of `␣wres` on return from the handler determines how Windows processes the key after the callback has completed. Note, however, that in many cases by the time you process an `onKeyDown` event handler, there is already an `onKeyPress` key in the message queue. If you want to translate characters (for example from lowercase to uppercase), use the `onKeyPress` event handler. You can use `␣wres` to modify or suppress the action of a keystroke. The values you can assign to `␣wres` include:

Valid Values for `␣wres` for KeyDown and KeyUp Events

Value	Behavior
0	Accept the default action.
-1	Ignore the KeyDown or KeyUp event.
-2	Ignore the event and beep.
<i>any positive value</i>	Windows virtual key code. See <code>␣warg[7]</code> .

Note: If you want to take an action for an object with one specific key, there is another technique you can use. Create a hidden menu as a child of the object and assign the key as the shortcut key for the object. You can then define the `onClick` event handler for the menu to perform the action you desire.

onKeyPress

Trigger:

When a key that generates a character is pressed and released on an object.

Classes:

Combo, CommandBar, DateTime, Edit, Form, List, Listview, MDIForm, Media, RichEdit, Trackbar, and Tree.

Remarks:

The KeyPress event occurs only by user action. This event-handler property is intended to handle actions that are associated with character values rather than specific keys. It treats a character the same, regardless of how the user forms it. For example, a user can generate a capital A by holding down the left shift key and pressing A, holding down the right shift key and pressing A, toggling the Caps Lock key and pressing A, or holding the Alt key and pressing numeric keypad 0, 6, 5.

The system can execute the handler for all the keys that generate characters that are normally associated with the class of the object in Windows. There are many keys that generate onKeyDown and onKeyUp event handlers that do not generate onKeyPress event handlers.

The keys property is closely associated with the onKeyPress event handler. If keys is empty, the system executes this handler for all keys that generate characters. If you specify values in keys, the system executes the onKeyPress event handler only for the character values in keys.

This does not mean you lose other characters. For example, in an Edit control you can specify that the system translate lowercase letters to uppercase letters by designating the 26 numeric conversions (97 to 65, etc.). The system puts uppercase letters, numbers and punctuation marks into the control in the same manner as if you do not specify any onKeyPress event handler. If you want to limit characters in an Edit control, for example to numerals, your handler must suppress the letter characters.

The `¶warg` parameter indicates the character that was generated. The value of `¶wres` on return from the handler determines how the system processes the character. Characters are represented in `keys`, `¶warg`, and `¶wres` (when you use a positive value) by their index in the ANSI character set. See the Atomic Vector and ANSI Sequence appendix in the *System Functions Manual* for the index values. The table below shows the other values you can assign to `¶wres`.

Valid Values for ¶wres for KeyPress Events

Value	Behavior
0	Accept the character (default).
-1	Ignore the KeyPress event.
-2	Ignore the event and beep.
any positive value	ANSI index value of the translated character you want to appear in the control.

Example:

Translate o umlaut (ö) to oe.

```

▽ o_umlaut
[1] ¶wres←-1
[2] cursor←('fmMain.edEdit' ¶wi 'selection')[1]
[3] texta← cursor↑('fmMain.edEdit' ¶wi 'text')
[4] textb← cursor↓('fmMain.edEdit' ¶wi 'text')
[5] texta←texta,'oe'
[6] 'fmMain.edEdit' ¶wi 'text' (texta,textb)
[7] 'fmMain.edEdit' ¶wi 'selection' (cursor+2)
▽

```

```
'fmMain.edEdit' 0wi 'keys' 246
'fmMain.edEdit' 0wi 'onKeyPress' 'o_umlaut'
```

Translate braces to brackets:

```
'fmMain.cbCombo' 0wi 'keys' 123 125
'fmMain.cbCombo' 0wi 'onKeyPress' '0wres←0warg-32'
```

onKeyUp

Trigger:

When a key is released on an object.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, List, Listview, MDIForm, Media, Option, Page, RichEdit, Scroll, Selector, Spinner, Trackbar, and Tree.

Remarks:

The KeyUp event can occur only by user action. This event handler is intended to handle actions that are associated with specific keys on the keyboard. The system executes the handler for all the keys that are normally associated with the class of the object in Windows. The system executes the onKeyUp event handler for some keys that do not trigger an onKeyDown event handler. This allows you to take necessary actions in conjunction with Windows default actions. For example, in an Edit control, you normally associate pressing the Enter key with finishing input. You can use the onKeyUp event handler for the Enter key to control the next action on the form. You can also use the onKeyUp event handler to perform actions for a key that a user releases after holding it down for an extended time.

If you write an onKeyUp event handler for a specific key, the system may already have taken an action for that key before it executes the handler. The order of execution when a keystroke triggers all three event handlers is onKeyDown, onKeyPress, onKeyUp. The 0warg parameter indicates the key that was released and its shift state. The elements of 0warg are listed under onKeyDown. The value of 0wres on return from the handler determines how Windows processes the key. These values are also listed under onKeyDown.

onLimit

Trigger:

When a user attempts to type in more characters than allowed in an Edit or RichEdit control or the edit field of a Combo control.

Classes: Combo, Edit, and RichEdit.

Remarks:

You can limit the number of characters allowed for user input by setting the limit property for either type of Edit control or a Combo control. Using this event handler, you can provide a message or other help rather than have your application stop responding. You could also use this event handler to accept fixed length input without having the user press Enter or click an OK button.

onMemory

Trigger:

When Windows cannot allocate enough memory to meet a specific request.

Classes: Combo, Edit, List, and RichEdit.

Remarks:

The system can trigger the Memory event in a situation where Windows is unable to allocate enough memory to hold the dynamic part of a control, such as the text property of an Edit or RichEdit control or the list property of a Combo or List control.

onModified

Trigger:

When the modified property of an object changes.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, and Tree.

Remarks:

The value of `EventArgs` for this event handler is a four-element integer vector:

- The first element is the new value of the modified property.
- The second element is the previous value of the modified property.
- The third element shows the relationship of this object to the object that caused the modification ripple. If this object's modified property is being explicitly changed, the third element is zero. If a parent's modified property was changed by being cleared, the value is 1. If a descendant's modified property was changed by being set or cleared, the value is -1.
- The fourth element is a reason code. If a user action caused the modification, the value is 1. If the modified property was changed under program control, the value is -1. If the Modify method caused the change, the value is -2.

Note that when you clear a modified flag, the state of the modified properties of that object, its parents, and its children are in transition. If you want to know the value of the modified property while the `onModified` event handler is running, you should check the first two elements of `EventArgs`, which show the old value and the new value. You should not reference the property directly within the `onModified` event handler.

onMouse

Trigger:

When a mouse action occurs over a Toolbox object.

Class: Toolbox.

Remarks:

The Mouse event occurs whenever the user presses a mouse button over a toolbox cell. When the user presses a mouse button over a toolbox cell, the toolbox captures it. The system reports all subsequent mouse

actions to that toolbox. Moving the mouse over other parts of the application interface does not affect other control objects until the user releases the mouse button.

When a Mouse event occurs for a toolbox, `EventArgs` contains three values: *row column change*

The *row* and *column* parameters identify which cell of the toolbox is selected (the upper-left cell is row 1, column 1). The *change* parameter identifies which mouse action occurred. *change* can have these values:

1 - **Press**

The user presses the left mouse button down while the cursor is over a toolbox cell. Once a cell has been pressed, and until the mouse button is released, all subsequent Mouse events for a specific toolbox refer to this cell. Releasing the mouse button can result in a Release, Click, or Double-click change code.

2 - **Drag-off**

The user drags the cursor off the cell originally selected, without releasing the mouse button, after a Press value occurs.

3 - **Drag-on**

The user drags the cursor back over the cell originally selected, without releasing the mouse button, after a Drag-off value occurs.

4 - **Release**

The user releases the mouse button when the cursor is not over the cell originally selected, after a Drag-off value occurs.

5 - **Click**

The user releases the mouse button over the cell originally selected after a Press value occurs.

6 - **Double-click**

The user double-clicks the mouse button over a toolbox cell. This value is always preceded by a Mouse event with the Click value on the same cell.

Generally, you want the affected toolbox icon or bitmap image to change when a Mouse event occurs so the user knows something has happened. Typically, you define three images for each cell to represent three possible states (unselected, selected, and transitional). A 3-D image that appears "popped up" usually represents the unselected state. A 3-D image that appears "pushed down" usually represents the selected state. A 3-D image that appears "pushed down" more deeply than the selected state represents the transitional state. A transitional state occurs when the user presses a cell but does not release the mouse button, which commits a change between selected and unselected states.

You can use the `WtToolbox` function in the WINDOWS workspace to simplify processing of Mouse events and to manage the toolbox.

onMouseDouble

Trigger:

When the user presses the mouse button on an object, releases it, and presses the mouse button a second time in a short time interval.

Classes:

Button, Check, CommandBar, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

The MouseDouble event can occur only by user action. The `EventArgs` parameter indicates the button the user releases, the shift state, and the mouse pointer coordinates of the event (see `onMouseDown`). The value

returned in `¶wres` determines the action Windows takes for this event after APL+Win processes it. For the values you can assign, also see `onMouseDown`.

You can use this event handler to have your program take a different action when the user double-clicks than when the user single clicks. When the user clicks on an object twice in succession in a short time interval, the `MouseDownDouble` event occurs rather than a second `MouseDown`. `MouseDownDouble` is not the same as a `DbClick` event, although the same user action triggers both event handlers for an object that does receive the `DbClick` event. The `DbClick` event occurs between the `MouseDownDouble` event and the (second) `MouseDown` event. You can suppress the `DbClick` event by setting `¶wres←-1` or `¶wres←-2` (adds a beep) in the `onMouseDownDouble` event handler.

Note: By default, frames and labels are transparent to mouse events. However, if you set the `pointer` property to a nonzero value, or if you set one of the `onMouseXx` event handlers, the control recognizes all mouse events. This can affect your application if you have controls that overlap a frame or label.

onMouseDown

Trigger:

When the user presses the mouse button on an object.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

The `MouseDown` event can occur only by user action. The `¶warg` parameter indicates the button the user pressed, the shift state, and the mouse pointer coordinates of the event. These elements apply to the `MouseDownDouble`, `MouseDown`, `MouseDownDrag`, `MouseDownEnter`, `MouseDownMove`, and `MouseDownUp` events.

Elements of `¶warg` for various Mouse Events

Element	Description
1	Row coordinate of the pointer when the event occurred, in current scaling units.
2	Column coordinate of the pointer when the event occurred, in current scaling units.
3	Mouse button the user pressed (<code>onMouseDown</code> and <code>onMouseDownDouble</code>) or released (<code>onMouseDownUp</code>); one of 1 = left, 2 = right, 4 = middle; (always 0 for other events).
4	State of the mouse buttons when the event occurred; the sum of 1 = left button down, 2 = right button down, 4 = middle button down.
5	Keyboard shift state when the event occurred; the sum of 1 = Shift down, 2 = Ctrl down, 4 = Alt down.

The value returned in `¶wres` determines the action Windows takes for a `MouseXx` event after APL+Win processes it. You can assign these values:

Valid Values for `¶wres` for `MouseDown`, `—Move`, `—Up`, and `—Double`

Value	Windows Behavior
0	Process the event (default).
-1	Ignore the event.
-2	Ignore the event and beep.

Note: There are special conditions for `Frame` and `Label` objects. See `onMouseDownDouble`.

onMouseDown

Trigger:

When the user drags the mouse (moves it while pressing a button) on an object.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

The MouseDrag event can occur only by user action. When the user drags the mouse, the system generates a string of events that continually updates the location. You can use this event-handler property to simulate movement along the path of the mouse cursor motion.

The `dwarg` parameter indicates the buttons that are pressed, the shift state, and the mouse pointer coordinates of the event (see `onMouseDown`). The value returned in `dwres` determines the action Windows takes for the MouseDrag event after APL+Win processes it. You can assign these values:

Valid Values for `dwres` for MouseDrag Events

Value	Windows Behavior
1	Process the MouseDrag event, but ignore MouseMove.
0	Process the MouseDrag event (default).
-1	Ignore the MouseDrag event.
-2	Ignore the MouseDrag event and beep.

Warning: Scroll objects capture the mouse when you press the left mouse button (to control the slider). Therefore, a scroll bar will get MouseDrag events for the left mouse button only when you move the mouse over the scroll bar with the left mouse button already pressed.

Note: There are special conditions for Frame and Label objects. See `onMouseDouble`.

onMouseEnter

Trigger:

When the user moves the mouse from outside an object over the window that defines its location.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

The MouseEnter event can occur only by user action. It occurs when the user moves the mouse over an object regardless of whether any buttons are pressed or keys shifted. This event occurs before any MouseMove events on the same control. The `dwarg` parameter indicates the button and shift state, and the mouse pointer coordinates of the event (see `onMouseDown`).

Note: There are special conditions for Frame and Label objects. See `onMouseDouble`.

onMouseLeave

Trigger:

When the user moves the mouse off an object.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

The MouseLeave event can occur only by user action. It occurs when the user moves the mouse outside the window that represents the object's location.

Note: There are special conditions for Frame and Label objects. See onMouseDouble.

onMouseMove

Trigger:

When the user moves the mouse on an object.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

The MouseMove event can occur only by user action. It occurs when the user moves the mouse while it is over the object with this property, regardless of whether any buttons are pressed or keys shifted. When mouse buttons are down, the system also sends MouseDrag events. Be sure that you do not unintentionally process the same event twice as a MouseMove and a MouseDrag event.

MouseDrag events occur before the MouseMove events in this case. If you return `0` from the MouseDrag event handler, the system ignores the MouseMove event. This feature helps avoid confusion. However, if you want to process drag events as both MouseDrag and MouseMove events, you can leave `0` as the default value (zero) for both, which causes both events to occur.

The `warg` parameter indicates the button and shift state, and the mouse pointer coordinates of the event (see onMouseDown). The value returned in `wres` determines the action Windows takes for this event after APL+Win processes it. The values are also shown under onMouseDown.

Note: There are special conditions for Frame and Label objects. See onMouseDouble.

onMouseUp

Trigger:

When the user releases the mouse button on an object .

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

The MouseUp event can occur only by user action. The `␣warg` parameter indicates the button the user releases, the shift state, and the mouse pointer coordinates of the event (see `onMouseDown`). The value returned in `␣wres` determines the action Windows takes for this event after APL+Win processes it. For the values you can assign, see `onMouseDown`.

Warning: Listview and Tree objects may not get all MouseUp events, because Windows captures the mouse with the underlying control for MouseDown. It depends on how the user moves the mouse before releasing the button. Scroll objects capture the mouse when you press the left mouse button (to control the slider). Therefore, a scroll bar will get MouseUp for the left mouse button only when you move the mouse over the scroll bar with the left mouse button already pressed and then release it.

Note: There are special conditions for Frame and Label objects. See `onMouseDouble`.

onMove

Trigger:

When the user moves a form, or when the program moves a form or a control.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

The system executes an `onMove` handler after an object is moved to a new location. `␣warg` is the position of the object's upper left corner, in scaling units of the parent object.

onNew

Trigger:

When a `␣wi` statement invokes the `New` or `Create` method on a class name (the first argument following the `New` or `Create` action) that is not one of the built-in APL Windows interface classes and also not a registered ActiveX control or server object, or invokes one of these methods on a class name of a built-in class that is named in the `newclasses` property. (In the latter case, you can suppress the event handler by putting an asterisk in front of the class name in the `␣wi` statement.)

Class: # (the system object).

Remarks:

In the `onNew` event handler, you define the behavior and characteristics to create a new class of object or modify the behavior and characteristics of an existing class. When the handler returns, the system resumes

its normal processing. The fundamental idea is to interrupt the system's normal processing at the point of creating APL Windows interface objects.

The basic form of the `onNew` event handler statement is: `'#' □wi 'onNew' 'UDC_Fn'` where you can use any name you choose for the function, `UDC_Fn`, that defines the special behavior. This is a very powerful facility, but you must take extra care when you use it. Since you are modifying the existing behavior of the `□wi` system function, APL+Win provides you with much more flexibility, but also much more responsibility. APL2000 recommends that you use the following form of the statement. This protects you when you load a workspace that does not contain the function you are using to define new behavior.

```
'#' □wi 'onNew' "⍎(0≠□nc 'UDC_Fn')/'UDC_Fn' "
```

During the `onNew` event handler, `□wself` contains the name of the object being created, which is normally the left argument to `□wi`. The variable `□warg` contains the class name, and `□wres` contains the normalized right argument to `□wi`, including the method name, the class name, and any settings. For example, if an object is created with a statement like:

```
'form' □wi 'cked.New' 'CheckEdit' ('caption' 'Check it out')
```

- `□wself` is normalized as: `'form.cked'`
- `□wevent` is always: `'New'` for either the `New` or `Create` methods
- `□warg` is normalized as: `'CheckEdit'`
- `□wres` is normalized as: `'New' 'CheckEdit' ('caption' 'Check it out')`

If you want to create a class with a new name (as opposed to modifying an existing class), define an `onNew` event-handler function that creates an instance of your class; presumably, your handler code will detect the name of the class to be created. You must create at least one object recognized by APL+Win that has the name stored in `□wself`. When the function returns, the system processes everything in `□wres` using the object you created in the event-handler function. If you have not modified `□wres` in your `onNew` event handler, that will consist of the rest of the arguments to the right of the class name. You can modify items in `□wres` or add items to `□wres` that further define your class.

You can also define specific modifications in characteristics or behavior to one of the classes built in to APL+Win. The fundamental idea is the same: You intercept the processing of the system at the point of creating an object, insert the behavior you want to define, and return to the system's processing. The name of the class you want to modify must be in the `newclasses` property.

See the [Creating User-Defined Classes](#) chapter in this manual for more information.

onOpen

Trigger:

When you open any object.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, and Tree.

Remarks:

The `Open` event can occur only for an object that is not open. It occurs when you invoke the `Hide`, `Open`, `Print`, `Show`, or `Wait` methods on a closed object using `□wi`. It can also occur when you invoke the `New` method and define the `onOpen` event handler in the same statement.

You can open a closed object either directly, using the `Open` method, or indirectly, using the `Hide`, `Print`, `Show`, or `Wait` methods. Refer to the explanation of the `Open` method for information about the order of the control object `Open` and `Resize` events.

If you caused this event by invoking the `Open` method with `□wi`, the variable `□warg` contains the values to the right of the action '`Open`'.

onPage

Trigger:

When a new page is started but before any printing is done.

Class: Printer.

Remarks:

The `Page` event occurs each time the system begins a new page but before any printing is done. `□warg` contains the page number of the new page. A new page begins when the sum of the printer's position property plus the height of the character about to be printed exceeds the height of the page.

onPaint

Trigger:

When an object requires painting because, among other causes, it is being:

- shown the first time
- exposed by removing an object that obscures it
- disabled (greyed out) or enabled
- re-created by Windows, because, for example, the `style` property changed value
- increased or decreased in size or repositioned with respect to its parent
- for a combo, when the list portion drops down or closes up
- for a form, when it is moved, gets the focus, or is restored from an icon.

A `Paint` event also occurs when you use the `Paint` method with `□wi`.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

The `□warg` parameter contains `⊘` during the `Paint` event.

Example:

The following example uses the `Draw` method to draw a rectangle in the middle of a client area. Note that `□wi` without a left argument assumes the current object `□wself`.

```

    ▽ fmFoo_Paint;size;rect
[1]    A Draw a rectangle in the middle of the client area.
[2]    A Get the size of the object's client area.
[3]    size ← □wi 'size'
[4]    A Define the coordinates and extent of the rectangle.
[5]    rect ← 1 1 2 2 × 4 ρ size ÷ 4
[6]    A Draw the rectangle.
[7]    □wi 'Draw' ('Rect' rect)
    ▽

```

onPrompt

Trigger:

When you:

- move the focus to a control with a nonempty prompt property;
- highlight a menu item with a nonempty prompt property but have not yet selected it;
- move the focus from a control with a nonempty prompt property;
- move the highlight from a menu item with a nonempty prompt property to a control or a menu item with an empty prompt property;
- make another window active.

Class: # (the system object).

Remarks:

This event-handler property enables you to display a prompt for a control or a menu item on the status line. You can have the application display the short text message when the user highlights an item and replace the prompt or remove it when the user moves the highlight. Note that highlight is different than focus; the object that is highlighted may or may not be the same object that has the focus. When the user clicks on a control, the focus (in the Windows sense) moves to that control. However, a menu object does not receive focus. When a menu item is clicked, the focus is on another object, or on the form itself. Therefore, the onFocus event handler can perform different actions than the onPrompt handler.

The Prompt event includes losing the highlight to an item without a defined prompt property so the prompt for the previous item can be erased. `□warg` is a character vector that contains the name of the object that has received the focus or highlight.

Interpretation of the prompt property value is completely arbitrary. It might simply be the status line prompt text, or it could be a key used to look up a string resource or INI file entry containing the status line prompt text. It is completely up to the application how it organizes and uses the information. Note that you must set the onPrompt event handler for the system object explicitly each time you start APL; its value is not saved.

onProtected

Trigger:

When the user starts to edit text flagged as "protected" in a RichEdit control.

Class: RichEdit.

Remarks:

The fifth element of the `selstyle` property allows you to mark text as protected. There is no visual change to the text when you do this. If the user attempts to edit text that is protected, this event handler is triggered. You can prohibit the editing by setting `□wres ← 0`.

onReopen

Trigger:

When you take a program action that causes APL+Win to close and re-create a Windows object without actually closing the object in APL+Win.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, and Tree.

Remarks:

Typically, the Reopen event can occur if you change the style property such that Windows treats the object differently, for example, changing a Combo control to a drop-down list with no edit window from any other style. It can also occur if you change a border or edge value.

Note that if you change the primary value of the style of a Selector, it causes a Reopen event on the Selector and on the selected Page. Since the other pages have also been re-created, they are now closed. When a user or your program causes a different page to be selected, the onOpen event handlers (not onReopen) will run for the newly selected page and its children.

onReqResize

Trigger:

When the system recalculates the height of a RichEdit control that is required to display all the text.

Class: RichEdit.

Remarks:

When the user enters text into a RichEdit control, the system constantly recalculates the height needed to display all the text. It returns a four-element vector in `⎕warg`, showing the row and column where the upper left corner is located, the new height and the width. You can use this vector as an argument to the where property if you want to resize the control.

onResize

Trigger:

When you change an object's size.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

The Resize event can occur by Windows system action or by user action. It occurs by system action when resizing or opening an object such as when a form is minimized, maximized, or restored. It also occurs when the user changes the form size by dragging the sizing border of a resizable form.

This event occurs normally as part of opening an object. The Resize event occurs following the Open event and before the Paint event. See the descriptions of the Open and Wait methods for detailed explanations.

The `dwarg` parameter indicates the new size of the object and whether it is being maximized, minimized, and so forth. The values in each element of `dwarg` are:

Elements of `dwarg` for Resize Events

Element	Description
1	New height of the window in current scaling units of this object.
2	New width of the window in current scaling units of this object.
3	New state; 0 = resized, restored, or created; 1 = minimized; 2 = maximized.

onResource

Trigger:

When a resource for an `Imagelist` cannot be found, is not the proper size, does not have the proper attributes, or cannot be loaded or used for some other reason, or if you set the icon or pointer property for any object that has one to an invalid filename or resource, this event occurs for the system object.

Class: # (the system object).

Remarks:

This event handler allows you to handle a problem with an image resource without signalling an error that stops your application. `dwarg` can contain 2, 3, or 4 elements:

- The first element is a character vector that contains the name of the object for which the resource was intended; for example the name of an `imagelist`.
- The second element is a character vector that contains a reason for the failure; for example, 'resource is wrong height', 'invalid resource type', or 'unable to load'.
- The third element, if it exists, is a numeric index of the offending item in the argument, property, or operation.
- The fourth element, if it exists, is a character vector that contains a further identification of the problem. This could be a subset of the argument, a name, or a portion of the path.

`dwres` initially is zero. If you explicitly set this to 1 in the `onResource` event handler, the system signals a `OWI_RESOURCE_ERROR`.

onScroll

Trigger:

When the text in an `Edit` field scrolls vertically.

Classes: `Edit` and `RichEdit`.

Remarks:

When a user types more lines than a multiline `Edit` field can display, the text automatically scrolls, unless you have inhibited automatic scrolling in the `style` property. The system executes the `onScroll` event handler whenever the lines that are visible in the field change position. This occurs when the user types new material or moves through the field with the arrow keys or a scroll bar.

onSelChange

Trigger:

Whenever the selection property of a RichEdit control changes as a result of user action, and, if not suppressed, when the selection changes under program control.

Class: RichEdit.

Remarks:

The selection property changes when a user selects text by highlighting one or more characters. It also changes when the user enters or deletes text, as the selection cursor changes position in the control. Many of the properties of a RichEdit control are related to the current selection.

onSelect

Trigger:

After the selection state of one or more entries in a Listview changes due to user action or setting the value property, or when the user clicks on a date in a visible (not drop-down) calendar in a DateTime control.

Classes: DateTime and Listview.

Remarks:

For a Listview, this is similar to the onClick event handler for the List object. It occurs when the user selects a new item by clicking on an item or moving through the list using the arrow keys. Unless suppressed, it also occurs when you set the value property under program control. `Warg` is empty. A Listview also has an onClick event-handler property. If the user clicks on an item that is already selected in a Listview, the Click event occurs but the Select event does not.

Because Microsoft changed the underlying message structure regarding mouse clicks and selection changes, you can sometimes see anomalies. In a Listview, in report viewmode where you have only one column, the left and right arrows do not change the selection as they do, for example, in largeimage viewmode, or in a List control, where the left and right arrows act as up and down arrows. Despite this, the operating system treats a left or right arrow as a selection change. If this is important to your application, you should suppress the left and right arrows in an onKeyDown event handler.

For a DateTime control (style 0), the onSelect event handler fires after the onChange event handler. The onChange handler may fire many times without onSelect as the user moves through a calendar without actually selecting a date.

onSend

Trigger:

When you invoke the Send method using `W1`.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, and Tree.

Remarks:

This event handler facilitates arbitrary signaling between objects. The value of `Warg` is the value that

follows 'Send' in the argument to `owi`. If you set `owres` in an `onSend` event handler, it is returned as the result of the `Send` method.

onShow

Trigger:

Occurs when you place an object on the screen or when you invoke the `Show` method. You can cause the `Show` event by invoking the `Show` method using `owi`, or by changing the `visible` property to a positive value. If the object was hidden, the `visible` property for anything other than a form changes to a 1; for a form, a negative number changes to its absolute value. You can reference an object's `visible` property to determine the state it is in.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

The value of `owarg` is 0 except when you invoke the `Show` method. In this case, `owarg` contains the values to the right of 'Show' in the `owi` statement; for example, 'fmCustomer' `owi` 'Show' 12345 causes `owarg` to have the value 12345. This could be a customer number whose record you want to retrieve and place on the form using the `onShow` event handler so your application will display the customer data and the form simultaneously.

onSort

Trigger:

When you invoke the `SortChildren` method on a `Tree` object and include 'onSort' in the argument.

Class: Tree.

Remarks:

When the handler is invoked, `owarg` contains the identifiers of two nodes to be sorted. The handler must return a negative value if the first identifier should precede the second and return a positive value if the first value should follow the second. You can have `owres` return zero if the two nodes have equivalent order.

`SortChildren` triggers this event handler as many times as it requires to sort the children of the node you specify as one of its arguments. You can use this event handler with `SortChildren` when you want to reorder a node's children other than alphabetically, but it is ineffective if the `Tree` is closed.

onSpin

Trigger:

When the user clicks one of the buttons of a `Spinner`.

Class: Spinner.

Remarks:

The `Spin` event occurs before the value actually changes. `owarg` is a two-element integer vector: the first element is the increment of change; it can be positive or negative. When the user first clicks the spinner, the magnitude is 1, but if the user holds the mouse down on the spinner, the magnitude of the increment increases. The second element is the current value.

onStart

Trigger:

When the player starts playing a file in a Media control.

Class: Media.

Remarks:

The Start event occurs for all types of players.

onStop

Trigger:

When the player stops playing a file in a Media control.

Class: Media.

Remarks:

The Stop event occurs for all types of players. For the MCI player, `¶warg` contains 1 if the device was stopped; zero if it was paused.

onSystemChange

Trigger:

When a user changes a global Windows setting, such as the screen resolution. Specifically, the SystemChange event occurs when Windows sends `WM_DISPLAYCHANGE`, `WM_SETTINGCHANGE`, `WM_SYSCOLORCHANGE`, or `WM_PALETTECHANGE` messages. Additional changes may be added.

Class: # (the system object).

Remarks:

`¶warg` contains a two-element nested vector: the first element is a character vector designating the type of message received: 'DisplayChange', 'SettingChange', 'SysColorChange', or 'PaletteChange'. The fact that an item changed, thus triggering the event, is the most relevant part of this result. The item that changed is secondary, and the way it changed may or may not be necessary to know.

The second element is specific to the type of change.

- For a DisplayChange, the second element is a three-element numeric vector showing the screen height in pixels, screen width in pixels, and color depth in bits.
- For a SettingChange, the second element is a two-element nested vector with an integer value in the first element (you can learn more about this value by looking into the Microsoft documentation for `WM_SETTINGCHANGE`). The second element of the second element normally indicates the type of change (for example, Device, RefreshDesktop).
- For a PaletteChange, the second element is not relevant at this time.
- For a SysColorChange, the second element is an empty vector.

onTimer

Trigger:

Automatic when the timer object is open and enabled.

Class: Timer.

Remarks:

A Timer object allows you to make regular callbacks to APL+Win. A timer triggers Timer events only when it is both open and enabled. You set the frequency (in milliseconds) with the interval property of the timer. The default is 1000 (1 second). If you set the interval to zero, Timer events occur whenever the message queue is idle; that is, when there is no user input activity to any application and no other application is demanding CPU usage. When the interval is zero, your application performs background processing at the maximum speed that does not lessen the responsiveness of the system to user input or keep other applications from processing efficiently in the foreground.

`□warg` contains the system tick count, the number of milliseconds since Windows started; this value rolls over on the 50th day.

You can use a timer to blank the screen automatically after a specified time expires, perhaps if another event handler did not already blank the screen. Instead of intercepting every `MouseMove` event, you can define a timer to signal every 200 milliseconds to check the position of the mouse. You can also define a timer with the interval set to zero so the system notifies your application when the system is not busy.

onTooltip

Trigger:

When a tooltip window appears or disappears.

Class: # (the system object).

Remarks:

This event-handler property allows you to take an action when the system displays or hides a tooltip. For this event handler, the value of `□warg` is a three-element vector.

When a tooltip appears, the first element is the name of the control, the second element is the text of the tooltip, and the third element is an index. For all controls except a `Toolbox`, the index is 1. For a `Toolbox` the value is the index (in origin 1) of the tool button for which the tip is being displayed, counting left to right and top to bottom. When a tooltip disappears, the value of `□warg` is (' ' ' 0).

Example:

Suppose you are using the `onPrompt` event handler to display a status line message, but you want to coordinate prompts and tooltips. When the `Prompt` event occurs, you update the status line with the prompt.

When the `Tooltip` event occurs, if there is a tooltip to display, change the status line to show the prompt from the same control. When the tooltip disappears, reinstate the previous prompt message.

```

z←'fmMain' □wi 'New' 'Form'
z←'fmMain.bnOK' □wi 'New' 'Button' ('where' 1 1 1.5 6)
↳ ('prompt' 'Click me') ('tooltip' '(im)pressive')
z←'fmMain.edText' □wi 'New' 'Edit' ('where' 1 10 1.5 10)
↳ ('pointer' 5) ('prompt' 'Write in me')
↳ ('tooltip' 'Key field')
bar←'fmMain.stBar' □wi 'New' 'Status'

'#' □wi 'onPrompt' 'bar □wi 'SetStatus' 1 (□warg □wi 'prompt')'

'#' □wi 'onTooltip' 'overlay'

▽ overlay
[1] :if □warg[3]=1
[2]   bar □wi 'SetStatus' (-1) ((1>□warg) □wi 'prompt')
[3] :else
[4]   bar □wi 'SetStatus' (-1) ''
[5] :end
▽

```

onUnfocus

Trigger:

When an object loses the input focus.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, List, Listview, MDIForm, Media, Option, RichEdit, Scroll, Selector, Spinner, Trackbar, and Tree.

Remarks:

The Unfocus event can occur either by Windows system action or by user action.

When the system executes an onUnfocus event handler, Windows has a message to move the focus to another object. If you use this event handler to validate data, and you want to force the user to correct errors, you must be careful about when you reset the focus. Trying to place the focus back to an object in its onUnfocus handler may not be effective because Windows may override your setting, with unpredictable results. One way to avoid this problem is to validate the input data for the entire form when the user presses the OK button (in the button's onClick event handler) rather than validating each object. You can also use the onExit and onExitError event handlers for validation.

Example:

In the example below, as you click on one form, it triggers the event handler for the other.

```

'fmOne' □wi 'New' 'Form' ('onUnfocus' '□wself □wevent')
'fmTwo' □wi 'New' 'Form' ('onUnfocus' '□wself □wevent')

```

onUpdate

Trigger:

When the display of text in a RichEdit control changes on the screen.

Class: RichEdit.

Remarks:

When the text property of a RichEdit control changes, the Change event fires before the Update event.

When only the visible text changes, for example, because of scrolling, only the `onUpdate` event handler fires. You could use this event handler to call the `ReqResize` method, for example, when a user action does not automatically trigger the `onReqResize` event handler.

onUserKeyDown

Trigger:

When the focus is in a callback field (defined in the `format` property of a `DateTime` control) and the user presses a key. This event handler is similar to the `onKeyDown` handler for any control that has it, including a `DateTime` control, but the `UserKeyDown` event fires only in a callback field.

Class: `DateTime`.

Remarks:

When the focus is in a callback field and the user presses a key, `¶warg` is a two-element nested vector. The first element is a character vector that identifies the field, for example `'XXXX '`. The second element is a three-element numeric vector; the first element contains the virtual key code for the key that was pressed, the second element contains the keyboard shift state, and the third element contains the toggle state of the CapsLock key when the event occurred.

The keyboard shift state is the sum of three elements, 1=Shift Key down, 2=Control Key down, and 4=Alt Key down. However, note that you will not get values 4 and 5. The underlying `DateTime` control processes Alt+ any key, and Alt+Shift+any key in a different manner than the other combinations, so these do not trigger this event handler. You can see Alt+Ctrl+any key or all three shift keys down+any key. The toggle state of the CapsLock key is on=1, off=0.

The initial value of `¶wres` is the value property of the control. You can reset `¶wres` or change any of the elements of `¶wres` in your event handler. When the event handler returns, the system updates the value of the control; this in turn triggers the `onFormatString` event handler, which you can use to update all the callback fields. Note, however, that due to an anomaly in the processing of the underlying control, if you change only the seventh element of `¶wres`, the system may not update the value; in this case you must reset the value property to equal the changed `¶wres` in your `onUserKeyDown` handler.

onValidate

Trigger:

When a new `Page` of a `Selector` object is selected, when the current page is closed, or when the `Validate` method is invoked, or just before the selected node of a `Tree` changes.

Classes: `Page`, `Selector`, and `Tree`.

Remarks:

You can use the `onValidate` event handler to validate user input on the previously selected `Page` and to allow or disallow the `Page` to change. Both the `Selector` and the `Page` have `onValidate` event handlers. The `Validate` event occurs before the `Change` event.

`¶warg` contains two items: a numeric code that indicates what caused the validation to occur and a nested vector of the newly selected page and the previously selected page. The code is 2 if the user caused the page to change by mouse, 1 if by keyboard input, or zero if the reason is unknown. The code is negative if the program induced the change; -1 for the `Validate` method; -2 if the value property was changed; or -3 if the `Close` method was applied to the currently selected page.

The `onValidate` handler for the currently selected `Page` is executed first. If it succeeds, the `onValidate` handler for the `Selector` is called. When the event is fired, `□wres` is set to 1. The handler must set `□wres←0` before returning to indicate failure; any nonzero value indicates success. If the validation fails, a well-designed interface will display a modal message box indicating the reason for failure, and on return from the message box, put the input focus on the offending control so the user can fix the problem. In the case of edit controls this should usually be accompanied by selecting the text to make replacement easy.

For a `Tree`, the `onValidate` event handler fires just before the selected node changes. The first element of `□warg` contains a numeric code that identifies what action is causing the change. If the user changes the selection, the value is positive: 2 for a mouse click, or 1 for a keystroke. If the system does not recognize the cause, the value is zero. If the application causes the change, the value is negative: -1 if you invoke the `Validate` method, or -2 if you set the `value` property. You can set the `value` property directly, or indirectly by using the `ShowNode` method with the `select` operation.

In all cases except using the `Validate` method, `□warg` is a three element vector; the last two elements are the numeric node identifiers of the previously selected and newly selected nodes. If you use the `Validate` method and supply an argument, `□warg` is a two-element vector with the argument being the second element. If you do not supply an argument, `□warg` is a scalar.

onWait

Trigger:

Occurs during processing of the `Wait` method.

Classes: `Form` and `MDIForm`.

Remarks:

The `onWait` event handler is a good place to position a long-running, cancelable calculation that updates a progress gauge on the screen. This calculation program must call `□wgive` to recognize user actions. When the calculation is complete, use the `Close` or `Hide` methods on the form. This removes the form from the screen and causes the `Wait` method to return without entering a wait loop.

You can use the `onWait` event handler to execute code when a secondary, application-modal form is opened. When you trigger `onWait`, `□warg` contains the values to the right of '`Wait`' in the `□wi` statement. You can pass arguments to the handler to tailor the form, or you could notify the user of progress during a long-running calculation. An `onWait` event handler can also use `□wgive` to allow the user to interact with other applications while input to this application is temporarily restricted. When you use the `Close` or `Hide` method on the form, the form is removed from the screen and the `Wait` method returns.

Chapter 4: Windows Interface: Methods

Introduction

This chapter describes the methods, or functions, that are built into the Windows Interface (WI) facility in APL+Win. You apply methods to objects to perform the fundamental actions required to create and use a graphical user interface.

The syntax you use for any method is:

`'object' □wi 'method' {arguments}`

where *'object'* is a character vector containing the fully qualified name of the object and *'method'* is a character vector containing the name of the method. *arguments* are any additional parameters. For some methods, arguments are required; for example, the New method requires a Class name. For most methods, arguments are optional. Some methods do not allow arguments.

Some methods cause event handlers to be executed as part of their action. In these cases, the methods sometimes pass the arguments for the handler in the system variable □warg. For example, if you use the Open method as shown below,

`'fmMain' □wi 'Open' 'Joe' 12345`

it opens the fmMain object and then executes the onOpen event handler with □warg set to 'JOE' 12345 while the event handler runs. The event handler determines the meaning of the argument values.

The methods in this chapter are listed alphabetically. They include the following:

Abort	Delete	GetInfo	Pop	SetInfo
AddImages	DeleteNodes	GetPos	Popup	SetLinks
AddRows	DeleteRows	GetRows	Print	SetPos
Arrange	DevMode	Help	PrintDialog	SetRows
AutoFit	DevNames	Hide	Printers	SetStatus
BeginLabelEdit	Draw	HitTest	Push	Setup
CancelLabelEdit	Eject	HwndToName	Put	Show
CharToLine	EnsureVisible	InsertNodes	Redo	ShowNode
Click	EnumEnd	InsertRows	Ref	SortChildren
Close	EnumNext	LineToChar	ReleaseObjects	Stepit
Command	EnumStart	Modify	ReqResize	Stop
ContextHelpMode	Event	New	Reset	Undo
Copy	Exec	Notify	Resize	UseObject
Create	Expand	Open	ScrollCaret	Validate
Cut	FindNode	PageSetup	Seek	VT
DdeExecute	Focus	Paint	Send	VV
DdePoke	Get	PaneWhere	Set	Wait
DdeRequest	GetCells	Paste	SetCells	XInfo
Defer	GetImages	Play	SetImages	

Abort

Description:

Aborts the current print job.

Class: Printer.

Remarks:

The Printer object stops printing the current job and discards it. The Printer object remains open, and you can continue printing if you have another job defined.

AddImages

Description:

Populates an Imagelist or adds images to it after it is initialized.

Class: Imagelist.

Remarks:

The Imagelist must be open when you use AddImages. The argument can be any of the forms that are valid for the imagenames property. In addition, you can supply a singleton integer bitmap handle that you obtain through low-level Windows calls or through the Draw method using the ?Bitmap operation. Note that using AddImages copies the bitmap, so you can free the bitmap, using the Wfree function from the Windows workspace, immediately. You do not need to wait until you have finished using the Imagelist object.

Once you use AddImages, the imagenames property is set to an empty vector. The result of using AddImages is the number of images previously in the Imagelist.

AddRows

Description:

Adds items to a Listview object by adding rows to its list property.

Class: Listview.

Remarks:

The argument is a nested matrix that has the same form as the list property. The result is an integer vector of the added RowIDs. New rows are added at the end of the list; you can reorganize them by resetting the sortorder property.

Arrange

Description:

Tiles or cascades all non-minimized forms that are children of the current MDI form or arranges the icons

for all minimized child forms; for a Listview in largeimage or smallimage view, moves the icons to fit a grid based on the setting of the viewalign property.

Classes: MDIForm and Listview.

Remarks:

To tile forms is to arrange them in equal size areas so they are all visible. To cascade forms is to overlap them so the title bar of each is visible on the screen, but only the form on top of the cascade is wholly visible. These are standard Windows actions.

For an MDIForm, you must specify a scalar argument. These are the possible values:

Argument Values for the Arrange method for MDIForm

Value	Description
0	Arrange child form icons for minimized forms
1	Tile non-minimized child forms horizontally
2	Tile non-minimized child forms vertically
3	Cascade non-minimized child forms

For a Listview, you specify a character vector argument. These are the possible values:

Argument Values for the Arrange method for Listview

Value	Description
'default'	Allow the system to arrange items.
'aligntop'	Position items along the upper border.
'alignleft'	Position items along the left side border.
'snaptogrid'	Move items to nearest grid location.

The snaptogrid value is useful only if you have moved items to individual locations using drag-and-drop, positioning methods, or properties that specify position. See the description of the SetPos method for more information.

Example:

```
'mdiExample' □wi 'New' 'MDIForm'  
'mdiExample.fmOne' □wi 'New' 'Form'  
'mdiExample.fmTwo' □wi 'New' 'Form'  
'mdiExample' □wi 'Arrange' 1
```

AutoFit

Description:

Provides automatic sizing of one or more columns of a Listview object in report view.

Class: Listview.

Remarks:

You can specify one or more column numbers as an argument, or you can use 'all' to specify all columns. The system makes the width of each column fit the longest text string that is in a cell in the column. If you

specify selected columns, the unspecified columns remain at their current width and the overall width of the control is moved to the right as far as necessary.

Optionally, you can insert the string 'header' as a first argument. In this case the system makes the width of each specified column large enough to include the header text string if it is longer than any cell text entry. If you use this optional argument, and you include the rightmost column identifier in the arguments (or use 'all'), the system expands the last column to fill the width of the visible listview if the remaining columns require less than the entire space. If the columns require more space than is visible on the screen, the columns are pushed to the right, and the user must scroll.

BeginLabelEdit

Description:

Opens an editing session on the label of an item of a Listview or a node of a Tree.

Classes: Listview and Tree.

Remarks:

The style property must be set (512 for Listview, 16 for Tree) to enable label editing. The argument to this method is the identifier of the item or node you want to be available for editing. The system moves the focus to the item, making it visible if necessary, highlights it appropriately, and encloses the text of the label in a box that allows character input. Invoking this method triggers the BeginLabelEdit event. `¶wres` returns a 1 if the label is available for editing, or a zero if it is not.

CancelLabelEdit

Description:

Ends the current session of user editing of the label of an item of a Listview or the node of a Tree.

Classes: Listview and Tree.

Remarks:

The argument to this method can be zero or 1. If you use zero, editing ends and the label becomes what the user has typed up to this point. If you specify 1, editing is canceled and the label retains the value it had before the editing. If you do not specify an argument, the default value is 1. This method triggers the onEndLabelEdit event handler, which returns a value in `¶wres`. You can use `¶wres` from the event handler to set the label to any character vector value you want.

CharToLine

Description:

Returns the line number for a specific character.

Classes: Edit and RichEdit.

Remarks:

The CharToLine method takes a scalar integer argument that is the position of a character in the control, for example, the first element of the selection property. It returns the line number, in origin zero, on which that

character appears. You can think of "position" as the number of characters that precede the character in the text property of the control. Using -1 as the argument returns the line number of the last line in the control.

Click

Description:

Emulates a click event and triggers the onClick event handler.

Classes: Button and Menu.

Remarks:

You can use this method to take the action specified by the onClick event handler for a Button or Menu at a time when the user does not actually cause a click event. This allows your program to emulate the user action of clicking a menu choice or button, which triggers the object's onClick event handler.

Close

Description:

Requests permission to close an APL+Win WI (Windows interface) object. If no onClose event handler exists for the object or if the handler agrees, the system closes the object.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, ListView, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

When you use the Close method, you are essentially asking the object to close itself. The system executes the object's onClose event handler, if you defined one; the event handler can accept or reject the attempted closing by setting the value of the `□wres` system variable. If no handler exists, or if it does not set `□wres`, or if it sets `□wres` to 1, the system closes the object. If the handler sets `□wres` to zero, the system does not close the object.

You can include an argument to the Close method. The value or values of the argument are contained in `□warg` during the onClose event handler.

When you close an object, the system also closes all objects that it contains. For example, closing a form closes the form itself and all the controls on the form. However, when the system closes the children of the object on which you use the Close method, it does not execute their onClose event handlers. It does execute the onDestory event handler for the object itself and for any of the children for which one is defined. The Destroy event occurs for all classes except Printer, Timer, Imagelist, CommandButton, and Menu.

The result of the `□wi` function when it executes the Close method indicates whether the object was successfully closed (1) or not (0). If the object is already closed (not open) when you use the Close method, `□wi` returns 1 (indicating that the object is closed), but the system does not execute the onClose event handler.

After an object is closed, its definition remains in the APL+Win WI system in a closed state. You can reference properties of a closed object to obtain the values they last had while the object was open. For

example, after closing a Form you can reference the `text` property of an Edit control on the form to determine what input the user had entered. You can reopen a closed object by using the `Open` method.

On a Selector control, if you `Close` the currently selected `Page`, the system sets the `value` property of the Selector to zero.

If you invoke the `Close` method on the system object, you exit APL. You can supply arguments to the method to modify the behavior: An argument of zero shuts APL down unequivocally without executing the `onClose` event handler; an argument of 1, the default value, lets the system perform the normal queries as determined by your setting on the Options menu. If the argument is 2, the system suppresses any prompts, but if one would have been given, cancels the exit.

Example:

The example shows that a form does not close when the result of its `onClose` event handler is zero; when `□wres` is 1, the form closes.

```
'fmMain' □wi 'onClose' '□wres ← 0'  
'fmMain' □wi 'Close'  
0
```

```
'fmMain' □wi 'onClose' '□wres ← 1'  
'fmMain' □wi 'Close'  
1
```

CloseDoc

Description:

Ends the printer document if already started and suppresses automatic opening of the printer document.

Class: Printer.

Remarks:

See the description for the `CloseDoc` method in the description for the Printer class for further details.

Command

Description:

Sends a multimedia command to an MCI device in a Media control.

Class: Media.

Remarks:

The `Command` method applies only to an MCI device player. The argument is a character vector. For example, you can send `'set door open'` to a `cdaudio` device to eject the CD, and `'play repeat'` to a `digitalvideo` (AVIVideo) device. See the documentation for the device or the Microsoft documentation for the Media Control Interface for a list of possible commands.

The result is a character vector with the response from the device if no error occurred, or a two-element nested vector with the integer error code and a character vector error message if an error occurred.

ContextHelpMode

Description:

Generates the pointer-plus-question mark cursor and cedes control of the cursor to Windows; the user can then click on a control, which triggers the Help event.

Classes: Form and MDIForm.

Remarks:

This method causes the same effect as a user clicking the "?" button in the title bar of a form, when that button exists; see the `border` property for details. The cursor changes to a normal pointer augmented by a large question mark, and Windows acquires the cursor. When the user next clicks on a control, the Help event fires. See the description of the `onHelp` event handler. If the first click is on a top-level form but not on a control, the Help event does not fire and `MouseDown` does fire.

This action does not change the pointer property of any APL+Win object; you can invoke this method from a menu item or a command bar button to provide the context help capability without giving up the minimize and maximize buttons from the form's title bar.

Copy

Description:

Copies highlighted text from a Combo, Edit, or RichEdit control's input window or a specified subarray from the image property of a Picture control to the Windows clipboard.

Classes: Combo, Edit, Picture, and RichEdit.

Remarks:

For Combo, Edit, and RichEdit, this is the standard text processing action. The control's `translate` property determines whether the text is translated from APL to ANSI (default) or left untranslated.

For a Picture, the argument to Copy is a four-element numeric vector. The first two elements are the row and column of the upper left corner of the subarray you want to copy, measured in current scaling units for the control. The third and fourth arguments are the height and width of the subarray you want to copy. If you do not specify a subarray, the system copies the entire image.

You can move bitmaps between applications using the Windows clipboard and the Copy and Paste methods. You should use these methods carefully, because the clipboard is a shared resource. If you use these methods in an application, you could overwrite something the user placed on the clipboard, and you risk having the image overwritten between the time you copy something and then paste it.

Alternatively, you can move bitmaps within your APL+Win application using the Get and Put methods. These are equivalent to Copy and Paste but they use protected memory rather than the clipboard.

Create

Description:

Creates a new object, replacing an object with the same name if one exists.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List,

Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, and Tree.

Remarks:

When you use the Create method, you not only create an object, you set some irrevocable properties and characteristics for that object by specifying its class. You specify its name as the left argument to `□wi`. If an object with the name already exists, the system deletes it and all its children, firing the `onDestroy` and `onDelete` event handlers for any of those objects that have either or both defined. In all other regards, using Create is the same as using the New method, the description of which contains more information.

Cut

Description:

Removes highlighted text from a Combo, Edit, or RichEdit control's input window or a specified subarray from the image property of a Picture control saves it to the Windows clipboard.

Classes: Combo, Edit, Picture, and RichEdit.

Remarks:

For Combo, Edit, and RichEdit, this is the standard text processing action. The control's `translate` property determines whether the text is translated from APL to ANSI (default) or left untranslated.

For a Picture, the argument to Cut is a four-element numeric vector. The first two elements are the row and column of the upper left corner of the subarray you want to cut, measured in current scaling units for the control. The third and fourth arguments are the height and width of the subarray you want to cut. If you do not specify a subarray, the system cuts the entire image.

DdeExecute

Description:

Used by a DDE client to send a DDE *Execute* command to a DDE server.

Classes: Edit and Label.

Remarks:

A Label or Edit control that is part of a DDE conversation can use the DdeExecute method to send a command to the server. The argument to DdeExecute is the command to be executed. It is a character vector whose content depends on the command string format recognized by the server. For example, if the server is also an APL+Win application, the command triggers the server's `onDdeExecute` handler with `□warg` as the command's argument.

The result of `□wi` is 1 if the command executes successfully; otherwise the result is 0.

Many applications register themselves as DDE servers on the SYSTEM topic when they start execution, making themselves continuously available for commands. Note that spaces are recognized as characters in the `ddeTopic`.

Example:

The example shows a control that is a client in a DDE conversation with Excel. It sends a command to Excel to open a specific worksheet.


```
'fmClient.edIntRate' 0wi 'ddeTopic' 'EXCEL|SYSTEM'
'fmClient.edIntRate' 0wi 'ddeMode' 'cold'
'fmClient.edIntRate' 0wi 'DdeExecute' '[OPEN("INTRATES.XLS")]'
1
```

DdePoke

Description:

Used by a DDE client to send data to a DDE server.

Classes: Edit and Label.

Remarks:

The DdePoke method is used by an Edit or Label control that is a client in a DDE conversation. It causes the DDE *Poke* operation to occur, wherein a client sends data to the server. The control's `ddeltem` property determines the data item that is sent to the server. If the server is an APL Form object, the system puts the data into an Edit object's `text` property or a Label object's `caption` property.

DdePoke does not take an argument. The result of `0wi` is 1 if the operation was successful, meaning that the server received and accepted the data. Otherwise the result is 0.

DdeRequest

Description:

Used by a DDE client to request data from a DDE server.

Classes: Edit and Label.

Remarks:

DdeRequest is used by an Edit or Label control that is a client in a DDE conversation. Normally this occurs in a conversation where `ddeMode` ← 'cold' (manual link). It request the server to send an immediate update of the data. The Edit control's `text` property or the Label control's `caption` property receives the data from the server. The server's `ddeltem` property determines the data item that the server sends.

DdeRequest does not take an argument. `0wi` returns a 1 if the control receives the requested data correctly; otherwise the result is 0.

Example:

```
:if 'fmClient.edTBill30' 0wi 'DdeRequest'
  RunModel 'fmClient.edTBill30' 0wi 'text'
:else
  ErrorBox '30-Day T-Bill data not available'
:endif
```

DecryptFile

Description:

Decrypts files that were encrypted with the EncryptFile method.

Class: # (the system object).

Remarks:

The DecryptFile method converts an encrypted file into a decrypted file. This method is called as shown below:

```
status ← '#' □ WI 'DecryptFile' filename key  
status ← '#' □ WI 'DecryptFile' filename key filenameTo
```

You may specify one <filename> and <key> as arguments if you want to decrypted the file "in place", replacing the content of the encrypted version of file by the decrypted version of file. You may also include a second <filenameTo> if you want to retain the original file encrypted and create a decrypted version as a separate file with a different name.

Defer

Description:

Queues an expression, function, or event handler for execution, possibly at a later time; or invokes an arbitrary event for deferred execution on an object that is an instance of a user-defined class.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

In its simpler uses, the Defer method takes an argument that is an expression or function you want the system to execute. However, unlike other methods or statements that invoke a function, this does not place the statement to be executed on the stack immediately. The system must become "idle" before it executes the argument.

Becoming idle often means that the system must return from the function that invoked the Defer method before the deferred handler can run. The deferred statement can run before the calling function returns, if the calling function contains some action that allows the system to become idle, such as □wgive, □dl, or a Wait. Note that simply returning from the calling function is not sufficient to trigger the deferred action. All immediately impending actions must complete before the deferred statement is executed. You can use the Defer method effectively in cases where you want to control the sequence of events, for example, when using an onUnfocus event handler to redirect the focus, or using the Close method on a form from a deeply-nested □wi call invoked by the event handler of a child control.

You can also use the Defer method to invoke an arbitrary event on an object. In this case, the argument is more complex. You follow the method name with the handler and the event name (which localizes □wevent), followed optionally by any values you want to have in □warg and □wres. The syntax is:
object_name □wi 'Defer' handler eventname arguments result allow

where object_name is the name of your user-defined control and handler is the expression or function name that is the desired event handler; although the name is specified here, this expression can be different for each instance of your user-defined class. The remaining arguments are optional.

The allow argument is a flag that determines under which circumstances the handler executes. Normally, events are suppressed on closed objects or when the system is suspended, and this is the default behavior for Defer. If you don't specify the allow argument or set it to zero, then the event handler is not triggered

under these conditions. The `allow` flag must be an integer; it can be zero or any combination of the other values:

- 0 Don't allow events if the object is closed or the system is suspended; this is the default value
- 1 Allow events if the object is closed
- 2 Allow events if the object is opening (in transition between closed and open states)
- 4 Allow events if the system is suspended.

The `Defer` method also accepts zero as an argument (in place of the handler). When it encounters zero, it essentially ignores the call. This makes it possible for an undefined user-defined property (Δudp), which would nominally hold the event handler, to be referenced without generating an error.

The technique of using `Defer` (or the `Event` method) to invoke an arbitrary event is particularly important with an object that is an instance of a user-defined class. See the *Creating User-Defined Classes* chapter in this manual for more information.

Delete

Description:

Deletes an object from the APL+Win WI system.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, and Tree.

Remarks:

You invoke this method only by using `□wi`. This method does not occur as a system- or user-generated event. If the object you delete contains children, this method deletes those objects as well.

When you delete an object, you erase it from the APL+Win system. This is a more final action than when you `Close` an object. Unlike the `Close` method, the `Delete` method does not execute the `onClose` event handler for the object. Therefore an object that you delete does not have any opportunity to refuse deletion. Since the object is gone from the APL+Win system, you also cannot reference its properties after the deletion.

Several events occurs when you delete an object. If the object is open, the system closes it without executing its `onClose` event handler. However, the system does execute its `onHide` and `onDestroy` event handlers. In addition, the system executes the `onDestroy` event handlers for any children of the object you deleted. If the object is already closed, the system does not execute either event handler. Objects that the system cannot display on the screen do not have `onHide` or `onDestroy` event handlers; therefore the system cannot execute them. Finally, the system deletes the object from APL+Win memory, firing the `onDelete` event handlers for the object and its children.

With one statement you can use the `Reset` method on the system object to delete everything except the system object from APL+Win memory.

DeleteNodes

Description:

Deletes one or more nodes from a `Tree` object by deleting rows from its list property.

Class: `Tree`.

Remarks:

The argument to this method is an integer-valued scalar or vector of node identifiers. Deleting a node deletes its children.

The result of this method is a vector with one element for each node in the `DeleteNodes` argument. The element is 1 if the node was successfully deleted, 0 otherwise.

DeleteRows

Description:

Deletes one or more items from a `Listview` object by deleting rows from its list property.

Class: `Listview`.

Remarks:

The argument to this method is an integer-valued scalar or vector of `RowIDs`.

The result of this method is a vector with one element for each row in the `DeleteRows` argument. The element is 1 if the row was successfully deleted, 0 otherwise.

DevMode

Description:

Allows you to save or modify the `DEVMODE` structure used in Windows to specify settings for a printer.

Class: `Printer`.

Remarks:

If you invoke this method with no argument, it returns a character vector containing the current `DEVMODE` structure (and associated driver extra bytes). You can save this vector and use it as an argument for another printer or a later session of APL.

You can also use this method to change the printer settings. You should be comfortable using the system function `⌈wcall`, and you must understand that a `DEVMODE` structure contains driver extra bytes after the fixed part of the structure. The `DEVMODE` structure contains a `dmFields` member that identifies which of its other members should be used. You can build a `DEVMODE` structure as a character vector (only). If you supply a character vector argument to the `DevMode` method, the system merges the `DEVMODE` structure represented by the argument with the current settings. Fields that the printer does not support are ignored.

This method gives the low-level programmer the ability to change one or several fields of the `DEVMODE` structure without knowing the current state of all the fields. You could use this, for example, to select paper size or to select between 300 dpi and 600 dpi (assuming the printer supports it).

DevNames

Description:

Allows you to save or modify the `DEVNAMES` structure used in Windows that contains the printer name, printer driver name, and port name.

Class: Printer.

Remarks:

If you invoke this method with no argument, it returns a character vector containing the current `DEVNAMES` structure. You can save this vector and use it as an argument for another printer or a later session of APL. You can also use this method to change the printer settings, by using the system function `□wcall`. Note that if you simply want to change the printer name, you can use the `printername` property on the printer object; you can find the possible printer names by invoking the `Printers` method on the system object.

Draw

Description:

Lets you draw graphics on an APL+Win WI object.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

The Draw method lets you draw graphics on APL+Win WI objects such as forms and frames. You can also use the Draw method with a Printer object to send graphics directly to your printer. For example, you can create a report that includes graphics or a data plot, and print it by using use the Draw method to specify the graphics (and possibly text) on a Printer object.

The syntax of the Draw method is

```
'obj' □wi 'Draw' ( 'op1' arg1 arg2 ... ) ( 'op2' arg1 arg2 ... ) ...
```

where *obj* is an APL+Win WI object such as a form or frame, and *op* is the drawing operation you want the system to perform on the object. The arguments to *op* specify the position, size, or other aspects of how you want Draw to perform the operation. You can generally specify arguments to a drawing operation as a vector, as a nested vector of vectors, as a matrix, or as nested matrices. The table below lists the operations that Draw performs.

Draw Method Operations

Operation	Description	Type
Bitmap	Draws a bitmap.	Rendering
Brush	Sets the brush style Draw uses to fill solid figures.	State
Chord	Draws a figure defined by the intersection of an ellipse with a line segment.	Rendering
Circle	Draws a circle, ellipse, or pie wedge.	Rendering
Clear	Clears the specified object.	Rendering
Clip	Limits the region for Draw rendering operations within the same statement.	State
Color	Sets the colors Draw uses for text and the text background.	State

Operation	Description	Type
<code>Ellipse</code>	Draws an ellipse with arguments similar to <code>Rect</code> .	Rendering
<code>Font</code>	Sets the font used to draw text.	State
<code>Icon</code>	Draws an icon.	Rendering
<code>Imagelist</code>	Draws images from <code>Imagelist</code> objects.	Rendering
<code>Line</code>	Draws connected line segments.	Rendering
<code>Mode</code>	Sets drawing mode.	State
<code>Pen</code>	Sets the pen <code>Draw</code> uses to draw lines and borders.	State
<code>Point</code>	Draws points.	Rendering
<code>Poly</code>	Draws a closed polygon.	Rendering
<code>Rect</code>	Draws a rectangle.	Rendering
<code>Scale</code>	Changes scaling units for <code>Draw</code> operations.	State
<code>Seg</code>	Draws multiple series of connected line segments.	Rendering
<code>Text</code>	Draws text.	Rendering
<code>Window</code>	Defines a new region for <code>Draw</code> rendering operations within the same statement.	State

`Draw` interprets the size and coordinate arguments you specify for an operation in terms of the `scale` property of the object on which you want to draw. For example, if the object has a `scale` property of 5 (pixels), `Draw` interprets the height and width arguments as n pixels high and m pixels wide.

Conventionally, `Draw` uses the upper-left corner of an object's client area as its origin (0, 0); position coordinates increase as you move toward the bottom-right corner. The first value is height and the second is width. You can create coordinate systems in which the upper-left corner is not the origin and the coordinates increase upward and to the left by setting a custom `scale` property where the first element is zero.

There are two types of `Draw` operations, rendering and state. Rendering operations draw a graphic on an APL+Win WI object. `Poly`, `Rect`, and `Text` are examples of rendering operations. State operations set a condition for a rendering operation. You use them to select attributes such as fonts, colors, and brush styles that `Draw` uses when creating a graphic.

Note that state operations are effective only for the statement in which you specify them. Setting a color with the `Color` operation, for example, applies only to rendering operations that follow it in the same statement.

You can prefix certain state-setting operations and three rendering operations, `Bitmap`, `Icon`, and `Text` with a question mark (?). Except for `?Text`, which has a different special meaning, this indicates that rather than performing the specified operation, `Draw` loads the specified object into Windows memory and returns a GDI handle as an item in the explicit result. You can use these handles during subsequent `Draw` operations to optimize your application's performance.

Note: It is important that you eventually release any handles that you obtain using this method by using the `Wfree` function from the `Windows` workspace. Windows allocates these handles from a pool of system resources that are shared by all applications.

The `?Text` operation does not return a handle; instead, it returns a pair of integers that give the height and width the system requires to display the text (in the applicable font and current scaling units).

Normally, the result of the `Draw` method is an empty matrix (0 0 0). However, if there are any handle-request operations, `Draw` returns a vector. Each item of this vector corresponds to one handle request.

Draw does not nest the results. Instead, Draw appends each result item sequentially as a separate item in the Draw result. For example, the following operation requests two bitmap handles and one icon handle:

```
(hb1 hb2 hi1) ← ⍺wi 'Draw' ('?Bitmap' 'bmp1' 'bmp2') ('?Icon' 'icon1')
```

The syntax and a description of each Draw operation are described below. The syntax descriptions use the following abbreviations:

<i>r</i>	Row position of the object to be drawn
<i>c</i>	Column position of the object to be drawn
<i>h</i>	Height of the object to be drawn
<i>w</i>	Width of the object to be drawn
...	A repeat of the previous argument or argument sequence
[]	Optional arguments

You can try the examples given in the descriptions by creating a form and executing the Draw method operation. The examples should work well with a scale of 5 and a size of at least 200 200. You can draw on different objects, such as a Picture or a Frame.

The examples in this section show a form for simplicity. If you set ⍺wself to the name of your form, you can omit the left argument. Use the following statement to clear the object upon which you are drawing:

```
'targetobject' ⍺wi 'Draw' 'Clear'
```

Bitmap

Description:

The Bitmap operation draws one or more bitmaps at the positions and with the sizes you specify.

Syntax:

```
'Bitmap' bitmap [rc [hw]]
'Bitmap' (bitmap [rc [hw]]) (bitmap [rc [hw]]) ...
'Bitmap' (n m ρ bitmap [rc [hw]] bitmap [rc [hw]] ...)
'?Bitmap' bitmap
'?Bitmap' bitmap bitmap ...
```

The *r* and *c* arguments specify the origin of the image. The *h* and *w* arguments determine the amount of space the system reserves for drawing the bitmap image. If the bitmap is larger than this size, the system clips the bitmap to this size. If you omit *h* and *w*, the system draws the bitmap to its full size.

bitmap can be either the name of a bitmap or a numeric bitmap handle you obtain by using the ?Bitmap operation or by using low-level Windows calls. Bitmap names, which are not case-sensitive, refer to bitmap file names or bitmap resources that have been compiled into APL+Win, or resource identifiers in the .DLL or .EXE identified in the Resource DLL entry in APLW.INI.

If you prefix Bitmap with a question mark (?), Draw loads the bitmap(s) you specify and returns their handles in the explicit result of the ⍺wi function that called the Draw method. When you finish, call the Wfree function from the Windows workspace to release the handles ?Bitmap returns.

Examples:

Draw the bitmaps Hello and Goodbye on a form:

```
⍺wself ← 'fmArt'
⍺wi 'Draw' ('Bitmap' ('hello.bmp' 2 4) ('goodbye.bmp' 2 52))
```

Query the bitmap handles using the ?Bitmap operation; then draw the bitmaps Hello and Goodbye:

```
hb ← □wi 'Draw' ('?Bitmap' 'hello.bmp' 'Goodbye.bmp')
'fmArt' □wi 'Draw' ('Bitmap' (hb,2 2050 4 88 4))
```

Brush

Description:

The Brush operation specifies the brush style and color that Draw uses to fill the interior of solid figures. To avoid filling the interior of solid figures, specify a transparent brush.

Syntax:

```
'Brush' style [color]
'Brush' -1 hbrush
'Brush' -2 bitmap
'?Brush' style [color]
'?Brush' -2 bitmap
```

The values allowed that determine the style of brush include:

Style	Description
-2	The name of the bitmap or handle passed as the second argument
-1	The brush handle passed as the second argument
0	A transparent brush; do not specify a color with this brush style.
1	A solid brush; the default
2	Horizontal hatching
3	Vertical hatching
4	Backslash diagonal hatching (upper left to lower right: \\)
5	Slash diagonal hatching (lower left to upper right: //)
6	Cross hatching
7	Diagonal cross hatching

The color value of the brush can be a vector of three numbers (*red, green, blue*) or a scalar (formed by combining the three numbers using the Base Value primitive function: $256 \pm \text{blue, green, red}$).

See the description of the color property in the Windows Interface: Properties chapter. The default value is the background color of the currently selected object.

hbrush is the handle you obtain by using ?Brush.

bitmap is the name of the bitmap or its handle. The system repeatedly tiles this bitmap pattern to create a patterned brush.

If you prefix Brush with a question mark, Draw loads the brush you specify into Windows memory and returns its brush handle, but does not render the bitmap on the screen or the printer. You can use this handle as an argument to subsequent Brush operations. If you create a style -2 brush (bitmap) from a bitmap name, Draw returns a handle value that is a two-item, nested scalar. The scalar consists of the brush handle and the bitmap handle that Draw uses to create the brush. If you create a style -1 brush using a bitmap handle, the result item includes only the brush handle. When you finish using a brush, use the WFree function from the Windows workspace to release the handle(s).

Note: Do not use WFree to release the bitmap handle you use to create a brush until you are through with the brush.

Examples:

Draw a circle and fill it with red, diagonal cross-hatching.

```
'fmArt' □wi 'Draw' ('Brush' 7 255) ('Circle' 50 50 40)
```

Draw a circle and fill it with solid green.

```
'fmArt' □wi 'Draw' ('Brush' 1 (0 255 0)) ('Circle' 50 50 40)
```

Draw a circle and fill it with a pattern from a bitmap.

```
'fmArt' □wi 'Draw' ('Brush' -2 'hello.bmp') ('Circle' 50 50 40)
```

Same as previous example, except using brush handles.

```
hb←□wi 'Draw' ('?Brush' -2 'hello.bmp')
'fmArt' □wi 'Draw' ('Brush' -1 hb) ('Circle' 50 50 40)
```

Cord

Description:

The Chord operation draws a figure defined by the intersection of an ellipse with a line segment. The ellipse is centered at the row and column you specify with the radius you specify. The line segment is drawn between two points on the ellipse determined by start and end. Draw outlines the figure using the currently selected Pen and fills the interior using the currently selected Brush.

Syntax:

```
'Chord' r c radius [aspect [start end [border]]]
'Chord' r c radius [aspect [start end [border]]] ...
'Chord' n m ρ r c radius [aspect [start end [border]]] ...
```

aspect specifies the ratio of the vertical to horizontal axes. An aspect ratio of 1 (the default value) draws a circle. However, if the scaling units of the object you are drawing on are not equal, the circle may appear on the screen as an ellipse. For example, if you draw in Character scaling units (the default), the circle appears to be about twice as tall as it is wide.

If *aspect* is greater than 0, Draw measures the radius along the horizontal axis. If *aspect* is less than 0, Draw measures the radius along the vertical axis. The *start* and *end* arguments specify the starting and ending points of the chord measured counter-clockwise, in degrees, from the right horizontal axis. The default angles are 0 360.

border determines the type of border Draw uses for the graphic. By default, the system draws the circumference using the currently defined Pen. If the current Pen is transparent, the system ignores *border*. *border* is a scalar whose value can be 1, Draw the circumference; 2, Draw the chord; or 3, Draw both the circumference and the chord

Examples:

```
□wself ← 'fmArt' □wi 'New' 'Form' ('scale' 5)
□wi 'Draw' ('Pen' 1 (255 255 0) 4) ('Chord' 100 100 40 2 10 200 1)
□wi 'Draw' ('Pen' 1 65535 4) ('Chord' 100 180 40 2 -20 170 1)
```

Circle

Description:

The Circle operation draws a circle, ellipse, or pie slice centered at the row and column you specify with

the radius you specify. It draws the graphic using the currently selected Pen and fills it using the currently selected Brush.

Syntax:

```
'Circle' r c radius [aspect [start end [border]]]
'Circle' r c radius [aspect [start end [border]]] ...
'Circle' n m ρ r c radius [aspect [start end [border]]] ...
```

aspect specifies the ratio of the lengths of the axes. If *aspect* is greater than 0, Draw uses the specified *radius* along the horizontal axis and multiplies the magnitude of the radius by the aspect ratio to measure the vertical axis. If *aspect* is negative, Draw uses the specified *radius* along the vertical axis and multiplies the magnitude of the radius by the absolute value of the aspect ratio to measure the horizontal axis. An aspect ratio of 1 (the default value) draws a circle if the height and width scaling units are equal.

If the scaling units of the object you are drawing on are not equal, Draw maintains the scaling units along each axis. Under these circumstances, if you use an aspect ratio of 1, the figure appears on the screen as an ellipse. For example, if you draw to the screen in Character scaling units (the default) with an aspect ratio of 1, the figure is twice as tall as it is wide. You can draw a circle with an aspect ratio of 0.5 or -2.

With other scaling units (which you can define with the first element of the *scale* property set to zero) and a positive aspect ratio, Draw also uses the specified *radius* along the horizontal axis in its scaling units and multiplies the radius measured in vertical scaling units by the aspect ratio to define the height of the figure. Similarly, with a negative aspect ratio, Draw uses the specified radius along the vertical axis in its scaling units and multiplies the radius measured in horizontal scaling units by the aspect ratio to define the width of the figure.

The *start* and *end* arguments specify the starting and ending angles of a pie slice measured counter-clockwise, in degrees, from the right horizontal axis. The default angles are 0 360. Note that the calculation of these angles assumes the horizontal and vertical scaling units are equal.

border determines the type of border Draw uses for the graphic. By default, the system draws only the circumference using the currently defined Pen. You can also draw a radial line from the center point to the circumference at the starting angle, ending angle, or both. If the current Pen is transparent, the system ignores *border*. *border* is a scalar that is the sum of any of these values:

Border Values for the Circle Operator

Value	Description
1	Draw the circumference (the default value).
2	Draw a radius at the starting angle.
4	Draw a radius at the ending angle.

Examples:

Clear the form and draw a 90 degree pie wedge:

```
'fmArt' □wi 'Draw' ('Clear') ('Circle' 100 100 50 1 0 90 7)
```

Clear the form and draw a circle with a 90 degree pie wedge taken from it:

```
'fmArt' □wi 'Draw' ('Clear') ('Circle' 100 100 50 1 90 0 7)
```

Clear**Description:**

The Clear operation erases an object and repaints it using the current Brush. If the object is a Printer,

`Clear` causes the page to eject and provides a new page. In the `Printer` context, the current `Brush` is not involved. To use a `Brush` as a background for a printed page, you must use the `Rect` operation to draw a rectangle to the `Printer`, then fill the rectangle with `Brush` values you specify.

Syntax:

```
'Clear'
```

Example:

Clear the form and paint it solid purple:

```
'fmArt' ⍵wi 'Draw' ('Brush' 1 (255 0 255)) ('Clear')
```

Note that `Clear` is a rendering operation; it is necessary in the statement. If you omit the `Clear`, setting `Brush` has no effect. If you place `Clear` before `Brush`, as in the examples under `Circle`, it would erase the object with the default color before evaluating `Brush`.

Clip

Description:

The `Clip` operation is a state operation that limits the region for `Draw` rendering operations within the same statement. Any such operation terminates (is clipped) at the borders of the specified area.

Syntax:

```
'Clip' r c h w
```

```
'Clip' points
```

Example:

Define a 32-by-32 frame; draw a 24-by-24 filled rectangle. `Clip` at row 8 column 8, and redraw the rectangle with a different color. You see less than half of it. (Compare this with the `Window` operation.)

```
'fmArt' ⍵wi 'New' 'Form' ('where' 1 1 10 10) ('scale' 2)
⍵wself ← 'fmArt.lPane' ⍵wi 'New' 'Label' ('caption' '')
⍵wi 'Set' ('where' 1 1 32.8 32.8) ('scale' 2) ('border' 1)
⍵wi 'Draw' ('Brush' 1 140) ('Rect' 0 0 24 24)
⍵wi 'Draw' 'Clear' ('Clip' 8 8 24 24) ('Brush' 1 255) ('Rect' 0 0 24 24)
```

The arguments, *r*, *c*, *h*, *w*, specify the row and column of the upper-left corner and the height and width of the rectangle that forms the new clipping region. You specify these values in terms of the current scaling units for the object. Since `Draw` operations are limited to the client area of the object, the `Clip` operation can only reduce this area. You cannot use `Clip` to draw outside an object.

After the `Clip` operation, all subsequent operations within the same statement are restricted to the specified region. Only those pixels that lie completely within the region are drawn. However, the coordinates of the drawing area are not changed. When you specify a rendering operation, the system draws relative to the same origin as before the `Clip`. If you `Clip` at row 10 and column 10 and `Draw` starting at row 0, column 0, you do not see either the top 10 rows or the leftmost 10 columns of your drawing.

If you specify an *n*-by-2 matrix of points, the system defines a polygon as the clipping region. See the description of the `Poly` operation for details on drawing polygonal figures.

`Clip` operations are not cumulative. If you perform a subsequent `Clip` operation, the system interprets the values relative to the client area of the control and the newly specified region replaces the existing specified region. You can follow a `Window` operation by a `Clip` operation.

Color

Description:

The `Color` operation sets the background and foreground colors for the text. If you do not specify the foreground (text) color, the system defaults to black.

Syntax:

```
'Color' colorBack [colorText]
```

colorBack and *colorText* use composite color values. You specify a composite color value as three color values of 0 to 255 for each of Red, Green, and Blue. For example, bright blue is expressed as 0 0 255, black as 0 0 0, and bright white as 255 255 255. You can express these color values in three ways.

- A three-element integer vector.
- A three-element integer vector enclosed as a scalar.
- A scalar formed by combining the three numbers using the Base Value primitive function:
256 ⊥ blue, green, red. Note the reversed order of the colors.

Example:

Display the text Hello in yellow letters on a blue background and the text Goodbye with the colors reversed.

```
'fmArt' ⌐wi 'Draw' ('Color' (0 0 255) (255 255 0)) ('Text' 'Hello' 20 20)
'fmArt' ⌐wi 'Draw' ('Color' (255 255 0) (0 0 255)) ('Text' 'Goodbye' 40 60)
```

For more information on expressing color values, see the description of the color property in the Windows Interface: Properties chapter in this manual.

Ellipse

Description:

The `Ellipse` operation draws one or more ellipses with the position and size you specify. Draw outlines the ellipse using the current Pen and fills the interior using the current Brush.

Syntax:

```
'Ellipse' r c h w
'Ellipse' (r c h w) (r c h w) ...
'Ellipse' (n m ρ r c h w r c h w ...) ...
```

The required arguments, *r*, *c*, *h*, *w* specify the row and column of the upper-left corner and the height and width of a conceptual rectangle inside which the ellipse is drawn.

Example:

Draw an ellipse and fill it with the color teal:

```
⌐wi 'Draw' ('Brush' 1 (0 255 255)) ('Ellipse' 5 5 50 100)
```

Font

Description:

The `Font` operation specifies the font that Draw uses with the `Text` operation.

Syntax:

```
'Font' name [size [style [charset [angle]]]]
```

```
'Font' hfont
'?Font' name [size [style [charset [angle]]]]
```

The `Font` operation uses the same arguments as the `font` property. `Font` also uses the *angle* argument to specify the inclination of the text, measured counter-clockwise in degrees, from the right horizontal axis. If the argument is a scalar integer, `Draw` treats it as a font handle.

For information on the *fontName*, *fontSize*, *fontStyle* and *fontCharset* arguments, see the description of the `font` property in the Windows Interface: Properties chapter in this manual.

If you prefix `Font` with a question mark (?), `Draw` returns a font handle you can use as an argument for subsequent drawing operations. Using handles is valuable because parsing the regular `Font` argument is time consuming; allocating font handles for fonts that your application uses frequently may speed up processing time for your application. When you finish, call the `Wfree` function from the Windows workspace to release the font handles.

The `?Font` operation also allows you to use scaling units other than those of the current object. To use a different scaling unit, set the `scale` property for the current object to the value you want, use the `?Font` operation to create a font in that scaling unit, and retrieve its handle. You can then change the `scale` property back to the original value and use the font handle. `Draw` displays the text using the font size you specified when you created the font handle regardless of the currently set scaling units.

Example:

Draw the word, Hello, in bold Arial font at a 45 degree angle:

```
'fmArt' ⌵wi 'Draw' ('Font' 'Arial' 80 1 'ansi' 45) ('Text' 'Hello' 120 2)
```

Note that your settings are important for text operations as well. If `scale` is set for characters (1), the corresponding statement would be similar to the following:

```
'fmNew' ⌵wi 'scale' 1
'fmNew' ⌵wi 'Draw' ('Font' 'Arial' 4 1 'ansi' 45) ('Text' 'Hello' 6 1)
```

Icon

Description:

The `Icon` operation draws one or more icons.

Syntax:

```
'Icon' icon [r c]
'Icon' (icon [r c]) (icon [r c]) ...
'?Icon' icon [icon]
```

icon can be either the name of an icon, which is not case-sensitive, or an integer scalar icon handle you obtain by using the `?Icon` operation. Icon names refer to ICO files or icon resources that have been compiled into APL+Win, or resource identifiers in the .DLL or .EXE identified in the Resource DLL entry in

`APLW.INI`. You can create icons with a Windows icon editor; examples include Microsoft's Visual Basic and Visual C++, and Borland's Resource Workshop.

The *r* and *c* arguments specify the row and column position of the upper-left corner of the icon. Icons are fixed in size; you cannot specify a height and width for them. You can use the `?Icon` operation to load icons into Windows memory and to get their handles, which you can use as arguments for subsequent `Icon` operations. When you finish, call `Wfree` from the Windows workspace to release the icon handles.

Example:

Draw the U.S. flag icon included with Microsoft Visual Basic:

```
ⓘwi 'Draw' ('Icon' 'c:\vb\icons\flags\flgusa01.ico' 20 2)
```

Imagelist

Description:

The `Imagelist` operation draws one or more images from one or more specified `Imagelist` objects.

Syntax:

```
'Imagelist' imglist [index [overlay [r c]]]
```

```
'Imagelist' (n m ρ imglist [index [overlay [r c]]] imglist [index [overlay [r c]]] ...)
```

You can specify one, two, three, or five arguments as a vector, or you can specify a matrix with each row having the appropriate structure. *imglist* is the name of an imagelist constructed with the Windows interface; you may specify it relative to the object being drawn upon. *index* is the origin 1 index to an image in the specified imagelist; the default is the first image. *overlay* is the origin 1 index to an overlay image from the same imagelist as specified by its `overlays` property. The overlay default is zero (no overlay).

The *r* and *c* arguments specify the row and column position of the upper-left corner of the image. If you do not specify these arguments, they both default to zero.

Line

Description:

The `Line` operation draws a series of connected line segments.

Syntax:

```
'Line' points
```

```
'Line' points ...
```

The *points* argument is a shape *n*-by-2 matrix of row and column coordinates of the points you want to connect. `Draw` uses the current `Pen` setting to draw the line segments. `Draw` does not close or fill the figure. Use the `Poly` drawing operation to create closed and filled polygons.

Example:

Draw a line with four connected segments, forming a square root symbol:

```
ⓘwi 'Draw' ('Line' (5 2 ρ 20 10 20 16 30 22 14 26 14 70))
```

Mode

Description:

The `Mode` operation specifies several drawing modes that affect how `Draw` performs other operations. You

can specify any one or more of the arguments; Draw does not change any of the *Mode* arguments you specify as zilde ($\emptyset$).

Syntax:

'Mode' *TransparencyMode RasterOpMode PolyFillMode*

TransparencyMode determines whether *Text* operations use a background fill between adjacent characters (1) or if the background shows through (0). If you set *TransparencyMode* to zero (transparent), the text background color assumes the color of the control object behind it. For example, the background color behind the text on a push button is the same color as the push button. If you set *TransparencyMode* for the button to *Opaque*, the text appears on the color you specify.

RasterOpMode sets the mode in which Draw performs graphics operations. Draw operations use the same codes that Microsoft uses for Windows' SetROP2 or for Visual Basic's DrawMode property. For the 16 possible values, see the description of the mode property in the Windows Interface: Properties chapter in this manual.

PolyFillMode determines how Draw fills the interior of figures you draw using the *Poly* operation. The default value is *Alternate* (0); you can set the value to *Winding* (1). If you set *PolyFillMode* to *Alternate*, the system fills only those areas that can be reached from outside the figure by crossing an odd number of lines. If you set *PolyFillMode* to *Winding*, the system fills all interior areas in the polygon. See the description of the mode property in the Windows Interface: Properties chapter in this manual for examples of the fill mode settings.

Pen

Description:

Pen specifies how Draw traces the borders of filled figures and draws line segments and points.

Syntax:

'Pen' *style* [*color*] [*size*]
'Pen' -1 *hpen*
'?Pen' *style* [*color*] [*size*]

The *size* argument specifies the width of the pen in pixels. The table below shows the valid values for *style*. You can specify values greater than 1 only when the value of *size* is 1. If *size* is greater than 1, Draw automatically sets the value for *style* to 1.

Valid Values of Style for the Pen Operation

Value	Description
-1	The next argument specifies a GDI pen handle
0	A transparent pen; do not specify color or size with this value
1	Solid, the default
2	Solid inside
3	Dash
4	Dot
5	Dash-dot
6	Dash-dot-dot

You can set the *color* of the pen in the same manner you define the *Color* operation. The default value is the current foreground color in the color property. You can use the ?Pen operation to obtain a pen handle

which you can use as an argument for subsequent `Pen` operations. Use the `Wfree` function from the Windows workspace to release any pen handle you obtain.

Example:

Draw a bright blue dashed line.

```
'fmArt' □wi 'Draw' ('Pen' 3 (0 0 255) 1)
↳ ('Line' (5 2 ρ 20 5 120 50 60 90 120 125 80 155))
```

Point

Description:

`Point` draws one or more points; each point equals one pixel.

Syntax:

```
'Point' r c [color]
'Point' (n m ρ r c [color] r c [color] ...)
```

The argument to `Point` is an n -by-2 matrix of row and column coordinates of the points you want to draw or an n -by-3 matrix when you specify the *color*. If you omit the third column, `Draw` displays the points using the current foreground color.

Example:

Clear an object and draw two points on it:

```
'fmArt' □wi 'Draw' ('Clear') ('Point' (2 2 ρ 40 40 100 100))
```

Note that individual pixels can be hard to see. Patterns are easily visible, however; for example, try:

```
'fmArt' □wi 'Draw' ('Point' (⊃(2/`40+3×119)))
```

Poly

Description:

The `Poly` operation draws a polygon.

Syntax:

```
'Poly' points
```

The *points* argument is an n -by-2 matrix of row and column coordinates of the vertices of the figure. `Draw` draws the line segments using the current `Pen` and fills the polygon using the current `Brush`. `Draw` creates an extra line segment between the starting and ending coordinates to close the polygon. To create an open figure, use the `Line` drawing operation.

Example:

Draw a polygon and fill it with yellow cross-hatching:

```
'fmArt' □wi 'Draw' ('Brush' 7 (255 255 0))
↳ ('Poly' (5 2 ρ 50 10 10 30 60 100 50 190 140 60))
```


Rect

Description:

The `Rect` operation draws one or more rectangles with the position and size you specify. Draw outlines the rectangle using the current `Pen` and fills the interior using the current `Brush`.

Syntax:

```
'Rect' r c h w [eh ew]
'Rect' (r c h w [eh ew]) (r c h w [eh ew]) ...
'Rect' (n m ρ r c h w [eh ew] ...) (r c h w [eh ew]) ...
```

The required arguments, *r*, *c*, *h*, *w*, specify the row and column of the upper left corner of the rectangle and its height and width. The optional arguments, *eh* and *ew*, specify the height and width of elliptical curves you can use to draw rounded corners. If you omit *eh* and *ew*, or specify them as zero, Draw creates a rectangle with square corners.

Example:

Draw two overlapping rectangles, one with rounded corners:

```
'fmArt' □wi 'Draw' ('Rect' (25 25 25 100) (12 12 50 50 10 20 ) )
```

Scale

Description:

The `Scale` operation changes the scaling units for Draw operations for the control. You can use `Scale` to differentiate the scaling units you use for Draw operations and the scaling units you use for other actions on the same control.

Syntax:

```
'Scale' scalemode
'Scale' [0] [r c] h w
```

The first element of `Scale` is *scalemode*. Its default value is the current value of the `scale` property for the object you are drawing on. If *scalemode* is zero, you can set `scale` as a two-element, four-element, or five-element vector. The last four elements of `scale` indicate the position (row, column of the upper left corner) and the size (height and width) of the object's client area in scaled units. See the `scale` property for more information.

The possible values for *scalemode* include:

Values for Scale Operation *scalemode*

Value	Description
0	Proportional scaling (defined in terms of 1 ÷ <i>scale</i>).
1	Characters (1 sys-font high; 1 sys-font wide).
2	Dialog units (1/8 sys-font high; 1/4 sys-font wide); the scaling units used by external dialog box editors.
3	Points (72 points per logical inch).
4	Twips (1440 twips per logical inch).
5	Pixels.

Seg

Description:

The `Seg` operation draws multiple independent series of connected line segments. (See the `Line` operation to draw one series of connected line segments.) `Seg` draws m connected line segments, each with $(n \div 2) - 1$ segments. Each point is specified by a pair of values, so n must be even, or you get a `LENGTH ERROR`.

Syntax:

```
'Seg' (m n ρ points)
```

`Draw` uses the current `Pen` setting to draw the line segments. `Draw` does not close or fill the figure. Use the `Poly` drawing operation to create closed and filled polygons.

Example:

Draw two lines, each with two connected segments:

```
'fmArt' ⌐wi 'Draw' ('Seg' (2 6 ρ 80 5 120 40 120 160 120 5 80 30 60 160))
```

Text

Description:

The `Text` operation draws text using the currently selected font at the position you specify. The text you want to draw can be a character vector, including `⌐tcnl` characters to create a multiple line display, or it can consist of or contain integer values representing Unicode character designations with appropriate fonts.

Syntax:

```
'Text' textstring r c [h w]
'Text' (textstring r c [h w]) ...
'?Text' textstring
'?Text' textstring [textstring]
```

The r and c arguments specify the location where you start to display the text. To limit the text to within a certain clipping area, you can use the optional h and w arguments. However, if you specify a nonzero angle with `Font` or include a `⌐tcnl` in `textstring`, `Draw` may not display all of your text when you use the h and w arguments. You can use the `?Text` operation to determine how large an area `Draw` needs to display the text you specify without truncation.

Examples:

Draw the word `Hello` in red letters on a white background:

```
'fmArt' ⌐wi 'Draw' ('Color' (255 255 255) 255) ('Text' 'Hello' 7 2)
```

Return the height and width values needed to display text in a specified font:

```
res←'fmArt' ⌐wi 'Draw' ('Font' 'Arial' 50 1 'ansi')('?Text' 'Goodbye')
      res
49 183
```

Unicode Arguments to the Draw Method Text Operation:

Windows 2000 and newer supports Unicode designations in some contexts. It seems apparent that Microsoft is moving in the direction of relying more on Unicode designations. Therefore, `APL+Win` allows numeric Unicode and `⌐UCS` character designations in the `textstring` portion of the argument to the `Text` operation of the `Draw` method.

Integer Method

Henceforth, *textstring* may still be a homogeneous character vector, or it may be a homogeneous numeric vector of Unicode designations, or a heterogeneous vector of characters and numeric Unicode designations. You can use this facility if you need to produce output containing characters outside the normal set provided by Windows.

The Unicode standard assigns unique numeric codes to text characters and symbols found in most world languages. Note that this does NOT mean that all these characters are available in a font on any given machine. You must have installed a font that includes all the characters you want to output. However, many of the standard Microsoft TrueType fonts support a few hundred characters outside the Windows ANSI set. Office 2000 contains several fonts that support non-western languages and one very large font that provides most of the Unicode definition. If a designated font does not contain the proper glyph for a specified character, the system draws an empty box or some blob in lieu of a character.

As part of the design of this feature, the interpretation of some character data in the argument to the `Text` operation of the `Draw` method may change. Some of this behavior is established; some of the design is yet to be established. The design questions consider the numeric designations of characters as they exist in the APL atomic vector (`⎕av`), Unicode, ASCII and Windows ANSI as well as positions in APLPLUS fonts and other fonts. The following is known:

The interpretation of character arguments in a homogeneous character vector is unchanged. Such arguments are interpreted according to the `translate` property. The interpretation of character arguments in a heterogeneous array is changing. However, for the commonly used characters, any change will be transparent. Those characters that fall with the range of `⎕av[32]` through `⎕av[126]` (except for `⎕av[124]`, the split style which sometimes becomes a solid vertical bar), have the same designation in `⎕av`, Unicode, ASCII and Windows ANSI.

Thus, those characters that are numerals, alphabetic characters in the English language, common punctuation marks, and most symbols found on any common English-language keyboard are interpreted consistently and as expected.

Warning: This facility is still being implemented. The interpretation of character arguments for symbols in the upper half of `⎕av`, which now generate characters that depend on the designated font, may change in the future. The following provides some information about those characters whose interpretation is still subject to definition.

In the range 160-255, the numerical assignments for Unicode match those for Windows ANSI. Many of the characters in APL fonts and in `⎕av` match those assignments, but some of these positions are APL symbols whose Unicode designation is outside the range that is currently accessible.

The range 128-159 is an area that was formerly reserved for control codes and is not used by Unicode for particular symbols. Some of these values represent symbols both in `⎕av` and Windows ANSI, but the symbols do not match. Some applications in Windows also use some of these positions for special effects, for example, to trade the so-called "smart quotes" for single direction quotes, and results may vary according to Windows quirks.

The range 0-31 is an area that is reserved for control codes and is not used by Unicode for particular symbols. Some of these values represent control codes in APL, for example `⎕tcnul`, but some of these values contain APL symbols or European characters that have different designations in Windows ANSI and Unicode.

Position `⎕av[127]` represents `⎕tcdel`, which matches the traditional designation for the delete character (control code).

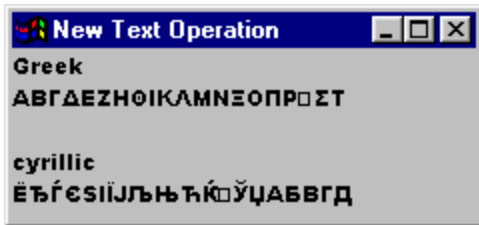
Example:

The following function creates a form and draws Greek and Cyrillic characters on it using Unicode numbers in the argument to the Draw method.

```

▽ Newtext
[1] 'f' ⚡wi 'Create' 'Form' ('where' 0 0 7 30)
[2] 'f' ⚡wi 'Set' ('caption' 'New Text Operation')('font' 'Arial Black')
[3] 'f' ⚡wi 'Show'
[4]
[5] 'f' ⚡wi 'Draw' ( 'text' ( (1 5 ρ 71 114 101 101 107) 0 0 ) )
[6] 'f' ⚡wi 'Draw' ( 'text' ( (1 20 ρ 912+ι20) 1 0 ) )
[7]
[8] 'f' ⚡wi 'Draw' ( 'text' ( (1 8 ρ 'cyri', 108 108 105 99) 3 0 ) )
[9] 'f' ⚡wi 'Draw' ( 'text' ( (1 20 ρ 1024+ι20) 4 0 ) )
▽

```

**⚡UCS Method**

The Draw method's Text operation also supports ⚡UCS character in exactly the same way as it previously understood integer arguments. However, unlike integers, ⚡UCS characters can be used alone without specifying coordinates like this:

```
⚡WI 'Draw' ('Text' (⚡UCS 97 98 99))
```

in the same way as it previously understood character data:

```
⚡WI 'Draw' ('Text' 'abc')
```

Whereas with integer data you must specify coordinates such as this:

```
⚡WI 'Draw' ('Text' (97 98 99) 0 0)
```

This change to the Draw method works for all object types that accept Draw method, including the Printer.

For best results when printing ⚡UCS characters, it's recommended that you set the font operation for the Draw method to a Unicode font.

Window**Description:**

The Window operation is a state operation that defines a new region for Draw rendering operations within

the same statement. Any such operation is drawn relative to the upper left corner of the new region defined by Window.

Syntax:

```
'Window' r c h w
```

The required arguments, *r*, *c*, *h*, *w*, specify the row and column of the upper-left corner and the height and width of the rectangle that forms the new drawing region. You specify these values in terms of the current scaling units for the object. Since Draw operations are limited to the client area of the object, the Window operation can only reduce this area. You cannot use Window to draw outside an object.

After the Window operation, all subsequent rendering operations within the same statement are drawn relative to the upper left corner of the newly specified region; that is, the origin changes. If you draw something starting at row 0, column 0, you are drawing in the visible area of the newly restricted region.

Window operations are not cumulative. If you perform a subsequent Window operation, the system interprets the values relative to the client area of the control and the newly specified region replaces the existing specified region. The origin changes to match the newly specified region. You can, however, follow a Window operation by a Clip operation.

Examples:

Define a 32-by-32 label; draw a 24-by-24 filled rectangle. Window at row 8 column 8, and redraw the rectangle with a different color. You see the entire rectangle in a new position. (Compare this with Clip.)

```
'fmArt' □wi 'New' 'Form' ('where' 1 1 10 10) ('scale' 2)
□wself ←'fmArt.lPane' □wi 'New' 'Label' ('caption' '')
□wi 'Set' ('where' 1 1 32.8 32.8) ('scale' 2) ('border' 1)
□wi 'Draw' ('Brush' 1 140) ('Rect' 0 0 24 24)
□wi 'Draw' 'Clear' ('Window' 8 8 24 24) ('Brush' 1 255) ('Rect' 0 0 24 24)
```

Eject

Description:

Ejects a page without closing the print job.

Class: Printer.

Remarks:

The Eject method resets the position property for the current Printer object to the origin (0 0). The page property does not change until you use the Print or Draw method on a new page.

For the Printer object, you can use an optional argument for the Eject method. Specifying

```
□wi 'Eject' 1
```

forces a page to eject even if nothing has yet been printed on the page. Using this argument does NOT open a document if it is not already open. An argument of 0 (zero) gives the old behavior, wherein a page does not eject unless something has opened the page, e.g., using the Print or the Draw method.

Note that eject refers to the page in the Printer object. You may not see the physical page until you close the printer object or something forces the print spooler to disgorge its contents. In the first sequence below, the y's are on page 2; in the second sequence, they appear on page 3.

```
'p' □wi 'Print' 'xxxxx'
'p' □wi 'Eject'
'p' □wi 'Eject'
```

```

      'p' □wi 'Print' 'YYYYY'
      'p' □wi 'Close'
1
      'p' □wi 'Print' 'xxxxxx'
      'p' □wi 'Eject'
      'p' □wi 'Eject' 1
      'p' □wi 'Print' 'YYYYY'
      'p' □wi 'Close'

1

```

EncryptFile

Description:

Encrypt any file.

Class: # (the system object).

Remarks:

The EncryptFile method encrypts a file that can later be decrypted via the DecryptFile method.

This method is called as shown below:

```

status ← '#' □WI 'EncryptFile' filename key
status ← '#' □WI 'EncryptFile' filename key filenameTo
status ← '#' □WI 'EncryptFile' filename key filenameTo offset
status ← '#' □WI 'EncryptFile' filename key filenameTo offset note

```

You can do in-place encryption (replacing the original file with an encrypted version) by specifying a single <filename> and <key>. You can also specify the second <filenameTo> as " as a place holder if you want to specify the additional offset or note arguments. In either case the file is encrypted in-place. Or you can specify the <filenameTo> with a real file name to encrypted into a different file without replacing the original.

The <offset> argument specifies a starting offset in the file where encryption should start. The content of the file before this point is not encrypted and remains as a clear text header. This is how log files are handled with a human readable prefix that contains only summary information followed by an encrypted section containing the potentially sensitive information.

The <note> specifies an optional note you want inserted as a prefix above the encrypted section. This is different than an <offset> of unencrypted content in that the note will disappear when the file is decrypted whereas the text that comes before the offset will remain in the decrypted file. By default, the system automatically places a default "note" ahead of the encrypted section that reads:

```

***** BEGIN ENCRYPTION *****
***
***          PLEASE DO NOT RESAVE THIS FILE          ***
***
*** Doing so could destroy its encryption making it ***
*** impossible to decrypt. This file is only useful ***
*** to the application vendor if transmitted VERBATIM ***
***
*** Only the vendor noted below can decrypt this file ***
***
***** BEGIN ENCRYPTION *****

```

```

Your Company Name
www.YourCompany.com/support?ref=10773
support@YourCompany.com
(987)000-0000

```

The company contact information comes from the [ContactInfo] setting of the INI file. See below in another section that talks about the Context setting and its other uses in error messages and log files.

You can supply a note that augments the default encryption note shown above with your own note or completely replace or eliminate the note. This is controlled by the first character of the note.

If the first character is '=' your note completely replaces the system note. The '=' is removed and your note is specified by the remaining characters following it. If you don't want any note at all specify '=' without any text following. If your note is not empty, the system suffixes it with a newline (CR+LF) before the start of encryption. This is added in addition to any newlines that might already appear at the end of your note.

If the first character is '+' your note works as a prefix before the standard system note. The '+' is removed and your note is specified by the remaining characters following it. If your note is not empty, the system suffixes your note with a newline (CR+LF) before starting the standard system generated part of the note. This is added in addition to any newlines that might already appear at the end of your note.

If you don't specify either '=' or '+' as the first character then the content of your note is used

as a prefix before the standard system note. This is similar to the behavior of a '+' prefixed note. Thus, the only reason you ever need to specify '+' as a prefix character is when you want '=' to appear as the first character of your note. In that case you can specify "+= the rest of your note". Otherwise, the '+' prefix doesn't do anything different than not using a prefix at all.

Both methods return a status code rather than signaling an error for some issues. But they do signal errors for file issues such as file not found, file already exists, file access errors, etc.

DecryptFile also signals an error if the key does not work for the target file. The status codes are:

+2 = This only applied to DecryptFile. The file was not encrypted by APL+Win and remains as is. File corruption could cause this to happen if the file has changed so that it no longer contains the "signature" features of a file encrypted by APL+Win.

+1 = File was successfully encrypted or decrypted

-1 = The onCrypto event cancelled. The file remain unchanged in the case of an in-place operation. If <filenameTo> specifies an output file, that file is not created.

-2 = This only applied to DecryptFile. The file was decrypted but is damaged (checksum mismatch detected).

If this is being done in place (i.e., the <filenameTo> argument was not specified or specified as ""), an error is signaled instead and the conversion is not performed. You can only accept a damaged conversion if you specify the non-empty <filenameTo> argument.

If the password does not match a "CRYPTO KEY ERROR" is signaled. In this case the system imposes a built-in 15 second delay before it will allow DecryptFile to be called again. This does not delay the return of the failed decryption attempt (which returns immediately upon discovering the key doesn't work for the target file). The delay occurs the next time you call DecryptFile. If it has been less than 15 seconds since the failure, it enters a delay loop waiting until the 15 second period has expired. This delay is intended to make automated password cracking less effective. During this delay, the onCrypt event (see below) is fired periodically to indicate how many seconds of delay remain.

If file appears to be encrypted by APL+Win but something is wrong with layout of internal checks then "CRYPTO CORRUPTION ERROR" is signaled. This could be caused if the encrypted file was loaded by a text editor and then resaved by that editor. It could also be caused by a transmission error that corrupted the file.

If res would be -2 (checksum mismatch) we do not allow overwriting the original file for in-place decryptions signal "CRYPTO CHECKSUM ERROR".

Encryption and decryption buffer overflows signal "CRYPTO INTERNAL ERROR".

File system errors signal a "CRYPTO FILE ERROR: 0xNNNNNNNN(n) ..."

EncryptLog

Description:

Character vector that specifies the encryption key for the active workspace.

Class: # (the system object).

Remarks:

By default, log and MiniDump files are not encrypted. You can only set its value. For security reasons, you cannot query its value. The most recently set value is saved with the workspace.

EnsureVisible

Description:

Ensures that an item of a Listview or the node of a Tree is visible in the control.

Class: Listview and Tree.

Remarks:

You can use this method to ensure that a specific item is visible. You can use this only when the control is open; if the control is closed when you invoke this method, the system signals an error.

For a Listview, the argument is a one- or two-element vector. The first element is the RowID of the item you want the user to see; the second element is a flag that specifies whether it must be fully visible (1) or can be partly visible (0). If you do not specify the second element, it defaults to 1. Some systems may force the item to be fully visible even if you specify zero. The system returns 1 if the operation was successful.

For a Tree, the argument to this method is the identifier of the node you want to make visible. The system expands the parent node if necessary and scrolls the tree to make the node visible. The system returns 1 if the parent was already expanded but the node was scrolled to bring it within the visible portion of the tree; it returns zero otherwise.

dEnumEnd

Description:

Ends the enumeration of a collection in an ActiveObject and frees the internally cached interface that was returned by the EnumStart method.

Class: ActiveObject.

Remarks:

This method requires no argument; in general, you do not need to invoke this explicitly, because the system invokes it implicitly when you reach the end of enumerating a collection or when you release the collection.

EnumNext

Description:

Iterates over the next n or fewer items from a collection in an ActiveObject and returns a vector of interface pointers, which you can use to create APL objects that represent the items of the collection.

Class: ActiveObject.

Remarks:

This method (along with EnumStart and perhaps EnumEnd) allows you to enumerate collections in ActiveX Objects that don't have a built-in property that allows you to access members of the collection. If your collection has an Item property and a Count property (that show up as xItem and xCount in the properties list), then you probably don't need these methods.

This method allows a non-zero integer argument; if you do not supply an argument, it defaults to 1. It returns a vector of interface pointers, which you can use to create APL objects that represent the items of the collection. The shape of the vector will be the number you supply as an argument if there are enough items remaining in the collection. You can use the pointers, for example, to reference the name property of each

item to identify it. When the method has enumerated all items in the collection, it does an implicit EnumEnd, and returns an empty result until you call EnumStart again.

EnumStart

Description:

Begins enumerating a collection in an ActiveObject from the start.

Class: ActiveObject.

Remarks:

This method requires no argument; you must call it before you can use EnumNext. The result is zero on a failure and a non-zero interface pointer on success. You do not need to capture this pointer unless you want to access the collection using `EnumCall`, but you do need to use the method any time the collection has been refreshed.

Event

Description:

Invokes an arbitrary event on an object.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

You follow the method name with the handler and the event name (which localizes `Event`), followed optionally by any values you want to have in `EventArgs` and `EventArgsList`. The syntax is:

object_name `Event` *handler* *eventname* *arguments* *result* *allow*

where *object_name* is the name of your control and *handler* is the expression or function name that is the desired event handler; although the name is specified here, this expression can be different for each instance of the class. The remaining arguments are optional.

The *allow* argument is a flag that determines under which circumstances the handler executes. Normally, events are suppressed on closed objects or when the system is suspended, and this is the default behavior for Event. If you don't specify the *allow* argument or set it to zero, then the event handler is not triggered under these conditions. The *allow* flag must be an integer; it can be zero or have any combination of the other values:

- 0 Don't allow events if the object is closed or the system is suspended; this is the default value
- 1 Allow events if the object is closed
- 2 Allow events if the object is opening (in transition between closed and open states)
- 4 Allow events if the system is suspended.

The Event method also accepts zero as an argument (in place of the handler). When it encounters zero, it essentially ignores the call. This makes it possible for an undefined user-defined property (*undefined*), which would nominally hold the event handler, to be referenced without generating an error.

The technique of using Event (or the Defer method) to invoke an arbitrary event is particularly important with an object that is an instance of a user-defined class. See the Creating User-Defined Classes chapter in this manual for more information.

Exec

Description:

Executes a character vector in the context of an object whose name the system variable `⌵wself` specifies.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, and Tree.

Remarks:

This method exists in this system primarily to support easier migration of applications built using APL+DOS versions of the APL graphical user interface.

In APL+Win, you can obtain the same effect by localizing `⌵wself` and assigning the name of an object to establish a context for execution; for example:

```
⌵wself←'fmMain.ed'  
⌵wi 'text' 'Name'
```

This has exactly the same effect as writing a statement using the Exec method:

```
'fmMain.ed' ⌵wi 'Exec' '⌵wi 'text' 'Name''
```

The standard syntax for the same effect is the following statement:

```
'fmMain.ed' ⌵wi 'text' 'Name'
```

You may find the Exec method, or setting `⌵wself`, useful if you execute several `⌵wi` operations on the same object, using `⌵wself` as an implicit left argument to each operation.

If the code that the Exec argument executes stores a value in the APL+Win global variable `⌵wres`, the system returns this value as the explicit result of `⌵wi`. The default result of `⌵wi` is `0 0 0`.

Expand

Description:

Controls the display of a node of a Tree object and its children.

Class: Tree.

Remarks:

The argument is a two-element nested vector. The possible first arguments, and their effects, are shown below; the second argument is the numeric node identifier. This method returns a 1 if successful; zero if unsuccessful, for example if you attempt to expand a node with no children or one that is already expanded.

Operations for the Expand Method

Operation	Result
'expand'	Opens the children for display.
'collapse'	Hides the children so they do not or will not display.

Operation	Result
'toggle'	Switches between expanded and collapsed.
'expandbranch'	Opens all descendants for display.
'collapsebranch'	Hides all descendants so they will not display. (See Note 1.)
'collapsereset'	Removes all descendants. (See Note 2.)

Note 1: If you collapse a node, its children are hidden but not necessarily collapsed, so that when you re-expand the node and display it, the node's grandchildren may or may not display. If you use the `collapsebranch` operation, the system collapses all the descendants of the selected node. If you then re-expand the node, its children are all collapsed so only the next generation displays.

Note 2: The `collapsereset` operation is intended for use with the `forceparent` state and the `onGetVirtualData` event handler to allow you to add children to the tree only when they are needed for display. If neither the `?forceparent` nor the `forceparent` state is set, the system sets `forceparent` when you use `collapsereset`.

FindNode

Description:

Identifies a node that has certain characteristics in a `Tree` object.

Class: `Tree`.

Remarks:

The argument is a two-element nested vector. The first argument is a character vector containing one of the operations shown below. The second argument for those that require it is the node identifier. This method returns a node identifier or zero if none qualify.

Operations for the FindNode Method

Operation	Node Identifier Returned
'root'	First top-level node (no second argument)
'parent'	Parent node
'child'	First child node
'next'	Next sibling node
'prev'	Previous sibling node
'children'	All children of the node (returns a numeric vector)
'firstvisible'	First visible node (no second argument)
'nextvisible'	Next visible node
'prevvisible'	Previous visible node
'drop'	Drag and Drop target (no second argument)
'select'	Currently selected node (no second argument)

Focus

Description:

Shifts the input focus to an object.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, List, Listview, MDIForm, Media, Option, RichEdit, Scroll, Selector, Spinner, Trackbar, and Tree.

Remarks:

The Focus method provides a way to specify the input focus for an object under program control. You can also set the focus property of the system object to give a control the focus. Under program control, you can determine which object has the focus by referencing the focus property of the system object. Note that Frame, Label, Menu, Picture, Status, and Toolbox objects cannot have the focus. When you click on a menu item, the focus does not change.

FormatMessage

Description:

Formats error codes into message strings.

Class: # (the system object).

Remarks:

This method is primarily useful decoding any standard error message number (such as returned by the GetLastError low-level Windows API function) including the error property for the Printer object.

Get

Description:

Returns a rectangular subarray of the image property of a Picture control.

Class: Picture.

Remarks:

The argument to Get is a four-element vector. The first two elements define the row and column of the upper left corner of the portion of the image that you want to specify measured in current scaling units for the control. The third and fourth elements specify the height and width of the subarray. The result is a matrix of color bits whose shape is determined by the number of pixels in the subarray.

You can use this method in conjunction with the Put method to examine or modify a portion of a graphical image without referencing the image property (for example, when you specify the image by the name of the bitmap).

Example:

The following commands take the center quarter of a 32-pixel bitmap and replace it with solid red.

```
array ← 'fmForm.picA' ⍵wi 'Get' 8 8 16 16  
array ← 255 + array × 0  
'fmForm.picA' ⍵wi 'Put' array 8 8
```

GetCells

Description:

References the cell values in specific rows of a Listview object.

Class: Listview.

Remarks:

The first argument is an integer vector of RowIDs. The second argument is an integer vector of column indices for the cells you want to reference. The result is a nested matrix of character vectors containing cell values with one row for each RowID you specify and one column for each column index you specify.

GetImages

Description:

References the image indices in specific rows of a Listview object.

Class: Listview.

Remarks:

The argument is an integer vector of RowIDs. The result is a three-column matrix with the image index values for the specified rows.

GetInfo

Description:

References the specification for one or more nodes of a `Tree` object.

Class: `Tree`.

Remarks:

This method can have two or three arguments. The first argument is one of the operation names detailed below. The second argument is a scalar or vector of identifiers of the nodes you want to query. You can also specify one node with some or all of its children by using a third argument. If you specify a third argument, you may specify only one node in the second argument.

The third argument specifies the depth of children to return; 0 means no children (the result will have 1 row), 1 means just the node's immediate children, 2 means children and grandchildren, etc.; you can specify `-1` for all descendants. The result returned is a vector or matrix depending on the information requested. If you specify the depth, the result is a matrix that includes the node identifiers as the first column, depth as the second, and the requested information in subsequent columns.

You can use the following operations: `'list'`, `'depth'`, `'label'`, `'images'`, `'checkstate'`, `'highlightbold'`, `'highlightcut'`, `'highlightdrop'`, `'highlightselect'`, `'expanded'`, `'expandedonce'`, `'forceparent'`, `'haschildren'`, `'wherelabel'`, `'whererow'`, or `'handle'`.

- The `'list'` operation returns a matrix with one row for each node and one column for each of elements 3, 4, and 5 from the list property: label, image indices, and states.
- The `'depth'` operation returns a vector of integers containing the depths for the nodes from element 2 of the list property.
- The `'label'` operation returns a vector of character vectors containing the text labels for the nodes.
- The `'images'` operation returns a matrix with one column for each of the non-selected, selected, overlay, and user state image indexes.
- The `'checkstate'` operation returns a vector showing the user state image index for each checkbox: 3 if checked, 2 if not checked. .
- The `'highlightbold'`, `'highlightcut'`, `'highlightdrop'`, `'highlightselect'`, `'expanded'`, and `'expandedonce'` operations each return a Boolean vector denoting the specified state of the nodes.
- The `'forceparent'` operation returns an integer vector with the force-parent state of each node. A 0 means normal: the behavior of the node is determined by whether the node has children. A 1 means the node behaves as if it has children whether or not it really does. A negative one (`-1`) means the `GetVirtualData` event is triggered to determine how the node behaves.
- The `'haschildren'` operation returns a Boolean vector identifying whether or not each node has children. (This is not affected by the `forceparent` state; it is based on the existence of child nodes.)
- The `'wherelabel'` and `'whererow'` operations return a numeric matrix with the position and size of each node's label or entire row. The matrix has four columns containing the row and column coordinates and height and width, respectively, of the appropriate part of the node in the client area of the `Tree` object using its current scaling units.
- If a node is not visible within the client area of the `Tree`, but could be scrolled into view without expanding any node, the result reflects the node's current position within the scrollable region relative to the upper left corner of the client area (thus the position may be negative or larger than the `size` property of the `Tree`). If any of a node's ancestors is collapsed, that row of the result is all zeros. If the `Tree` is closed, an error is signaled.

- The 'handle' operation is for low-level programming; it returns the handle used internally by the control to identify the node. Certain operations may change this value (e.g. re-opening the Tree), so this value should be retrieved each time it is needed rather than stored in an APL variable.

GetPos

Description:

References the exact positions of items in largeimage or smallimage view in a Listview object.

Class: Listview.

Remarks:

In largeimage view, this method applies to the poslargeimage property, and in smallimage view, it applies to the possmallimage property. The argument to GetPos is a vector of integer RowIDs; it returns a two-column matrix of the row and column coordinates of the corresponding items in the current scaling units of the control. You can use this method only when the Listview is in one of the appropriate viewmodes.

GetRows

Description:

References the values in specific rows of a Listview object.

Class: Listview.

Remarks:

The argument is an integer vector of RowIDs. The result is a matrix with the same form as the list property.

GetSession

Description:

Reference the contents of an open editor session window.

Classes: # (the system object).

Remarks:

The syntax is:

```
'#' □wi 'GetSession' sessionid
```

The argument, sessionid, may be either a character vector representation of an open editor session name or a singleton integer representing an origin 1 index into the result of the sessions property. This method returns the contents of an editor session window designated by sessionid.

Help

Description:

Executes the onHelp event handler for an object or for an object in its parentage chain.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

The Help event fires when a user presses the F1 key; clicks the "?" button in the title bar of a form or MDIForm and then clicks on a control; or when the application invokes the ContextHelpMode method, which invokes the same behavior as clicking a "?" button, and the user then clicks on a control. The event fires on the target object, that is, the one has the focus when the F1 key is pressed or that is clicked with the pointer-plus-question-mark cursor, if its onHelp event handler is defined. If the handler is not defined on the target object, the system searches up the parentage chain for an object that does have the handler defined. When it finds one, the event fires on that object; if the event handler sets `Wres` to zero, the system continues searching up the parentage chain, including the system object, for another onHelp event handler.

Hide

Description:

Removes an object from the screen by changing the visible property.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

You invoke this method only by using `□wi`. This method does not occur as a system- or user-generated event. If the object you hide contains other objects, this method hides those objects as well. The Hide method leaves negative numbers in the visible property unchanged or negates a positive number.

When you invoke Hide, the system triggers the `onHide` event handler for the object, if one exists. The event handler runs after the object disappears. When you Hide an object, it still exists in the Windows environment. If you later display the object by using the Show method or by setting its visible property to 1, the object has the same property values and appears as it did before it was hidden, including if it was maximized or minimized.

Hide differs from Close because the object remains open while it is hidden. This difference can be significant, for example, if you hide a Form that contains a Timer object as a child. If you close the Form, the system also closes the child Timer object and stops executing `onTimer` event handlers for it. If you Hide the Form, however, the system does not close the form and its children. Therefore, the child Timer object continues executing `onTimer` event handlers even though the Form is no longer visible on the screen.

If the object was already open and not visible on the screen, this method has no effect other than to trigger the `onHide` event handler. If the object was not open, invoking Hide also invokes the Open method to open the object, which triggers the `onOpen` and `onResize` event handlers for the object. It also triggers the `onOpen` event handlers for any objects that this object contains. The system then executes the `onHide` event handlers for the object and its children.

When the system executes an `onHide` event handler, `□warg` contains the argument value passed to `□wi` following the 'Hide' action name. If the `onHide` event handler sets `□wres` during execution, `□wi` returns its value as an explicit result. Normally `□wi` returns (0 0 ρ 0) as its result.

If you invoke Hide on a form that is waiting, you break the Wait loop.

HitTest

Description:

Defines the location of a row and column coordinate position in terms of a Listview item, a Tree node, a visible calendar of a DateTime control, a Status bar pane, a Selector tab or button, or a CommandBar child.

Classes: CommandBar, DateTime, Listview, Selector, Status, and Tree.

Remarks:

The argument to this method is a two-element numeric vector that specifies a row and column in the current scaling units. For most classes, the method returns a two-element vector. For Listview, it returns a three-element vector containing the item id under the cursor, where on the node the cursor is, and the column id under the cursor (meaningful only for report view).

For the other classes, the first element is the identifier of the child, date, page, pane, or node if the location is on a child, date, page (tab or button area only), pane, or node; it is blank or zero otherwise. The second element is a character vector containing one of the applicable location values provided by Windows.

For a Listview, the second element can be empty or one of the following: image, label, imageuser, above, below, toleft, toright, aboveleft, aboveright, belowleft, belowright, nowhere.

For a Tree, the second element can be one of the following: image, button, label, imageuser, indent, noderight, above, below, toleft, toright, aboveleft, aboveright, belowleft, belowright, nowhere, or an empty vector.

For a DateTime control, the first element is a date, if the location is on a date; otherwise it is empty. The second element can be: calendarbkgn, calendardate, calendardatenext, calendardateprev, calendarday, calendarweeknum, titlebkgn, titlebuttonnext, titlebuttonprev, titlemonth, titleyear, todaylink, nowhere, or empty (if the control reports an unknown value). The values that include "prev" and "next" refer to the partial months preceding and following the months displayed in full. The "todaylink" entry refers to the display of today's date below the calendar.

For a Selector, the second element can be 'image', 'label', 'tab', 'nowhere', or empty. The HitTest method on a Selector returns 'nowhere' for the working area of its child pages.

For a Status, the second element can be 'pane', 'grip', 'nowhere', or empty.

For a CommandBar, the second element can be 'caption', 'grip', 'child', 'nowhere', or empty.

HwndToName

Description:

Returns the APL name of the object whose window handle you supply as the argument.

Class: # (the system object).

Remarks:

This method is primarily useful in low-level Windows API programming where the handle would be obtained using `□wcall`. If the argument is not a window handle or is the handle of a window associated with a different application, the result is an empty vector. If the argument is the handle of an unnamed child

of an APL+Win WI object, such as the temporary edit window in which the user can edit the label of a node of a Tree or an item of a Listview, the result is the parent or closest named ancestor of the window.

Info

Description:

Returns a single list of the names of the applicable methods, properties, and event-handler properties for that class.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

Each `□wi` class now has an `Info` method. The syntax is:

```
object □wi 'Info' flags pattern
```

You can use some combination of the letters, 'm', 'p', and 'e' as flags to include methods, properties, and event-handler properties respectively. Invoking the `Info` method with no argument is equivalent to specifying all three letters. Invoking `□wi 'Info' 'm'` or `□wi 'Info' 'p'` provides the same list as invoking `□wi 'methods'` or `□wi 'properties'`. Invoking `□wi 'Info' 'e'` returns a list of the event-handler property names, including the `on` prefix, as differentiated from the event names that you get with `□wi 'events'`.

Note that the list is returned alphabetized in the APL style, so the method names with an initial uppercase letter appear first. The event-handler property names appear among the regular properties.

When the object is an ActiveX object, you can also specify either of the letters 'a' or 'b' as flags to limit the list to the ActiveX actions (methods, properties, event-handler properties) or the built-in actions (that is, the APL+Win properties, methods and event-handler properties that are associated with the object). If you specify neither flag, you get all the actions.

When the object is an ActiveX object, you can also specify the letter 'x' as a flag to request extra ActiveX information. In this case, the display is a matrix of information: the first column is the name of the action; there are three columns of codes that tell you whether the action is a method, property, or event-handler property, and whether it is an ActiveX or a built-in action; and the fifth column gives the VTABLE offset of the ActiveX action or zero if there is no table or if it is a `dispinterface` action that has no such offset.

If you want the default list, but you want to specify a pattern, use an empty character vector (' ') for the *flags* argument.

You can specify a pattern in a manner similar to that you use for `XInfo`. If you specify letters, the list contains those actions that start with the specified letters; if you specify an asterisk followed by letters, the list contains all actions that include the string anywhere in the name.

InsertNodes

Description:

Adds nodes to a Tree object by adding rows to its list property.

Class: Tree.

Remarks:

This method requires three arguments. To insert a single node:

- The first argument is the numeric identifier of the parent node. If the node you want to insert is a top-level node, use the character string 'root' as the first argument.
- The second argument is the numeric identifier of the sibling node (node on the same level) that physically precedes the new node in the tree. You can also use the character strings 'first' or 'last' to place the new node at the beginning or end, respectively, of its group of siblings, or you can use the character string 'sort' as the second argument. This causes the system to place the new node in alphabetical order, assuming the previously inserted nodes are already in alphabetical order. If the previous nodes are not alphabetized, the position of the new node is undefined. **Note:** If you are using virtual data for labels, you should not use this method when the tree is closed.
- The third argument is a nested vector containing the structure for one row of the list property. You can use zero for the node identifier and allow the system to assign a unique value; you can omit the last two columns if you want to assign only the depth and the label.

You can insert multiple nodes in one statement by making the third argument a nested matrix with one row for each new node and one column for each element of the structure for the list property. You can use zero for the node identifier and allow the system to assign unique node identifiers. You can omit the last two columns if you wish to define only the depths and labels.

Note: If you use a matrix to insert multiple nodes and you specify 'first', 'last', or 'sort' as the second argument, the placement argument applies only to nodes at the same depth as the node in the first row of the matrix. The system inserts children of the highest-level new nodes in the order in which they appear in the matrix.

InsertRows

Description:

Inserts items into a Listview object by adding rows to its list property.

Class: Listview.

Remarks:

This method requires two arguments. The first argument is the RowID of the item you want to precede the new rows. The second argument is a matrix that has the same form as the list property. The result is an integer vector of the inserted RowIDs.

LineToChar

Description:

Returns the position for the first character on a specific line.

Classes: Edit and RichEdit.

Remarks:

The LineToChar method takes a scalar integer argument that is the origin-zero line number in the control. It returns the position of the first character on that line; that is, the number of characters that precede the character in the text property. This is the same value you would find in the first element of the selection property if you selected that character or placed the selection cursor in front of it. Using -1 as the argument returns the position of the first character on the line containing the cursor.

Modify

Description:

Changes the modified property of the object, which invokes the ripple effect on modified properties in the parentage tree.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, and Tree.

Remarks:

If you invoke Modify with no argument or an argument of 1, the system changes the modified property of the current object to 1. If you use an argument of zero, the system changes the modified property of the current object to zero. If you use an argument of -1, the system changes the modified property of the current object to zero only if the modified property has the value 1; if this object was changed by virtue of a child being changed, its modified property is not reset.

New

Description:

Creates a new object.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, and Tree.

Remarks:

When you use the New method, you not only create an object, you set some irrevocable properties and characteristics for that object by specifying its class. You specify the class as the second element of the right argument to `□wi`, following the method name. You specify the name of the object as the left argument to `□wi`. The way that you form the name determines whether it is a top-level object or whether another object contains it. If an object with the name already exists, the system signals an error. For example, to create a frame named fColor on an already existing form named fmSample, you specify:

```
'fmSample.fColor' □wi 'New' 'Frame'
```

You can also specify other properties in the same call by appending nested vectors to the right argument.

```
'fmSample.fColor' ⍵wi 'New' 'Frame' ('where' 5 5 30 20)
↳ ('caption' 'Available colors')
```

The list of items following the class name is called an action-list. The action-list can contain pairs of property names and their values as shown above; it can also contain method names with optional arguments. Items in the action-list have the same syntax and semantics as the right argument to `⍵wi` when you set properties; you cannot reference a property to determine its value in a statement that uses the `New` method. If you use relative-name prefixes in the actions, they are based relative to the object the `New` method creates. The result of the function is the fully-qualified name of the object.

When you create an object with the `New` method, the system automatically opens the object after it process all members of the action-list, unless one of the actions in the list is the `Close` method.

An important and powerful extension to the `New` method allows you to clone objects. First, you create an object-definition vector by referencing the `def` property of another object. Then you use the `New` method with the object-definition vector in place of the class-name argument. Note that the object-definition vector may be arbitrarily complex. It can define a single object or a whole collection of objects, such as a `Form` containing many controls along with all their handler properties.

The following example creates a form named `fmOne` that contains a button named `bnOk`; it then replicates two more instances of the form (and its button) named `fmTwo` and `fmThree` but changes their captions to "Form Two" and "Form Three" and sets their `where` properties to display them in different positions:

```
'fmOne' ⍵wi 'New' 'Form' ('caption' 'Form One') ('where' 1 1 5 20)
'fmOne.bnOk' ⍵wi 'New' 'Button' ('caption' 'Ok') ('onClick' '⍵self ⍵event')
fmOne_def ← 'fmOne' ⍵wi 'def'
'fmTwo' ⍵wi 'New' fmOne_def ('caption' 'Form Two') ('where' 1 25)
'fmThree' ⍵wi 'New' fmOne_def ('caption' 'Form Three') ('where' 1 50)
```

The `onClick` event handler defined for the `fmOne.bnOk` control prints the names of the control and the event whenever it is clicked. Through the cloning of the `fmOne_def` object-definition vector, this handler also works for `fmTwo` and `fmThree` forms as well. Depending upon which button you click you'll get one of the following three responses from the original and two cloned forms:

```
fmOne.bnOk Click
fmTwo.bnOk Click
fmThree.bnOk Click
```

Note that the three forms are completely independent even though they are clones. Deleting `fmOne` does not affect `fmTwo` or `fmThree`. Also note that although the `fmOne.bnOk` button was cloned from form `fmOne`, its names on the two cloned forms are `fmTwo.bnOk` and `fmThree.bnOk`.

You also use the `New` method with ActiveX objects. You can use any one of three identifiers: the descriptive name, the globally unique identifier, or the programmatic id as the argument. For an `ActiveControl`, this information is in the `xclasses` array; for an `ActiveObject`, you can find the information using the `XInfo` method on the system object. You can also use a pointer acquired, for example, by using a method on the parent of the new object that identifies or invokes the child. (See the *Using ActiveX in APL+Win* chapter in this manual for more details.)

Notify

Description:

When this method is invoked on an instance of APL that is an ActiveX server, it triggers an onNotify event handler on the client; specifically on the client's object that represents the server.

Class: # (the system object).

Remarks:

Although this method exists on the system object for any instance of APL, it has meaning only when you are using APL as an ActiveX server object. You use this method to send a notification to the client at a time determined under the control of the APL server code. In addition to the server-to-client notification, this mechanism provides the capability for a return acknowledgement (or the passing of other information) from the client to the server by setting the third element of `□wres` in the onNotify event handler. See the APL+Win as an ActiveX Server chapter in this manual for more information.

Open

Description:

Brings an object into existence in the Windows environment making it responsive to user input, DDE interactions, and other stimuli by enabling its event-handler properties.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, and Tree.

Remarks:

Using the Open method makes an object alive and ready for action. When an object is open, it can respond to user input, DDE interactions, or program actions by means of its event handler properties. Under almost all circumstances, the system can execute an object's event handlers only when the object is open. Before you can open an object, you must define it in the APL+Win system using the New or Create method.

If the object is a form or a control, it may become visible on the screen (depending upon the visible property) as a result of Open. For utility objects such as a Timer, the system starts executing the appropriate event handlers (depending on the setting of the object's enabled property).

If you use the Open method on an object that is already open, it has no effect. The object remains open, and the system does not execute any event handlers. If you use the Open method on an object that is closed, the following sequence of events occurs:

1. The system opens the object. If the object is a form or a control, it remains hidden when first opened. Once the object is open, the system executes its onOpen event handler. During execution of this event handler, `□warg` contains the value of any extra parameters passed with the Open method. For example, if you invoked `□wi` with
`'fmMain' □wi 'Open' 1 2 'xyz' 4`
`□warg` has the value `1 2 'xyz' 4`. If you set the value of `□wres` during execution of the onOpen event handler, the system returns its value as the explicit result of `□wi`. Otherwise it returns the default value of `(0 0 ρ 0)`.
2. The system executes the onResize event handler for the object.

3. If the object has children, the system opens each child object in the sequence specified by their order properties. As it opens each child object, it executes that object's `onOpen` event handler. The value of `□warg` is `0` while the child objects' handlers run.
4. If the object is a form or control and its `visible` property is greater than zero, the system displays the object along with any of its children that have their `visible` properties set for display.

The `Hide`, `Print`, `Show`, and `Wait` methods automatically invoke the `Open` method.

OpenDoc

Description:

Opens and starts the printer document for printing.

Class: Printer.

Remarks:

See the description for the `OpenDoc` method in the description for the `Printer` class for further details.

PageSetup

Description:

Displays a common dialog box for configuring the page description for a printer.

Class: Printer.

Remarks:

When you invoke `PageSetup`, the system displays a standard dialog box where you can select the page size, source, orientation (portrait or landscape), and the four page margins. You can also invoke another dialog box that allows you to select a different printer. This method returns `1` if the user presses OK on the dialog box, zero if Cancel.

Optionally, you can supply one or two arguments to this method. The first tailors the dialog box and the second allows you to specify minimum values for the margins in terms of scaling units for the printer object. The first argument is a scalar that is the sum of values from the table below, or a nested vector of those values; if you supply a vector, enclose it in parentheses. You must supply this argument if you want to specify minimum margins.

If you provide the second argument (minimum margins), it must be a four-element vector specifying, in current scaling units for the printer, the minimum sizes of the left, right, top and bottom margins of the page. These values are converted by the system to the units displayed in the dialog box; users do not see the scaling unit values.

Values for Tailoring the Page Setup Dialog Box

Value	Description
0	Default box; use this value if you want to specify only minimum margins.
1	Disable margins control.
2	Disable orientation control.
4	Disable paper size and source control.
8	Disable printer selection control.
16	Display margin units in inches (not with 32).

32	Display margin units in millimeters (not with 16).
65536	Hide the network button on the printer selection control.
131072	Suppress warning when there is no default printer.

Example:

This example invokes a dialog box that allows the user only to set the margins in millimeters but leaves the minimum margins at the default values for the printer driver.

```
'prt' □wi 'PageSetup' (2 4 8 32)
```

See also the descriptions of the Setup and PrintDlg methods for similar configuration methods.

Paint

Description:

Repaints an open, visible object on the screen.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

Use the Paint method when you want to force a repaint of an open object. Normally, when property values change, the system automatically updates the display if necessary. The Paint method triggers the onPaint event handler, if one exists.

PaneWhere

Description:

Returns a four-column matrix that shows the location and size of one or more panes in a status control.

Class: Status.

Remarks:

Each row of the matrix is equivalent to the where property in current scaling units for a pane of the status bar. If you do not supply an argument, there is one row for each pane defined in the status property of the bar. Panes that are not currently visible return values of -1 in all four columns. You may supply an argument of one or more pane identification numbers.

Paste

Description:

Copies text from the Windows clipboard into the input window of a Combo, Edit, or RichEdit control at the insertion point, replacing highlighted text, if any; or copies a bitmap from the Windows clipboard to the image property of a Picture control.

Classes: Combo, Edit, Picture, and RichEdit.

Remarks:

For Combo, Edit, and RichEdit, this is the standard text processing action. The copied text remains

highlighted. The control's `translate` property determines whether the text is translated from ANSI to APL (default) or left untranslated.

For a `Picture`, the argument to `Paste` is a two-element numeric vector that specifies the row and column in the `image` property of the upper left corner of the area you want to replace, measured in current scaling units for the control. The height and width of the area replaced are determined by the size of the bitmap on the clipboard.

Play

Description:

Starts playing a stopped file or a new file that was not started automatically.

Class: `Media`.

Remarks:

You can use this method to restart, or start, playing a file. Optionally, you can include an argument of up to three elements. The first element specifies a starting location; the second element specifies an ending location; the third element specifies the number of repetitions for the animated cursor player or the Animation player. The third element is ignored by an MCI player.

The starting and ending locations are frame numbers, starting with zero, for the players that play `.ANI` and `.AVI` files exclusively. For MCI devices, the units differ depending on the device; they might be in seconds for example. Each of the elements can be a non-negative integer, or you can specify `-1`. For the starting position, this specifies the current position; for the ending position, it specifies the end of the file. For the third element, it means repeat indefinitely, (except for the MCI player). You can make an MCI player repeat by using the `Command` method with the `'play repeat'` argument.

Pop

Description:

Restores the `image` property of a `Picture` control with a graphical image you saved in memory using the `Push` method.

Class: `Picture`.

Remarks:

You can use this method in conjunction with the `Push` method to save and restore a graphical image quickly; the save and restoration are done more quickly than using `Get` and `Put`. You cannot, however, modify the image that you save with `Push`; nor can you `Push` the image from one `Picture` object and `Pop` it to another.

The argument to `Pop` is a two-element vector that specifies the row and column of the upper left corner in the `image` property of the area you want to replace.

Example:

```
'fmMain.picArt' ⌵wi 'Pop' 24 10
```

Popup

Description:

Displays a floating popup menu from which a user can make a selection. You can use this method to provide a control-specific menu that is not part of the menu bar for the form.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

You can invoke this method in response, for example, to a MouseUp event on any control; you can also invoke it in response to a DropDown event on a style 8 command button.

The children of the menu you use as the first argument to the method appear at a location you specify. When the user selects a menu item, presses Esc, or clicks somewhere other than on the control that invokes this method, the menu disappears. If the user selects a menu item by clicking or highlighting an item and pressing Enter, that action triggers the onClick event handler for the highlighted menu item unless you choose to return the name of the menu item instead as the result of the method.

The argument to this method requires the fully-qualified name of the menu followed by optional arguments that specify the location of the menu. The syntax for the Popup method includes all the following:

```
'object'  □wi  'Popup'  'menu'
'object'  □wi  'Popup'  'menu'  'exclude_object'
'object'  □wi  'Popup'  'menu'  alignment_value
'object'  □wi  'Popup'  'menu'  'exclude_object'  alignment_value
'object'  □wi  'Popup'  'menu'  alignment_value  'exclude_object'
'object'  □wi  'Popup'  'menu'  alignment_value  posRow  posCol
'object'  □wi  'Popup'  'menu'  alignment_value  posRow  posCol  exHeight  exWidth
```

Where:

- *object* is the name of the object on which the Popup method is invoked
- *menu* is the fully-qualified or relative name of the menu you wish to display. A popup menu can be a top-level object; that is, a child of the system object. It can also be a child of a form or of another menu item that is the child of a form. For a popup menu to be effective, it must have children, because only the children appear in response to the Popup method.
- *exclude_object* is the name of a □wi object that defines an exclusion rectangle which the menu does not overlap when displayed;
- *alignment_value* is the sum of five values explained in the table below;
- *posRow* and *posCol* define an explicit point (instead of the cursor location) on which to align the menu or define the exclusion rectangle. The point is relative to the where location of *object*;
- *exHeight* and *exWidth* define the size of the exclusion rectangle below and to the right of the position specified by *posRow* and *posCol*.

By default, the system positions the menu relative to the cursor. You can specify, in addition to the name of the menu, an alignment value that changes the corner or edge of the menu that aligns relative to the cursor; or you can specify an explicit position (relative to the object on which the Popup method is invoked) on which the menu aligns.

Additional options allow you to specify an area where the menu does NOT appear; that is, an area from which the menu is excluded. This makes menu placement easier in two ways. First, you can specify an area coincident with an object from which the menu is excluded; this forces the menu, by default, immediately below the object. (This object can be the one on which the Popup method is invoked or any other object on the screen.) This technique is particularly effective for drop-down command buttons; you can specify the button itself as the excluded area and the menu appears immediately below it when the user clicks the drop-down arrow that is part of the style 8 command button.

The second benefit occurs when the location of the menu is close to an edge of the screen; if there is not enough room for the menu to display normally, the system adjusts its position away from the edge. If you have specified an exclusion area, the system displays the menu above or beside the rectangle, but not overlapping it. If you specify an explicit point but not the size of the rectangle, and you are close to the bottom of the screen, the menu may overlap an important region anyway.

Popup Method Alignment Values

Value	Description
0	Default: Put the top-left corner of the menu at the cursor or point; the menu responds only to the left mouse button.
2	Make the menu respond also to the right mouse button.
Horizontal alignment; select zero or one of the values.	
4	Center the top of the menu on the cursor or point.
8	Put the right side of the menu at the cursor or point.
Vertical Alignment; select zero or one of the values.	
16	Center the side of the menu at the cursor or point.
32	Put the bottom of the menu at the cursor or point.
Display and Processing Options; select zero, either or both.	
64	Vertical alignment matters more than horizontal alignment. (See note 1.)
256	Return the name of selected menu item or ' ' if canceled. (See note 2.)

Note 1: Normally horizontal alignment matters more than vertical alignment. This is important only when you have specified an exclusion object or rectangle and the menu cannot be placed in the desired alignment position specified for Popup. This setting determines the preferred alternative alignment of the menu relative to the exclusion rectangle.

Note 2: When you specify 256 the name of the selected menu item is returned rather than firing the onClick event handler on the selected menu. If no menu item is selected (that is, the user cancels menu mode) this option causes ' ' to be returned.

The alignment values work in combination; they allow nine placement positions relative to the cursor. You can add 2 to any of the other values to activate the right mouse button for selecting a menu item. The system does not prevent you from selecting meaningless combinations (such as 4 and 8). Higher numbers override lower numbers. When you specify an exclusion rectangle, the edges or center of the menu align with the edges or center of the rectangle. (Since the rectangle is excluded, the value 20 is meaningless; specified relative to the cursor, it centers the menu both vertically and horizontally.)

If you invoke the Popup method in response to a mouse click on a very large object, such as your main form, you should query `EventArgs` in the event handler and display the menu only in response to the non-default mouse button. Then, if the user makes a normal click elsewhere on the form, the menu will disappear rather than being persistent. Alternatively, see the example event handler below.

Note that the seven different versions of the syntax are not very different. If you use the `where` property of the `exclude_object` to specify the exclusion rectangle numerically, the seventh line is exactly the same as the

fifth. When you specify *exclude_object* as a name, it can come before or after the alignment value, or you can have either or neither. The sixth line specifies a null exclusion rectangle, which aligns the menu at a point specified by *posRow posCol* rather than at the cursor location.

Examples:

The first example displays a menu immediately below the command button for which the `onDropDown` event handler is fired. Note that you can use relative naming to refer to the object on which the method is invoked or its children when specifying the name of the menu or the excluded object.

In the second example, the `Popup` method is invoked on a form but specifies a button below which the popup menu should be displayed. In the third example, the menu displays to the left of the named object. The fourth example shows an explicit exclusion rectangle and the menu responding to the right mouse click as well as the left. You could use the `Pop` function for an `onMouseDown` or `onMouseUp` event handler for a form to display a popup menu that is not irritatingly persistent.

```
1. 'form.bar.cmdbtn8' □wi 'onDropDown' "□wi 'Popup' 'menu' '.' "
```

```
2. 'form' □wi 'Popup' 'menu' '.button'
```

```
3. 'fm.obj1' □wi 'onMouseDown' "□wi 'Popup' 'fm.menu' 24 'fm.obj2' "
```

```
4. 'fm.obj1' □wi 'onMouseDown' "□wi 'Popup' 'fm.menu' 2 4 4 2 8"
```

```
▽ Pop menu
[1] A Display popup menu (but only if not already running)
[2] :if 0=menu □wi 'ΔPop_running'
[3]     menu □wi 'ΔPop_running' 1
[4]     □wi 'Popup' menu
[5]     menu □wi 'ΔPop_running' 0
[6] :endif
▽
```

Print

Description:

Prints text in the current font.

Class: Printer.

Remarks:

The `Print` method causes a printer you specify to print the text of the argument in the current font at the location you specify with the `position` property. The argument to the method is a character vector with embedded New Line (`□tcnl`) or Form Feed (`□tcff`) characters. This method opens the Printer object if it is not open.

The `Print` method also supports `□UCS` or integer data (UCS indexes) in the range 0-65535. This is NOT full Unicode (it's `□UCS`). So you can therefore print like this:

```
'printer' □WI 'Print' 'abc'
```

or

```
'printer' □WI 'Print' (□UCS 97 98 99)
```

or

```
'printer' □WI 'Print' 97 98 99
```

For best results when printing □UCS characters, it's recommended that you set the font property for the Printer object a Unicode font.

PrintDlg

Description:

Displays a common dialog box for configuring a printer.

Class: Printer.

Remarks:

When you invoke PrintDlg, the system displays a standard dialog box where you can select a printer, whether to print to a file, the print range, the number of copies, and whether to collate. You can also invoke another dialog box that allows you access to other printer properties. The exact nature of this dialog depends on the printer driver and the operating system. This method returns a zero if the user cancels the dialog box; it returns additional information about the selection if the user presses OK (see below).

Optionally, you can supply one or two arguments to this method. The first tailors the dialog box and the second allows you to specify the range of possible page numbers to be printed. The first argument is a scalar that is the sum of values from the table below, or a nested vector of those values; if you supply a vector, enclose it in parentheses. You must supply this argument if you want to specify the second argument.

The second argument (for page ranges) is a four-element vector specifying the minimum and maximum allowable page numbers to be printed and the initial values in the "from" and "to" boxes in the dialog. You must supply this argument if you want the user to be able to select a page range. If you do not supply this argument, it is the same as disabling the pages option by including 32 in the first argument.

Values for Tailoring the Print Dialog Dialog Box

Value	Description
Print Range: Select only one value	
0	Default box; "All" is initially selected as print range.
1	"Selection" is initially selected as print range.
2	"Pages" is initially selected as print range.
Tailoring Options: Select any meaningful combination of values	
4	"Print to file" is initially checked.
8	"Collate" is initially checked.
16	Disable "Selection" option (not with 1).
32	Disable "Pages" option (not with 2).
64	Disable "Print to file" check box.
128	Hide "Print to file" check box (not with 4).
256	Display "Copies" and "Collate" options even if unsupported (see below).
65536	Hide the network button on the printer selection control.
131072	Suppress warning when there is no default printer.

This method returns a zero if the user cancels the dialog box; if the user presses OK, it returns various values depending on the print range selected and whether 256 was included in the first argument. If 256 was not included, the method returns a scalar 1 if "All" was selected or a scalar 2 if "Selection" was the print range.

If "Pages" was selected, the method returns a two-element vector; the first element is 3 (for Pages) and the second element is a two-element nested vector showing the first and last page numbers of the page range.

If 256 was included in the first argument, the method returns a four-element vector. The first element is as above, 1 for "All," 2 for "Selection," and 3 for "Pages." The second element is an enclosed zilde for a first element of 1 or 2; a two-element nested vector of the selected page range if the first element is 3.

The third and fourth elements are useful only when the user requests multiple copies and the printer driver does not support multiple copies or collation. The third element shows the number of copies for which the application is responsible, and the fourth element is 1 if the application must collate the pages.

These two elements work in conjunction with the `copies` and `collate` properties. If the user selected only one copy, both the `copies` property and the third element of the return are 1. If the user selected multiple copies and the driver supports this, the third element is 1 and the `copies` property shows the number of copies the driver will print. If the driver does not support multiple copies, the `copies` property is 1, and the third element shows the number of copies requested.

Similarly, if the user did not request collation, both the `collate` property and the fourth element are zero. If the user requested collation and the driver supports this, the fourth element is zero and the `collate` property is 1, indicating that the driver will handle the collation. If the driver does not support collation, the `collate` property is zero and the fourth element is 1, showing that the user requested collation that the printer driver cannot provide.

A request to collate has meaning only if there are multiple copies and multiple pages. If the driver does not support collation, but would support multiple uncollated copies, presumably Microsoft's print dialog treats the driver as though it supports neither.

Caution: You should invoke `PrintDlg` before you begin to construct the print job, so the values the user sets are established. New values established by the Print Dialog will not take into account any pages that were started before the dialog was invoked.

See also the descriptions of the `PageSetup` and `Setup` methods for similar configuration methods.

Printers

Description:

Returns a list of the printer drivers installed on the local machine; returns or sets the current default; or searches a network for printers that are accessible.

Class: # (the system object).

Remarks:

If you invoke this method without an argument, the result is a two-column matrix showing the names of any printer drivers and the names of the corresponding ports. These drivers can represent printers that are physically attached to the local machine as well as remote printers to which the machine has a network connection.

If you follow the method name with the argument `'default'`, the method returns the current default printer. If you follow the method name with the arguments `'default'` and a character vector that contains the name of one of the installed printers, that printer becomes the default for the system.

```
'#' 0wi 'Printers' 'default'  
HP LaserJet 4/4M PostScript LPT1:  
'#' 0wi 'Printers' 'default' 'Brother MFC-4500'
```


If you follow the method name with the operation name 'Tree', you can do an iterative search for all the printers that are accessible from the host machine. The method returns a four-column matrix from the `PRINTER_INFO_1` structure showing the Name, Comment, and Description fields and a 32-element Boolean vector of status flags.

The information you see depends on the operating system, what kind of networking you have access to, if any, and possibly what applications or what versions of applications you have installed. If you simply use 'Tree', the name field shows the top-level print providers. This shows the names and descriptions provided on one machine.

```
(1 2 3) (1 3) 0 '#' 0wi 'Printers' 'Tree'
Windows NT Local Print Provider Locally connected Printers
Windows NT Remote Printers      Remote Printers
Windows NT Internet Provider    Internet URL Printers
```

You can use one of the names from the first column to search the next level. Note that Microsoft misspells "Provider" in some systems. You must use their neologism. This shows the names and comments of some of the local drivers on that same machine.

```
(1 4 7) (1 2) 0 '#' 0wi 'Printers' 'Tree' 'Windows NT Local Print Provider'
Linotronic 330      Linotronic 330,AdobePS Linotronic 330,
HP LaserJet 4/4M PS HP LaserJet 4/4M PS,HP LaserJet 4/4M PS,Bethesda, MD
Fax                Fax,Windows NT Fax Driver,
```

This shows the names, comments and description of two of the accessible remote printer containers.

```
(9 21) (1 2 3) 0 '#' 0wi 'Printers' 'Tree' 'Windows NT Remote Printers'
Windows NT Remote Printers!COGNOS COGNOS Logged on Domain
Windows NT Remote Printers!NAMERICA NAMERICA Logged on Domain
```

This shows the size of the next level of the tree. If you are on a large network and do not have access to many of the printers, this step can take a significant amount of time to complete.

```
0 '#' 0wi 'Printers' 'Tree' 'Windows NT Remote Printers!NAMERICA'
266 4
```

Push

Description:

Stores a copy of all or part of the image property of a Picture control in memory, but not on the clipboard.

Class: Picture.

Remarks:

You can use this method in conjunction with the Pop method to save and restore a graphical image quickly; the save and restoration are done more quickly than using Get and Put. You cannot, however, modify the image that you save with Push; nor can you Push the image from one Picture object and Pop it to another.

The system maintains only a single saved image (for each Picture control), not a stack. If you use Push a second time, the system replaces the saved image with the portion of the current value of the image property you specify.

The argument to Push is a four-element vector. The first two elements define the row and column (in current scaling units) of the upper left corner of the portion of the image that you want to specify. The third and fourth elements specify the height and width of the subarray.

Example:

Stores a 5-by-10 array of the upper-left corner of a Picture:

```
'fmMain.picArt' Øwi 'Push' 0 0 5 10
```

Put

Description:

Replaces a subarray of the image property of a Picture control with new values. You can use this method to change all or part of the graphical image for a Picture control without reassigning the image property.

Class: Picture.

Remarks:

The argument to Put is a three-element vector. The first element is the replacement subarray, which must be a matrix of color values with one element per pixel. The last two elements specify the row and column in the image property of the upper left corner of the subarray you want to replace, measured in current scaling units for the control. The height and width of the area replaced are determined by the size of the named subarray. If you specify only the replacement array, the system starts replacing at the upper left corner of the image property. Using the Get and Put methods avoids the potential problem that exists when using Copy and Paste of another application overwriting the clipboard.

Example:

The following command replaces the very center quarter of a 32-pixel bitmap with solid red.

```
'fmForm.picA' Øwi 'Put' (16 16 ρ255) 8 8
```

Redo

Description:

Redoes the last action that was undone in a RichEdit control, if Windows is capable of recognizing and remembering that action.

Class: RichEdit.

Remarks:

The canredo property is a Boolean flag that tells whether Windows can take this action. You can use this flag, for example, to activate or grey out a menu item that would trigger this method.

Ref

Description:

References multiple properties for an object.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

You normally reference a single property by naming the property as the right argument to Øwi with no other values. For example,

```
'fmMain'owi'opened'
1
```

The Ref method allows you to reference multiple properties by providing a nested vector of character vectors containing the property names as the additional arguments toowi. The result is a nested vector of values in the same order that you listed the properties; for example:

```
'fmMain'owi'Ref' 'opened' 'where' 'size'
1 1 1 15 40 13.25 39.2
```

You can also prefix property names with relative object names. This allows you to reference the properties of several different controls in one statement; for example:

```
'fmMain'owi'Ref' 'ed1.text' 'ed2.text'
Jane Smith
```

ReleaseObjects

Description:

Frees all validated pointers to ActiveX objects.

Class: # (the system object).

Remarks:

You can use this method to free all validated pointers you have acquired, from any call that returns an ActiveX pointer to APL, during a long-running callback when your program no longer needs the underlying resources. Note that this method does not take an argument, and you cannot apply it selectively.

ReqResize

Description:

Triggers the onReqResize event handler for a RichEdit control.

Class: RichEdit.

Remarks:

The onReqResize event handler calculates the height needed to display all the text in a RichEdit control. It returns a four-element vector showing the location, calculated height and current width. You can use this as an argument to the where property to resize the control.

Reset

Description:

Deletes all the objects from the APL+Win WI system.

Class: # (the system object).

Remarks:

You invoke this method only by usingowi. This method does not occur as a system- or user-generated event. When you use Reset on the system object, the system deletes all objects currently defined in APL+Win WI (except for the system object itself). It executes the onHide and onDestroy event handlers for any form that is open and the onDestroy event handler for any control objects on that form that are open. The system also resets theowself,owevent,owarg, andowres system variables to their default values.

Resize

Description:

Executes the onResize event handler for an object.

Classes:

Button, Check, Combo, CommandBar, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Page, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

The system executes the onResize event handler for an object when it opens the object or when its size changes. If you define an onResize event handler, you usually include logic to rearrange the relative placement of controls on a form or within a frame when the form changes size.

There may also be times when you want to force the system to perform rearrangement calculations. For example, if you have an onResize event handler for a form that calculates new sizes for its children, when the user drags an edge of the form, the system triggers the event handler. However, if you want the logic of the size calculation to be part of each child object, you could have the onResize event handler for the form invoke the **Resize** method on each affected child. Each child would be an independent self-contained entity, reacting to its own rules and standards.

ScrollCaret

Description:

Ensures that the text where the selection cursor is positioned in a RichEdit control is visible on the screen.

Class: RichEdit.

Remarks:

If the user has scrolled the text in a RichEdit control so the location of the selection cursor has moved off the screen, or if the selection has been set under program control so it is off the screen, the system moves the text so the selection cursor is in the visible portion of the control.

Seek

Description:

Moves to a specific location in a file.

Class: Media.

Remarks:

You can use this method to move to a specific position in the file and display that frame without continuing to play the file. The argument is a scalar integer that is the file position. You can reference the **range** property to find the valid values.

Send

Description:

Executes the onSend event handler for an object to enable communication between objects.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, and Tree.

Remarks:

The only immediate action of this method is to execute the `onSend` event handler for the named object. The value of `□warg` is whatever is to the right of 'Send' in the `□wi` statement. If the `onSend` event handler sets the value of `□wres`, the system returns this value as the result of `□wi`. The default value for `□wi` for the `Send` method is (0 0 ρ 0).

The `Send` method allows your GUI objects to assume a more formally object-oriented style of behavior. In effect, `Send` allows you to define an unlimited set of additional methods that an object can perform.

For example, suppose a form contains a frame that contains fields extracted from a database. Further assume that another control indicates when a new record should be retrieved and displayed in a field. The frame could respond to the request to get a new record by running the following `onSend` event handler.

```
'fmMain.fDataRecord' □wi 'onSend' 'fDataRecordSend'
```

```

    ∇fDataRecordSend
[1]    :select 1>□warg
[2]    :case 'GetRecord'
[3]    recno←2>□warg

    . . .      code to access the database and display the record
[15]   :case 'PutRecord'
    . . .
    ∇

```

Another part of the application could then use the following statement:

```
'fmMain.fDataRecord' □wi 'Send' 'GetRecord' 25
```

The advantage of using this approach is that the knowledge about the controls on the frame and the means used to read the data are centralized inside the frame's `onSend` handler rather than being dispersed through other parts of the application code. You could replace the frame so it had a different number of fields or accessed a different database manager with no change to the rest of the application.

Set

Description:

Sets multiple properties for an object.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

You can set a single property by naming the property and the value you want to assign as the right argument to `□wi`. For example, to maximize a form, you can specify:

```
'fmMain' □wi 'visible' 3
```

The Set method allows you to set multiple properties by providing a nested vector of nested vectors containing the property names and their values as the additional arguments to `□wi`; for example:
`'fmMain' □wi 'Set' ('size' 15 40) ('caption' 'Sample Form')`

You can also prefix property names with relative object names. This allows you to set the properties of several different controls in one statement; for example:

```
'fmMain' □wi 'Set' ('ed1.text' 'John') ('ed2.text' 'Doe')
```

SetCells

Description:

Changes information for items of a Listview object by changing the cells columns of its list property.

Class: Listview.

Remarks:

This method has three arguments. The first argument is an integer vector of the RowIDs of the rows you want to modify.

The second argument is an integer vector of the indices of the columns you want to modify, where column index is defined as it is in the `columndisplay` property: the origin one index to the columns following the RowID and images columns. You can add more cells to a row than there are columns in the existing list property. Specify the column indices with the next higher integer values. You cannot add rows with this method. The RowIDs must already exist.

The third argument is a nested matrix of character vectors with the new cell values. You must have one row for each RowID you specify and one column for each column index you specify.

SetImages

Description:

Changes images for items of a Listview object by changing the image indices column of its list property.

Class: Listview.

Remarks:

The first argument is an integer vector of the rows you want to modify. The second argument is an integer matrix with the new image, overlayimage, and userstateimage indices respectively.

SetInfo

Description:

Resets the specification for one or more nodes of a Tree object.

Class: Tree.

Remarks:

The first argument is one of the following operations: `'list'`, `'label'`, `'images'`, `'checkstate'`,

'highlightbold', 'highlightcut', 'highlightdrop', 'highlightselect', 'expanded', 'expandedonce', or 'forceparent'.

The remaining arguments identify the nodes to modify and the new attributes for that node. You may specify them as a vector of node identifiers followed by a vector or matrix containing the new values or as a matrix whose first column is the node identifiers and whose subsequent columns are the new values. The result is a Boolean vector showing 1 if successful, 0 otherwise for each node you specified.

Examples:

The following statements are equivalent:

```
'fmA.trB' □wi 'SetInfo' 'highlightbold' (1 5 10) (1 0 1)
'fmA.trB' □wi 'SetInfo' 'highlightbold' (3 2 ρ 1 1 5 0 10 1)
```

SetLinks

Description:

Sets or deletes a link name on an object by adding or removing a row from its links property.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Imagelist, Label, List, Listview, MDIForm, Media, Menu, Option, Page, Picture, Printer, Progress, RichEdit, Scroll, Selector, Spinner, Status, Timer, Toolbox, Trackbar, Tree, and # (the system object).

Remarks:

The argument to this method is a two-column matrix, or a two-element vector for a single link, where the first column contains the link name, a single name that is treated as a child of the object, and the second column contains the name of the target object to which the link applies. After setting the links property, your program can refer to the target object as though it is a direct child of the object that contains the link. If you use a relative name for the target object, the base name is the name of the object that defines the link. To delete a link, specify the second column as an empty character vector.

See the Programming the Windows Interface: Intermediate chapter for more information on naming with links.

SetPos

Description:

Sets the exact positions of items in largeimage or smallimage view in a Listview object.

Class: Listview.

Remarks:

In largeimage view, this method applies to the poslargeimage property, and in smallimage view, it applies to the possmallimage property. The argument to SetPos is a three-column matrix of RowIDs and row and column coordinates. The result is the arrangement of the Listview items.

You can use this methods only when the Listview is in one of the appropriate viewmodes. You can set the poslargeimage and possmallimage properties directly when the control is in other modes; they become effective when you switch to the appropriate viewmode. You can reorder the items of a Listview in report or

list view by using the sortorder property.

SetRows

Description:

Changes the items of a Listview object by changing the values of the cells and image indices of its list property.

Class: Listview.

Remarks:

The argument is a matrix with the same form as the list property. All the RowIDs you specify must already exist; the system updates the image and cell information with the values you specify. You can use -2 for the images entry to leave the indices unchanged. If you specify more columns than exist, they are added to the list property.

SetStatus

Description:

Changes one row of the status property matrix of a Status object.

Class: Status.

Remarks:

The full argument is a nested vector containing the pane number, text, width, border values, image indices, and pane-specific tooltip text. You can omit trailing elements of the argument. You can also provide a simple character vector; the system assumes this to be the (column 2) text for the main pane.

Setup

Description:

Displays a common dialog box for configuring a printer.

Class: Printer.

Remarks:

When you invoke Setup, the system displays a standard dialog box where you can select a printer, orientation (portrait or landscape), and the paper size and source. You can also set some graphics options. This method returns 1 if the user presses OK on the dialog box, zero if Cancel. In earlier versions of APL+Win, this method returned an empty matrix if the user pressed the OK button in the dialog box.

See also the descriptions of the PageSetup and PrintDlg methods for similar configuration methods.

Show

Description:

Displays a form or control object.

Classes:

Button, Check, Combo, CommandBar, CommandButton, DateTime, Edit, Form, Frame, Label, List, Listview, MDIForm, Media, Option, Picture, Progress, RichEdit, Scroll, Selector, Spinner, Status, Toolbox, Trackbar, and Tree.

Remarks:

The Show method is the opposite of Hide. If the object is open and visible on the screen, the only effect is to trigger the onShow event handler. If the object is not open, the Show method first invokes the Open method, which triggers both the onOpen and onResize event handlers for the object. It also triggers the onOpen event handlers for any objects that this object contains. The system then executes the onShow event handlers for the object. If you specify values to the right of 'Show' in the `□wi` statement, these values are passed to the onShow event handler in `□warg`.

If the object is open but hidden, this method changes the visible property value by making a negative value positive. The system displays the object, or its icon if it is minimized.

When applied to a Form object, the Show method is similar to the Wait method, except that the Show method allows the application to continue without waiting for input.

ShowNode

Description:

Makes a node of a Tree object the currently selected node or the current drag-and-drop target, or positions a node at the top of the visible portion of the tree.

Class: Tree.

Remarks:

The first argument is one of the character vectors: 'select', 'drop', or 'firstvisible'. The second argument is the node identifier if you want to specify a node; you can also specify zero or the character vector 'none' to remove the attribute from the currently selected node or the current drag and drop target.

This method returns 1 if successful; zero otherwise. This method can be unsuccessful because the onValidate event handler set `□wres` to zero. You cannot use this method if the tree is closed.

SortChildren

Description:

Sorts the children of a specified node in a Tree object.

Class: Tree.

Remarks:

The argument is the identifier of the node whose children you want to sort (or 'root' if you want to sort

the top level nodes), optionally followed by one or both of 'recurse' or 'onSort'. If you specify only the node identifier, the system uses the labels and sorts the immediate children nodes alphabetically. If you specify 'recurse' the system recursively sorts all the children's children. Specifying 'onSort' invokes the onSort handler to determine the relative order of the children. `❏wi` returns 1 if successful, 0 otherwise.

Note: You should be cautious in using this method when the tree is closed. The system cannot query a closed tree for labels if you have any marked as virtual. It uses the last value from the `GetVirtualData` event, if any; otherwise it uses an empty vector. Using the 'onSort' argument also is ineffective when the tree is closed.

Stepit

Description:

Advances the current position of a Progress bar by its step increment and redraws it.

Class: Progress.

Remarks:

You cause this behavior only by using this method. There is not a comparable system- or user-generated event. If stepping causes the current position to go out of range (above the maximum position or below the minimum position), the current position wraps around.

If you provide no argument, the current position (first element of the `value` property) is advanced by the step increment (the third element of `value`). You can provide an integer argument to this method that represents the amount by which you want the current position to change.

Stop

Description:

Stops the current file in a Media control from playing.

Class: Media.

Remarks:

For .ANI and .AVI files, and some other files, an image remains on the screen.

Undo

Description:

Undoes the last action in a RichEdit control, if Windows is capable of recognizing and remembering that action.

Class: RichEdit.

Remarks:

The `canundo` property is a Boolean flag that tells whether Windows can take this action. You can use this flag, for example, to activate or grey out a menu item that would trigger this method.

UseObject

Description:

Establishes and retains the validity of a pointer to an ActiveX object for the duration of a callback.

Class: # (the system object).

Remarks:

If you acquire a pointer to an ActiveX object, for example from a low-level Windows call, you can use this method to ensure the pointer's validity for the duration of a callback.

Validate

Description:

Triggers the onValidate event handler.

Classes: Page, Selector, and Tree.

Remarks:

You can use this method, for example, to perform the validation that would be done if the user tried to switch Pages by clicking on a tab. This might be used in the onClick handler for an OK or Apply button. When applied to a Selector, the onValidate event handler of the current Page is invoked first. If it returns a nonzero value in `OnVres` then the onValidate event handler of the Selector is called. The method returns 1 if both onValidate event handlers return a nonzero value, otherwise it returns 0. When applied to a Page, only the onValidate event handler for the Page is invoked. The result is 1 if the event handler returns a nonzero value, otherwise it is 0.

VT

Description:

Returns the tagged datatype of an ActiveX object or creates a tagged datatype value in APL that corresponds to the datatype system used by ActiveX objects (and Visual Basic).

Class: # (the system object).

Remarks:

VT stands for Variant Type. You can use this method to query the datatype of an ActiveX object, for example, a cell or range of cells in a spreadsheet. You can also use it to create a tagged datatype value in APL, which you can then put into an object such as a cell of a spreadsheet. This method is analogous to the system function `Qdr` in APL+Win, which returns the datatype of its argument or converts an argument to a specified datatype.

APL+Win generally uses four datatypes: 64-bit floating point, 32-bit integer, 8-bit character, and 1-bit Boolean. The first three datatypes have corresponding datatypes in ActiveX; generally if you are using data in any of these representations, the conversions occur correctly and automatically. (Some ActiveX applications also have a Boolean datatype, but that datatype generally is a 1-byte per value representation that does not correspond to the APL+Win Boolean, which actually is a 1-bit per value representation for Boolean arrays.)

If you want your application to read or set any of the other datatypes used in ActiveX, you must determine the type or tag such data so both APL+Win and the ActiveX object treat them the same. When you read an ActiveX datatype, you can get one of two values for most datatypes; one code for a single datum, and a different code for an array of such data. When you are tagging data for use by an ActiveX object, you generally must tag each individual value, even in a homogeneous array.

The datatypes used in ActiveX and their corresponding codes are shown below:

VB Type Name	Scalar Type	Array Type
Empty	0	
Null	1	
Integer	2	8194
Long	3	8195
Single	4	8196
Double	5	8197
Currency	6	8198
Date	7	8199
String	8	8200
Object	9	8201
Error	10	8202
Boolean	11	8203
Variant	12	8204
DataObject	13	8205
Byte	17	8209

Note 1: Unlike APL, you cannot have arrays of empty or null values.

Note 2: Variant type codes 12 and 8204 are not types of actual values. An array of variant type is similar to the APL heterogeneous array. It can contain values of other types mixed together. The scalar variant type is a place holder for a value that may take any one of several types. You cannot have an actual scalar value of variant type 12, but you can have a variant array type 8204.

Note 3: When the type code is one of the object type codes scalar values 9 or 13, the corresponding value is a pointer to the object.

You can provide a single argument to this method that is a variable containing a value or values (presumably tagged, that you might have acquired, for example, by using the at sign (@) as a suffix following a method or property name in Excel). If the value is tagged as an array, VT returns a code for the array; if the value is an individual element or if values in an array are tagged individually, VT returns one or more of the scalar datatype codes. If you simply display a tagged array in APL, you see "VT" (an indication that it is tagged), the array datatype code and the array. If you display individually tagged elements, you see for each element: VT, the scalar datatype code and the element value.

If you want to create tagged values, you must supply two arguments; the first argument is the raw value that you want tagged as a specified datatype, and the second argument is one of the datatype codes. Examples of using the VT method are shown in the Using ActiveX in APL+Win chapter.

As an alternative to using this method (and VV below), there are cover functions, named VT and VV, that you can use in lieu of making an explicit `ⓘwi` statement. You can find these functions in the VT_VV workspace in the `Tools` subdirectory where you installed APL. In particular, it may be easier to use the cover function with the Each operator to create tagged values than it is to use the method, which would require a statement such as: `'#' ⓘwi'' (⊂⊂'VT'),'array','type`

VV

Description:

Returns the raw value of an object that contains tagged ActiveX (or Visual Basic) datatype values; the object can be an ActiveX object that contains tagged datatype values or an APL variable that has been assigned tagged datatype values.

Class: # (the system object).

Remarks:

You use this method as a companion to the VT method, to extract the raw value from an object that contains a tagged datatype value. You supply a (presumably tagged) value as the argument, and the method returns the simple APL values. See the Using ActiveX in APL+Win chapter in this manual for more information.

Wait

Description:

Displays a form and initiates a modal wait for input. You must display the main form of an application (the first form that allows user actions) using the `Wait` method.

Class: Form and MDIForm.

Remarks:

This method triggers the `onWait` event handler for a form, if one exists. If the form is not open, `Wait` first invokes the `Open` method, which triggers both the `onOpen` and `onResize` event handlers for the form. It also triggers the `onOpen` event handlers for any objects that this form contains. The system then executes the `onWait` event handler for the form. If you specify values to the right of `'Wait '` in the `□wi` statement, these values are passed to the `onWait` event handler in `□warg`.

When you use the `Wait` method, the system starts a modal wait loop, which continues until the form is closed or hidden. A modal wait disables all forms that were previously opened in this application. The user cannot switch to any form this application previously opened; you must close or hide a modal form before the user can access a form that was previously open.

When you close a modal form, the system also closes all forms that were opened after it. It triggers the `onDestroy` event handlers for each open form, but it triggers the `onClose` event handler only for the modal form.

If you use the `Wait` method on a form other than the main form of your application, you essentially start a subapplication within your main application, which operates until the modal form is closed or hidden. At this time, the form that was in a modal wait loop when you opened the last modal form resumes control. This method allows you to create layers in your application, where a user must complete all actions on one layer before returning to the previous layer.

You can also use modal forms when you must have a response before the application can continue. However, you should avoid using modal forms unless necessary. The `style` property for a form allows different ways for the form to interact with the `Wait` method.

An event handler must eventually close or hide the modal form. The explicit result of `□wi` is the form's `value` property. You can use this to pass a nested array that contains the state of the objects on the form when you close it. When you close the main form of your application, the system closes all the other forms that are still open and returns from `Wait`.

Usually, the main form is the first form you open. You can display a form using the `Show` method before you invoke any `Wait` method, for example a display-only form at the beginning of the application while the system performs its initialization steps.

Note: If you want to experiment with the `Wait` method in immediate execution mode, be sure your form has a system menu or a Cancel button (style 2) before you invoke `Wait`.

XInfo

Description:

Lists ActiveX controls and objects that are registered and available on the system.

Class: # (the system object).

Remarks:

The result of XInfo is a three-column matrix that gives information about ActiveX controls and objects that are registered on your system. The columns are the programmatic identifier of the object, whether it is an ActiveControl or an ActiveObject, and its descriptive name. One entry in the matrix may refer to another registry entry; usually the referenced entry will have a similar descriptive name with a different suffix. This technique of registering ActiveX objects allows manufacturers to update their products and have you call the most recent version without requiring you to change your code.

You can simply invoke this method to get the entire list, but it may be very large. You may want to query the shape of the result before you display the list. You can use a character vector argument with the XInfo method to limit the list. Using the following statement returns all the registered ActiveX objects starting with the letters MS.

```
'#' □wi 'XInfo' 'MS'
```

You can also use the asterisk (*) as a wild card symbol in the argument to XInfo. If you made the argument to XInfo in the above statement '★MS', the statement would return all the registered ActiveX objects that contain MS anywhere in their name.

Many objects will exist in families, where an application calls objects for its constituent parts. You can see such relationships if you have, for example, Excel, installed on your system. The Excel application actually creates ActiveX objects for its charts and worksheets.

Chapter 5: Programming the Windows Interface: Introduction

APL+Win provides the means for the APL programmer to exploit the power of the Microsoft Windows environment while enjoying the rapid software development and high-performance array computing of a modern 32-bit APL system. This chapter is an introduction to Windows programming in APL+Win.

Overview

APL+Win implements its Windows interface using the system function `⌈wi` (Windows Interface) and a group of related system variables. Using `⌈wi`, you can:

- Create instances of forms, controls, and invisible objects that are built-in to APL+Win to construct a graphical user interface for your application.
- Reference and set properties for all of these objects.
- Invoke methods that send a message or cause an effect in the application.
- Respond to events that occur in the application by executing an APL statement or function.
- Create and use any of a vast array of third-party controls that adhere to the ActiveX (Component Object Model) standard. These have also been known as OCXs in the past.
- Activate and communicate with ActiveX objects that function as servers; see the Using ActiveX in APL+Win chapter.
- Create an instance of APL to act as an ActiveX server; see the APL+Win as an ActiveX Server chapter.
- Use interface techniques common to Windows applications, such as drag-and-drop.
- Communicate with other applications through standard Windows methods, such as Dynamic Data Exchange (DDE).
- Design your own classes of objects or modify the behavior of built-in APL classes; see the Creating User Defined Classes chapter.

In addition to all the functionality available through the standard Windows Interface system function, you can:

- Access the Windows Application Programming Interface at a low-level by using the system function `⌈wcall`.
- Access facilities in a Dynamic Link Library using either `⌈wcall` or the system function `⌈na`.
- Employ the add-on product APL Link or its associated workspace C30 to query and transfer data to and from database applications using the Open Database Connectivity (ODBC) standard.

Understanding the Fundamental Elements

The Windows interface facility in APL+Win is based on an object-oriented approach to software design. An object is a software structure that can be thought of as a self-contained entity with a simple and well-defined set of behaviors. All the details of how the object works are inherent in the definition of the object. The only information necessary to use the object is a simple description of its characteristics and capabilities. You can describe an object by the class to which it belongs, and its properties, methods, events, and event handlers.

- A class is a category of objects, all of which have certain common characteristics. If you have a pair of radio buttons on a form that allow the user to choose either high resolution or low resolution, each of those buttons (with its associated caption that describes the choice) is an instance of the Option class.
- Properties are attributes or characteristics of an object that describe the object's state; they affect the object's appearance and behavior. Most of an object's properties are read/write, which means that they can be changed at any time during the life of an object. A few properties are read-only. Some read-only properties are a function of the object's class, for example, events, methods, and properties, which are unchanging. Other read-only properties change as a result of system actions.

You set or reference properties under program control using the system function `⌈wi`. Setting a property can be more than simply assigning a variable, however, since the system often takes action when a property is set. Setting the `range` property for a spinner provides no visual indication of a change; it simply sets lower and upper limits for the allowed value. In fact, a user would have no indication of when the limit is reached except for the display you create or the action you associate with the spinner's value. Setting the `caption` property for a button, however, not only stores the caption inside the data structure that represents the button; the system also displays the new caption on the button immediately. Even more dramatically, setting the `border` property for a form can, in addition to changing the appearance of the border itself, cause the default buttons on the title bar to disappear and prevent the user from changing the size of the form by dragging an edge or corner.

- Methods are actions that you can invoke on an object. For example, you can use the `Show` method to make a form visible on the screen and the `Hide` method to make it invisible. Methods are usually limited in scope, but your program controls when these actions take place. Invoking a method is similar to calling a function: you provide the name of the method to apply to the object, and sometimes arguments to further specify the action. But, conceptually, it is more like a request for the object to do something. The object has the right to refuse, or do something different, to keep itself in a correct and self-consistent state. For example, you can ask a form to close itself by invoking its `Close` method. The form is responsible for determining whether it can be closed safely and for taking any actions that need to be handled first.
- Events are notifications that something has happened to which an object may want to respond; for example, when a user clicks the left mouse button with the mouse cursor above a list on your form, Windows tells APL that this action has taken place. APL then signals a `Click` event to the List control. The system can handle this event by running the list's `onClick` event-handler property.
- Event handlers are actions that the system takes when a specific event occurs in the application. You define event handlers by specifying APL statements that you want the system to execute when the object recognizes an event. For example, you set a list's `onClick` property to a character vector containing an APL statement you want the system to execute in response to the `Click` event. The statement can, of course, have different effects depending on which item in the list was clicked.

Event handlers can be very broad in scope; you can use the full power of the APL language to perform whatever actions you deem valuable. However, your program does not control directly when these actions take place. An external impetus, the action that causes the event to which your handler function responds, triggers the handler.

Events usually come from the underlying operating system, but they also occur as a side effect of actions taken in your own APL code. Invoking a method frequently triggers an event on the object on which you invoke the method or possibly on another object; for example, using the `Focus` method to place the

focus on a control triggers both an `Unfocus` event on the object that had the focus and a `Focus` event on the target.

When the user tabs out of an edit box or otherwise moves within the application, the system triggers an `Exit` event. If you reset the system variable `□wres` in your `onExit` event handler, that generates an `ExitError` event. You can use this pair of event handlers to validate user input.

When the user types or deletes a character in an edit control, the system triggers a `Change` event. If your program places a default value in the edit box, it also triggers the `Change` event by default. You can choose to suppress the trigger when caused by a program action, while retaining the event if caused by user action or a method, by resetting the `suppress` property for the edit box.

If you do not define a handler for an event, your application code ignores the event. However, many events produce some sort of default action whether or not you define a handler. For example, when the user clicks a button, the image of the button on the screen changes to give the illusion of a button being depressed and released. If you have not defined a handler, the user can push the button and see the visual effect, but nothing else happens.

An interface program, even a simple one with only a few visual components drawn on the screen can be a very complex software structure. Because there are few constraints on the order in which the end user can take actions, the number of potential actions and interactions between pieces of an interface are enormous. It is the responsibility of your application to guide the user to the desired goal and to work correctly no matter where the user clicks the mouse or which keys the user presses.

The single most important concept to grasp is that Windows uses events to tell you what the user has done to your application's user interface. Every aspect of the user interface code centers around this idea. The need to be able to respond to events determines the topmost level of your application's structure.

Naming Objects

In order to create an object, you must name it and identify its class. Objects exist in a hierarchy. A form can be a top level object; the controls on that form are its children. Object names show this hierarchy. For example, a button named `bnCancel` on a form named `fmMain`, has the fully-qualified name `fmMain.bnCancel`. The hierarchy can be arbitrarily deep. You use the name in the left argument to `□wi` to identify the object that is being manipulated.

You can always identify an object by its fully qualified name, which describes its place in the entire hierarchy. But you can also refer to an object by its individual name in the context of the current object. The system variable `□wself` represents the current object, and its effect is similar to the idea of the current file directory in DOS. For example, when `□wself←'fmMain'`, you can use the name `'bnCancel'` to refer to the button whose fully qualified name is `'fmMain.bnCancel'`. This is more than convenient shorthand; it is a valuable programming technique that lets you write code that applies to more than one instance of an object.

The examples in the APL+Win manuals use a naming convention for objects with a prefix such as `fm` or `bn` to indicate what kind of object they are (form, button, etc.). APL+Win does not require this naming convention; you can use any name you like and the system works the same way. However, using this convention is very valuable in helping you read and understand code dealing with the interface. If the object's name indicates what kind of object it is, it immediately conveys information about the object. APL2000 recommends that you use this convention while you are learning to use APL+Win. After you are experienced with the system, you may find that you prefer to drop the convention or devise your own. If you intend to write code that other programmers will use or maintain, using the standard name prefixes is especially recommended.

Windows Interface and System Variables Summary

Windows Interface (`⌵wi`)

`⌵wi` is the general-purpose function for creating, controlling, and running user interfaces. You use it to create forms and controls, to set their properties, to reference their properties, to invoke methods, to associate handlers with events, and to trigger callbacks into those handlers when an event occurs. The following examples show some of its uses.

To create a new object and initialize its properties:

```
result ← 'object' ⌵wi 'New' 'class' ( 'property' value ) ( 'property' value )...
```

To set a property for a control:

```
'object' ⌵wi 'property' value
```

To reference a property:

```
value ← 'object' ⌵wi 'property'
```

To invoke a method and supply its arguments:

```
'object' ⌵wi 'method' args
```

To draw graphics on an object (using the Draw method to invoke a series of graphical operations and supply their arguments):

```
'object' ⌵wi 'Draw' ( 'operation' args ) ( 'operation' args ) ...
```

To define an APL expression as a handler for a specific event:

```
'object' ⌵wi 'onEvent' 'expression'
```

To give the input focus to a form and wait for user input, with handlers called when events occur:

```
'form' ⌵wi 'Wait'
```

Note: The word that you place immediately to the right of `⌵wi`, whether it is the name of a method, an event handler, or a property, is known as an *action*. Error messages, for example, refer to the element in this position as an action. This term is sometimes used in this chapter, and throughout the manual, in its specialized sense, but more often in the generalized sense of the word, as a system action or user action.

Current Windows Object (`⌵wself`)

The system variable `⌵wself` is a character vector that contains the fully qualified name of the current object. If you assign the name of an object to `⌵wself`, you can refer to the object implicitly by using `⌵wi` without a left argument. More importantly, during an event-handler callback, APL automatically sets `⌵wself` to the name of the object for which the event occurred. This allows you to write a handler that works with different objects, since any actions that you take with monadic `⌵wi` automatically apply to the current object. It also allows your code to identify the object to which an event applies.

Windows Event (`⌵wevent`)

The system variable `⌵wevent` contains the name of the event that is being handled during a callback. It allows you to write a single event handler function that applies to more than one type of event.

Windows Argument (`⌵warg`)

The system variable `⌵warg` contains the arguments, if any, associated with an event or method. Most user-triggered events do not have arguments, but a few do. For example, during the handling of a `Resize` event, `⌵warg` contains information about the window's new size. When a method you invoke under program control generates an event, the handler can use `⌵warg` to obtain the arguments you provided to the method.

Windows Result (`⌵wres`)

The system variable `⌵wres` specifies the explicit result, if any, that a method or event handler returns. The windows system variables are implicitly localized during a handler or method call, so your code can set `⌵wres` and `⌵warg` freely, as with `⌵wself`. As soon as the handler or method returns, the system restores them to their former values.

Building a Graphical User Interface in APL

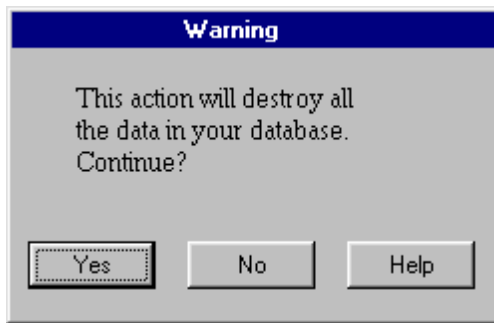
The visible portion of an APL user interface consists of forms and controls. A form is an individual window or dialog box that you display on the user's screen. A typical application might have one main form, which is visible at all times, and a group of dialog boxes, which are smaller forms that appear when they are needed for a special purpose and then are hidden again. By convention, the main form displays a title that identifies the application. Smaller dialog boxes usually have titles that identify their specific purpose.

Controls are graphical elements, such as a button, list box, or scroll bar, that appear on a form. Each control represents a single part of the user interface, such as a line of text, a graphic image, or a visual element that provides information. The user manipulates controls to provide input to the application.

The Windows user interface puts the user in control. Multiple applications may be running at once, and each application can have several forms open at once. The user generally can move or minimize forms at will and select one to work with next. Only one window can be active at a time, which means that the user's keystrokes and mouse actions apply to this specific window. This window is said to have the input focus; the Windows operating system denotes the window with the input focus by setting the color of its border and title bar. These color settings are system wide and are controlled by the Color settings the user selects with the Windows Control Panel utility.

Most windows in most applications are modeless, which means that the user can click on a different window and shift the input focus at any time. In special circumstances, however, you may want to use a modal form, which prevents the user from shifting the focus to another form. It is considered bad style to use modal forms except when the user needs to be told about something that requires immediate attention or where it is dangerous to let the user perform another action. The figure below shows a message box, one type of form that is often modal.

A Modal Message box



Using Controls to Get User Input

When a window is active, not only does it have the focus, one specific control has the input focus. The user can move the input focus among the controls inside a form using the Tab and Shift+Tab keys, or by clicking on a different control. Any control that can have the input focus, receives it when the user presses the left mouse button over that control. In a typical form, the first edit field a user normally types might have the focus initially.

Note that most, but not all user actions apply to the control that has the focus. In an edit box, for example, pressing any alphabetic or numeric key results in a new character in the box. Clicking the mouse button, if the mouse cursor is over the edit box, positions the input cursor in the field. Double clicking selects a word and highlights it. Arrow keys move the input cursor along the text that is displayed. The tab key generally causes the focus to move to another control. Pressing the Enter key may or may not have an effect. If the edit control is defined to be a multiline control, pressing Enter puts the input cursor on a new line, by virtue of entering a nonvisible newline character. If the edit box is a single-line style (the default), pressing the Enter key may have no effect, or it may be read by another control, typically a button that closes the form (with a caption like OK). Similarly, the Escape key may have the same effect as clicking a Cancel button.

Your program can use any control that can receive the focus to interact with the user. Various types of edit boxes allow free-form typing. A button allows the user to initiate an action. Clicking or otherwise selecting a menu item may initiate an action or expose a further cascade of choices. Selecting one radio button may be a passive choice that affects later actions, or it may also trigger an action. Radio buttons are often in sets of three or more mutually exclusive choices. A check box allows a single entry to distinguish between two alternatives; the nature of the choice and the caption should make it clear what both the checked and unchecked states imply. List controls allow an ordered set of choices, where you can decide if the user is limited to a single choice or may choose multiples. Other controls, such as listviews and trees, allow very extensive and complex displays of data and choices.

You use events and properties of a control to read the user input. A control generally recognizes an event each time the user selects it, activates it, or changes its essence in some way. In addition, the control has a property that stores the condition corresponding to its being selected. An edit control, for example, has a text property that holds a character vector that is the contents of the edit field. It also has a Change event that fires each time the user changes the text.

A check box and each radio button in a group has a value property that is a scalar one when it is checked or selected and zero if not. There is also a Click event that fires each time the user selects the control; in the case of the check box, the Click event also fires when the user deselects it. You can use the event handler

properties to take action at the time of a change, or you can reference the property at a time of your choosing, determine its value, and take an action based on the result.

Understanding the Basics

The sections below describe the fundamental mechanics of creating objects and using names to refer to objects for purposes of creating other objects, setting and referencing properties of objects, manipulating objects, and responding to events. These mechanics are necessary to programming the APL+Win interface.

Creating Objects

You create a new object by using either the `New` or the `Create` method with `□wi`. The left argument of `□wi` specifies the name of the object you want to create. The right argument contains the action `New`, followed by the class of the new object. For example, the following statement creates a new form named `fmMain`:

```
name ← 'fmMain' □wi 'New' 'Form'
```

The explicit result of `□wi` is the name of the new object:

```
name  
fmMain
```

You do not need to assign the result to a variable. To create a new button on the form, you can use:

```
'fmMain.bnClose' □wi 'Create' 'Button'  
fmMain.bnClose
```

The difference between the `New` and `Create` methods is apparent only when you already have an object with the name you are using to create a new object. The `New` method returns an error; the `Create` method deletes the existing object and creates your new object without returning an error.

Note: The examples in the next two sections do not show any explicit results; the section titled "Using Relative Object Naming" sometimes includes the results to more clearly demonstrate the concepts.

You can also specify the initial value of any number of properties when you create an object. You specify each property as one item in the argument; you must enclose each item, which consists of the property name followed by its initial value. For example:

```
'fmSample' □wi 'New' 'Form' ('where' 1 1 15 40) ('border' 18)  
⌘ ('caption' 'Sample Form')
```

You can create an object relative to the current object (identified in `□wself`) by omitting the left argument to `□wi` and specifying the name of the new object in the right argument. Thus, the following statements create a form with a line of text and two buttons on it:

```
□wself←'fmAsk' □wi 'New' 'Form' ('where' 5 5 15 50) ('border' 3)  
□wi 'lMsg.New' 'Label' ('where' 2 4 1.5 40)  
⌘ ('caption' 'Continue installation?')  
□wi 'bn1.New' 'Button' ('caption' 'Continue') ('onClick' 'DoInstall')  
⌘ ('where' 6 20 1.5 10)  
□wi 'bn2.New' 'Button' ('caption' 'Cancel') ('onClick' 'DoCancel')  
⌘ ('where' 8 20 1.5 10)
```

A group of statements like the preceding can produce unwanted visible activity on the screen, since each control appears as the system creates it. A good technique is to invoke the `Close` method on the form when you create it. Then you can create the controls, initialize their properties invisibly, and display the form with the `Wait` or `Show` methods when it is finished. This technique causes the form to appear instantly on the screen instead of allowing the user to watch each step of its creation.

```
□wself←'fmAsk' □wi 'New' 'Form' 'Close' ('where' 5 5 15 50) ('border' 3)  
□wi 'lMsg.New' 'Label' ('where' 2 4 1.5 40)  
⌘ ('caption' 'Continue installation?')  
□wi 'bn1.New' 'Button' ('caption' 'Continue') ('onClick' 'DoInstall')  
⌘ ('where' 6 20 1.5 10)
```



```

□wi 'bn2.New' 'Button' ('caption' 'Cancel') ('onClick' 'DoCancel')
↳ ('where' 8 20 1.5 10)
'fmAsk' □wi 'Wait'

```

Using Precompiled Object Definitions

You can also create a copy of an object, including all its children, from a variable that contains an object definition. Such a variable holds a precompiled form or portion of a form that you can load into the APL+Win interface without creating each object separately. You obtain the definition for an object by referencing its `def` property. To generate a definition for the form `fmEdit`:

```
editTemplate←'fmEdit' □wi 'def'
```

You can then use that definition as a template to create several identical forms. The name of each form will be the name you specify when you create the form, even though the name in the definition is different. The names of the objects on the forms will have the top-level parent name (the portion to the left of the first period) that you specify for each form; the remainder of the fully-qualified names will be identical across all the forms.

```

'fmEditColors' □wi 'New' editTemplate
'fmEditTextures' □wi 'New' editTemplate
'fmEditPatterns' □wi 'New' editTemplate

```

During the time that a form is in active use, all the information about the form and its controls is stored in a section of Windows memory that is separate from the APL workspace. You can reference the `def` property of an open object at any time to obtain the definition of the object in its current state.

The form definition contains the complete specification for the form and all its children, including each of their properties. It is an ordinary nested vector, but its internal structure is meaningful only to `□wi`. It contains various data structures that are not meaningful to APL primitives. If you display the array, it looks like a condensed text version.

Note: If you modify it, it is permanently damaged and your template is unusable.

The `def` property enables you to redefine an existing form. This technique has many uses. For example, suppose you have a form with many edit fields in which the user types input. An easy way to re-initialize the form to a clear state is to set its `def` property using a definition of the form that you made when every field was empty:

```
'fmInput' □wi 'def' fmInputDef
```

Because a form definition is a regular nested array, you can easily store it as part of your application. You can save definitions as variables in the workspace or store them in a component file. If the form has a large number of controls, it is more convenient to re-create it from an existing form definition than it is to create the form and all its controls and define all their properties with individual `□wi` statements for each control.

The form definition includes the actual event-handler properties (`onClick`, `onClose`, etc) of itself and its children, but it does not include the APL functions that they call. If you want to store a form outside the workspace along with all the functions that it can call, you must also store the functions that the form calls. If you adopt the naming convention of using the form name as a prefix in the names of the functions it calls, you can easily locate all of the relevant functions.

You can also create definitions for parts of forms. For example, suppose you want to have two different groups of controls occupy the same part of the screen and dynamically switch between them while the form is running. If you group the two sets of controls in frames and generate a template for each frame, you can switch very quickly between the two sets of controls.

To accomplish this, you first create a frame on the form that contains the group of controls. Once the frame is complete, you can save its definition in a variable and delete the frame:

```
frame1A←'fmMain.frame1' □wi 'def'  
'fmMain.frame1' □wi 'Delete'
```

Next, create the variant form of the frame and save its definition also. You can also delete the second frame.

```
frame1B←'fmMain.frame1' □wi 'def'  
'fmMain.frame1' □wi 'Delete'
```

At this point, you can show the form and define the first variation of the frame (described by `frame1A`).

```
'fmMain' □wi 'Show'  
'fmMain.frame1' □wi 'New' frame1A
```

At a later time, you overlay the frame with the second variant just by resetting the `def` property for the frame.

```
'fmMain.frame1' □wi 'def' frame1B
```

Note that you can follow the template with an action list that sets properties and invokes methods on the object immediately after it has been created. This gives you the means to create and initialize a form from a template without the form being visible to the user. You can `Open` or `Show` the form at a later time.

```
'fmMain' □wi 'New' fmMainDef ('Close') ('where' 0 0)
```

Using Relative Object Naming

While the standard syntax for using `□wi` has the name of the object (explicit or implicit from `□wself`) in the left argument and the action in the right argument, under many circumstances you can use only a right argument. This technique is called relative naming. There are two important rules that apply to relative naming conventions:

- Names are relative to a top-level object (typically a form), either the current object or the top-level object that contains the current object. If `□wself` is `fmForm.fFrame.bnOK`, then the top-level object is `fmForm`. This section uses the terms top-level object and descendant to describe the objects in this hierarchy. A descendant is any object in the hierarchy below the top-level object.
- Relative naming takes precedence over a left argument.

If `□wself` is not defined, you specify the top-level object in the right argument, followed by a colon. If you want an action to apply to the top-level object, you follow the colon with a property or method. If you want an action to apply to a descendant, follow the colon with the rest of the fully qualified name of the descendant, followed by a period and the method or property. For example:

<code>□wself←''</code>	A <code>□wself</code> is undefined
<code>□wi 'fmForm:Open'</code>	A Invoke a method
<code>□wi 'fmForm:children'</code>	A Reference a property
<code>□wi 'fmForm:fFrame.caption' 'Frame'</code>	A Set a property
<code>□wi 'fmForm:fFrame.bnOK.Click'</code>	A Third-level child

If `□wself` is defined as the top-level object, and you want the action to apply to that object, relative naming is not necessary; you can simply specify the property or method you want in the right argument. If you want an action to apply to a descendant, specify the portion of the right argument above that follows the colon.

<code>□wself←'fmForm'</code>	A <code>□wself</code> is top-level object
<code>□wi 'limitwhere'</code>	A property of the form
<code>□wi 'fFrame.Paint'</code>	A method on child of the form
<code>□wi 'fFrame.bnOK.Focus'</code>	A method on grandchild of form

If `□wself` is defined as a descendant, and you want an action to apply to a top-level object, begin the right argument with a colon and follow it with the method or property. Note that in the third line, specifying the right argument without the colon would place the focus on the button; including the colon puts the focus on the form.

```

□wself←'fmForm.fFrame.bnOK'      A □wself is the button
□wi ':color' (255 255 255)      A property of the form
□wi ':Focus'                    A puts the focus on the form

```

When □wself is defined as a descendant and you want an action to apply to that specific object, relative naming is not necessary. You can simply specify the right argument and □wself is the default left argument.

```

□wi 'size'                      A references size of the button

```

You can still use relative naming on other objects that are descendants; specify the argument in the same manner as when □wself is the top-level object. For example,

```

□wi 'fFrame.pointer' 2          A property of child of top-level obj

```

When you are specifying descendants, you must specify all parts of the name that follow the top-level object. The following example would fail, because the system looks for the nonexistent object fmForm.bnOK.

```

□wself←'fmForm.fFrame'
□wi 'bnOK.enabled'

```

Using Multiple Periods in the Arguments

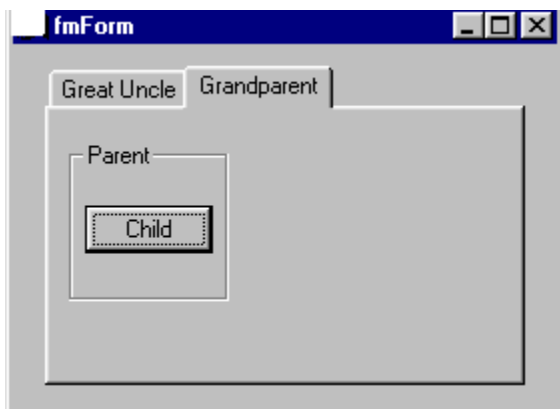
You can also specify multiple leading periods in the arguments to □wi to show levels of parentage. Starting the right argument with or without a period applies equally to the current object. Two periods in an argument imply the parent of the object identified by □wself, while three periods identifies the grandparent. The example below builds a form, shown in the figure, holding a selector that has two pages, the second of which contains a frame holding a button. The examples use the button as the current object.

```

'fmForm' □wi 'Create' 'Form' ('where' 1 60 13 35)
fmForm
'fmForm.selShow' □wi 'New' 'Selector'
fmForm.selShow
'fmForm.selShow.pgA' □wi 'New' 'Page' ('caption' 'Great Uncle')
fmForm.selShow.pgA
'fmForm.selShow.pgB' □wi 'New' 'Page' ('caption' 'Grandparent')
fmForm.selShow.pgB
'fmForm' □wi 'selShow.value' 2
'fmForm.selShow.pgB.fHold' □wi 'New' 'Frame' ('caption' 'Parent')
↳ ('where' 1 1 5 10)
fmForm.selShow.pgB.fHold
'fmForm.selShow.pgB.fHold.bnChild' □wi 'New' 'Button'
↳ ('where' 2 1 1.5 8) ('caption' 'Child')
fmForm.selShow.pgB.fHold.bnChild

```

Relative Naming of Objects



The following examples show the use of relative naming and the effect of periods in the right argument to `□wi`.

```
□wi.  
  □wself←'fmForm.selShow.pgB.fHold.bnChild'  
  □wi 'Ref' 'class' 'caption'  A Relative to the current object  
Button Child  
  □wi '.class'      A One dot is optional for an action  
Button  
  □wi '..children'   A Property of parent (frame)  
  fmForm.selShow.pgB.fHold.bnChild  
  □wi '...visible'   A Property of grandparent (page B)  
1  
  □wi '....self'    A Property of great grandparent (selector)  
fmForm.selShow  
  □wi '....pgA.visible' A Other child of great grandparent  
-1  
  □wi '.....border' A Property of top-level form  
114  
  □wi '.....methods' A Property of system object  
  Close Defer Event Notify ReleaseObjects Reset UseObject XInfo  
  □wi '.....events'   A But if you try to go too far. . .  
OWI OBJECT ERROR: Parent not found (too many '..'s)  
  '.....' □wi 'events'  
    ^
```

The next set of examples shows the use of relative naming in the left argument to `□wi`.

```
' :' □wi 'self'      A Colon shows top-level object  
fmForm  
  ' :selShow' □wi 'self' A Immediate child of top-level  
fmForm.selShow  
  ' ...' □wi 'self' A Grandparent of the current object  
fmForm.selShow.pgB  
  ' ...' □wi '.self' A One dot is optional for an action  
fmForm.selShow.pgB  
  ' ...' □wi '.fHold.self' A Dot is not optional for object  
fmForm.selShow.pgB.fHold  
  ' ...' □wi 'fHold.self' A Object not found (note empty result)  
  
  ' ...' □wi 'fHold.name'  
OWI OBJECT ERROR: Object "fmForm.fHold" not found  
  ' ...' □wi 'fHold.name'  
    ^
```

Note: When the right argument starts with an object name, the system ignores the left argument and works relative to the top level object. The next two examples show that the system is using a child of the top level form, regardless of the existence or absence of the left argument.

```
' ...' □wi 'selShow.self'  
fmForm.selShow  
  □wi 'selShow.self'  
fmForm.selShow
```

You can also use multiple periods within a string in either the left or right argument. The following examples show various legal forms of specifying objects.

```
'fmForm.selShow.pgB.fHold.bnChild...' □wi 'caption'  
Grandparent  
  'fmForm.selShow.pgB.fHold.bnChild....pgA' □wi 'caption'  
Great Uncle  
  □wi ' :...events' A This is the system object.  
  Close Destroy ExitError Help New Prompt Resource SystemChange Tooltip
```

Taking the First Steps

The sections below describe a simple form with some basic controls. Although the example is almost trivial in its simplicity, the technique is the bedrock for applications of enormous complexity. To make it easier to execute the examples, all the steps are included in a log file that is included with APL+Win.

In your APL session, click on the Session Log choice on the Options menu. **IMPORTANT:** Make sure the Append to File option for the Logging Method is selected. Click the Browse button, go to the \APLWINxx\Docs directory, and select Apl_doc1.log. Click the Begin or Restart Logging option and the OK button.

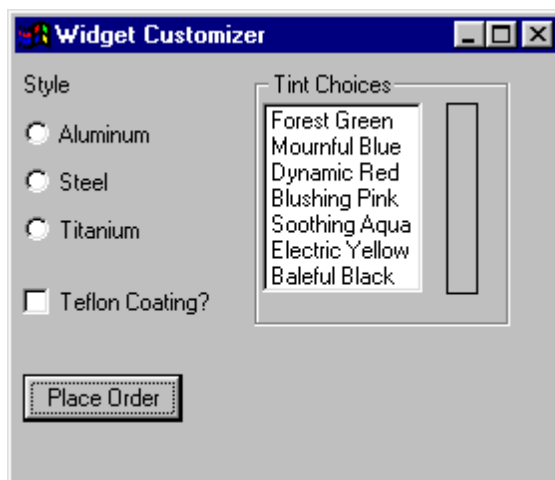
The system reads the file into your session; you can scroll up to see the entire log file. **IMPORTANT:** Before you continue, return to the APL Session Log screen, select Turn Logging Off, and click OK. This will prevent you from extending the log file with your experimentations. If you wish to save your own statements in a log, assign a different file name and click Begin or Restart Logging and OK.

If you don't wish to use the log file, a Make function is listed at the end of the next section. You can reproduce that function to build the form. The event handler functions are also listed.

A Simple Form to Customize Widgets

The example below shows some basics of a user interface, both content and interaction. This assumes the user has somehow expressed interest in widgets, and is presented with the main form. The three radio buttons on the left provide a choice for the user, who can select any one of them. The system responds by filling in the circle. The user can select only one; if a second is clicked, the system unselects the first one by blanking the circle back to white. The square check box allows a single yes/no choice; the system alternately fills in the box with a check mark and blanks it. Clicking on one of the color choices in the list causes a display of that color in the vertical box, a Picture control, beside the list. Clicking the Place Order button causes another form to pop up for a few seconds.

Widget Customizer



The nature of the second form depends on what the user has done to the visible form. If no style is selected, an error form is presented. This is a modal form with only an OK button to dismiss it. If the user has selected a style, an acknowledgement form appears.

On the acknowledgement form, the message differs depending on the style selected. An additional message appears if the check box is checked.

This example demonstrates the basics of an interface. The elements of the form may be accessed in any order, they may be changed any number of times until a final action (in this case clicking the button) is taken, and they all work together for a purpose. Events trigger the display of colors and the showing of the subsidiary forms; the event handlers contain the code that makes these actions happen. The painting event handler invokes a method to draw the color, and the ordering event handler references properties to determine its actions.

The Make Function

The listing below shows a modified form of the Make function that results after you build the form. If you do not use the log file, you can use this function instead to execute the example.

```
▽ fmMain_Make;x;▽wself
[1]  avfmMain_Make -- Created [date / time]
[2]  'fmMain' ▯wi 'Delete'
[3]
[4]  ▯wself<'fmMain' ▯wi 'New' 'Form' 'Close' ('where' 2 1 15 35)
[5]  ▯wi 'caption' 'Widget Customizer'
[6]
[7]  ▯wself<'fmMain.opLight' ▯wi 'New' 'Option' ('caption' 'Aluminum')
[8]  ▯wi 'where' 1.75 0.5 1.5 10
[9]
[10] ▯wself<'fmMain.opStrong' ▯wi 'New' 'Option' ('caption' 'Steel')
[11] ▯wi 'where' 3.25 0.5 1.5 10
[12]
[13] ▯wself<'fmMain.opBoth' ▯wi 'New' 'Option' ('caption' 'Titanium')
[14] ▯wi 'where' 4.75 0.5 1.5 10
[15]
[16] ▯wself<'fmMain.lMetal' ▯wi 'New' 'Label' ('caption' 'Style')
[17] ▯wi 'where' 0.5 0.5 1.5 6
[18]
[19] ▯wself<'fmMain.ckFinish' ▯wi 'New' 'Check' ('where' 7 0.5 1.5 12)
[20] ▯wi 'caption' 'Teflon Coating?'
[21]
[22] ▯wself<'fmMain.fColor' ▯wi 'New' 'Frame' ('where' 0.5 15 8 16)
[23] ▯wi 'caption' 'Tint Choices'
[24]
[25] ▯wself<'fmMain.fColor.lsHue' ▯wi 'New' 'List'
[26] ▯wi 'list' 'Forest Green' 'Mournful Blue' 'Dynamic Red' 'Blushing
[27] ▯wi 'where' 1 0.5 6.6 10
[28] ▯wi 'suppress' 1
[29] ▯wi 'onClick' 'ShowColor (▯wi "value")'
[30]
[31] ▯wself<'fmMain.fColor.picDisplay' ▯wi 'New' 'Picture'
[32] ▯wi 'imagesize' 5.875 1.75
[33] ▯wi 'where' 1 12 6 2
[34]
[35] ▯wself<'fmMain.bnContinue' ▯wi 'New' 'Button'
[36] ▯wi 'caption' 'Place Order'
[37] ▯wi 'where' 10 0.5 1.5 10
[38] ▯wi 'onClick' 'Ordering'
[39]
[40] 'fmMain' ▯wi 'Show'
▽
```

The ShowColor Event-Handler Function

This two-line function causes the system to draw the appropriate color on the picture control when one of the items in the list is selected.

```

▽ ShowColor num;shades
[1] shades←⌈(num=17)/(0 159 0) (127 63 255) (255) (255 127 127)
    (0 255 255) (255 255 0) (0 0 0)
[2] '..' ⌈wi '.picDisplay.Draw' ('Brush' 1 shades) 'Clear'
▽

```

The Ordering Event-Handler Function

This function pops up one of two forms, either the error message box or the order acknowledgement, when the user clicks the button on the main form. This function is basically only a stub for the many actions that would be required to actually accept an order. It would need to be expanded to be more realistic, but it does show a legitimate framework for an application.

```

▽ Ordering;style;garnish;sink
[1] :if 1=('fmMain.opLight' ⌈wi 'value')
[2]   style←'cheap'
[3] :elseif 1=('fmMain.opStrong' ⌈wi 'value')
[4]   style←'moderate'
[5] :elseif 1=('fmMain.opBoth' ⌈wi 'value')
[6]   style←'costly'
[7] :else
[8]   sink←'ERROR' ⌈wi 'Create' 'Form' ('size' 5 18) ('border' 3)
[9]   sink←'ERROR.bnOK' ⌈wi 'New' 'Button' ('where' 3 1.5 1.5 4)
    ('caption' 'OK') ('style' 2)
[10]  'ERROR' ⌈wi 'Show'
[11]  'ERROR' ⌈wi 'Draw' ('Text' ('You must pick a style',⌈tcln,
    'Click a style choice') .5 1)
[12]  sink←'ERROR' ⌈wi 'Wait'
[13]  'ERROR' ⌈wi 'Delete'
[14]  →0
[15] :end
[16] garnish←('fmMain.ckFinish' ⌈wi 'value')
[17] sink←'fmOrderForm' ⌈wi 'Create' 'Form' ('where' 2 40 9 30)
    ('caption' 'Widget Order Form') 'Hide'
[18] sink←'fmOrderForm.lBanner' ⌈wi 'New' 'Label' ('where' 1 1 3 21)
[19] :select style
[20] :case 'cheap'
[21]   'fmOrderForm.lBanner' ⌈wi 'caption' 'Your order has been placed'
[22] :case 'moderate'
[23]   'fmOrderForm.lBanner' ⌈wi 'caption' 'Thank you for ordering
    this outstanding product'
[24] :case 'costly'
[25]   'fmOrderForm.lBanner' ⌈wi 'caption' ('Oh, most honored client,
    we are happy to fulfill your request')
[26] :end
[27] 'fmOrderForm' ⌈wi 'Show'
[28] :if garnish=1
[29]   'fmOrderForm' ⌈wi 'Draw' ('Color' (255 255 255) 255)
    ('Text' ('expedite' 4 5))
[30] :end
[31] 'fmOrderForm' ⌈wi 'Draw' ('Text' 'Order Info to be added' 5 5)
[32] 0 0 ρ⌈dl 5
[33] 'fmOrderForm' ⌈wi 'Delete'
[34]
[35] 'fmMain.opLight' ⌈wi 'value' 0
[36] 'fmMain.opStrong' ⌈wi 'value' 0

```

```
[37] 'fmMain.opBoth' □wi 'value' 0  
[38] 'fmMain.ckFinish' □wi 'value' 0  
[39] 'fmMain.fColor.lsHue' □wi 'value' 0  
[40] 'fmMain.fColor.picDisplay' □wi 'Draw' 'Clear'
```

▽

Purposes of the Controls in the APL+Win Interface

It is useful to observe the roles that various objects play or can play in an interface. The short descriptions given below give an indication of how these controls are often used. Imaginative programmers have found many more ways to implement their applications.

Passive Controls

Some controls are primarily passive; they allow the display of information to the user, but provide little opportunity for interaction. The `Label` in the upper left corner is such an object. A label conveys information by virtue of the caption you assign or whatever you draw on it, but the user cannot modify it.

Similarly, a `Picture` object can display and store a myriad of shapes, forms, images, colors and words. You can arrange to allow the user to scroll an image that is larger than the visible area on the screen, but the only direct user input you can detect are mouse clicks. A `Frame` object conveys information not only by its caption, but also by the outline that encloses and therefore associates a group of controls. A frame is also useful to you as a programmer as a container. You can make other controls children of the frame; when you move or hide the frame, all its children automatically move or are hidden as well.

The `Progress` bar is another primarily passive control that allows you to convey to the user that constructive action is going forward. In addition to these passive controls, most visible controls have properties that allow you to convey information. You can assign a tooltip, for example, that pops up an ephemeral window with a text message; this is often used for a hint about how to use the control.

Container Controls and Objects

There is another group of objects that provide only limited user interaction; however these are anything but passive. They function primarily as containers for groups of controls that the user does access. These container controls allow the user to move, reorient, resize, hide, show, or otherwise manipulate the container and its associated controls as he or she works with the application or with multiple applications.

The `Form` is the most used of these objects. A form can be the primary container of an entire application, a dialog box, an editing window, a graphics window, a floating toolbox, or a simple message box. An `MDIForm` is primarily a container for forms. MDI stands for Multiple Document Interface where "document" is used in the Windows sense that corresponds to one instance of the application's major use.

You use an MDI form when you want to contain multiple, independent, primary forms, such as documents in a word processing application. (Note that Microsoft has gone away from the MDI in its word processing application in favor of a kaleidoscopic document interface that you would be well advised not to emulate.) The MDI form allows you to display multiple forms in a tiled, cascaded or stacked arrangement and to move among them easily.

A `Selector` object is also a container of containers; a selector contains multiple `Page` objects, each of which is a container of controls. It can also contain controls that apply to whichever page is active. A `CommandBar` is also a container of controls that the user can position on an edge of a larger container, such as a form, or cause to float anywhere on the screen.

Invisible Objects (not strictly Controls)

There are several classes of objects that do not have a visible presence on the screen in an application; they are generally not designated as controls. These objects, along with `Form` and `MDIForm`, can be top-level objects; that is, they can be created without a named parent; some of them can also be children of other objects.

The `Printer` object allows you to formulate output for any printer that is installed on the system.

A `Timer` object allows you to trigger an event at a predefined interval.

An `Imagelist` is a collection of images; the images can be displayed on the screen individually as properties of other controls, but the `imagelist` itself is never visible.

The system object, which you designate with the octothorpe (`#`) in `APL+Win`, is a utility class of which there is only one instance. You use it to control high-level functions including closing the application. You can think of it as the default parent of top-level objects; you can also think of it as representing the entire screen.

Controls that Allow a Single Choice or Action

A variety of controls allow the user to make a single choice or to take a specific action. The action is usually clear-cut and the application should make it obvious to the user what the choice entails. The `Button` control is perhaps the most commonly used. Activating a button, by clicking with the mouse, by pressing the `Enter` key or the spacebar when the focus is on the button, or using a keystroke combination, usually causes an application to perform some action indicated by the caption on the button itself. The button exhibits a visual change that makes it appear to be pressed down and then returned to its original state. A style of button, called a toggle button, allows a persistent state of being pressed, to indicate a semi-permanent choice. Buttons can have images in addition to or instead of their captions. You can change images to indicate state changes.

Menu items function similarly to buttons. Activating a menu item in any manner either causes a program action or cascades a string of further menu choices that lead to an action. You create the hierarchy of menu items and their captions (or images) and program the actions they represent. The cascading behavior is built in to the system. Menu items that cascade choices, unlike buttons, allow you to tailor them between the time the user clicks and the child menus appear by virtue of a `DropDown` event. You can also program menus to pop up anywhere in the application.

A `Check` control also allows a choice that is persistent; that is, unlike a button or menu where the response is immediate and the control resumes its existing state, a check box gives a visual indication that it has been checked. Its `value` property reflects the checked/unchecked state of the control. You can also use the `onClick` event handler to program an action at the time the control's state changes, or you can simply read the value at a later time.

An `Option` control also allows a persistent choice, and it gives a visual indication that it is selected. Options, also called radio buttons, differ from check boxes in that they work in groups; only one of the group may be selected at a time. If the user clicks one that is not currently selected, the system unselects the previously selected one at the same time it selects the new one. Options also differ internally from check boxes in that they trigger a `Click` event only when selected, not when unselected.

A `CommandButton` is a special button that is a child only of a `CommandBar`. It functions much as a normal button, but it has a flat appearance, which changes when the user simply moves the mouse cursor above it, even without clicking. You can display multiple images on a command button; which one is displayed depends on the state of the button. A command button can be a toggle button; you can also create a group of

command buttons that not only toggle, but work in concert like radio buttons, where only one of the group may be selected at a time.

A Toolbox is a single control that allows you to assign multiple images, each of which can act like an independent button when clicked. This feature is now largely obsolete. Using a command bar and sufficient command buttons is recommended for new applications.

Controls that Allow Multiple Alternatives

When choices must be made from a more complex set of alternatives, other controls are appropriate. A List control allows a simple ordered list of any length to be displayed to the user. You can decide whether only one item may be selected at a time or if the user may have multiple items selected simultaneously. A user can scroll a list that is longer than the visible window on the screen. When the focus is on the list, pressing a letter key causes the highlight to move to a choice that matches the letter. A Combo control also allows a list in conjunction with an edit box (see below).

A Tree control allows a graphical representation of a hierarchical list. A user can expand or collapse any portion of the tree to view or explore the parts that are of interest. The left side of the Windows Explorer that shows the directories on a computer is an example of a tree.

A Listview control allows the display of large quantities of nonhierarchical information. The basic set of items can be represented by captions or images or both. In addition, you can provide many columns of associated information. Often, the user can reorder the columns and manipulate the display in a variety of ways. The right side of the Windows Explorer that shows the contents of the drive or folder highlighted on the left is an example of a listview.

Controls that Allow a User to Type Input

When you cannot specify the choices, you may need to have the user type input. An Edit control allows the entry of alphanumeric information from the keyboard. An edit can be a simple one-line box of indeterminate length, as for a name, or of fixed length, as for a social security number. You can also choose to allow multiple lines and paragraphs of great size.

A Combo box is a hybrid control that combines a one-line edit box with a list (see above). The user can either choose from the list or enter a choice by typing.

A RichEdit control allows a rich variety of formatting options to enhance the entry functionality of an Edit control. The difference between a RichEdit control and an Edit control is like the difference between WordPad and Notepad.

Controls that Represent a Numeric or Physical Range

Some controls allow the user to physically manipulate an object to move through a range of some kind. The Spinner control consists only of two arrows pointing in opposite directions. Clicking the mouse or holding it down on one of the buttons causes the spinner to cycle through a range of numeric values. You can display the values by using a handler or by associating the spinner with another control, such as an edit box. You can also associate the spinner values with a list or another range of choices.

A Scroll bar allows the user to slide a block along a channel to select a position; your handler determines how the corresponding screen images appear to move in conjunction with the movement of the slider. You can also use a scroll bar as an analog input device.

A Trackbar operates like a scroll bar, but it has marks along the channel that represent discrete values, which the user selects by moving the slider to the appropriate position.

Special Purpose Controls

Finally, there are controls that are special purpose rather than general purpose. A `DateTime` control allows you to display or drop down a calendar from which a user can pick a date by clicking, or to display a date and/or a time in a box that resembles an edit control, using a variety of formats. The user can alter the content to select a date or a time.

The `Media` control allows you to display a variety of special file types on the screen and it allows the user to operate any one of several devices that attach to a computer.

There are also a vast number of commercial controls that use the ActiveX standard, which you can invoke from APL+Win as though they were part of the Windows interface.

DPI-Awareness

APL enables system-wide DPI scaling in the session manager window and the `□wi` (Windows Interface) system objects (excluding any ActiveX controls and objects) when started with a manifest file (e.g., `aplw.exe.manifest`) in the same folder specifying the entry colored in red below:

```
<?xml version="1.0" encoding="UTF-8"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
<asmv3:application xmlns:asmv3="urn:schemas-microsoft-com:asm.v3">
  <asmv3:windowsSettings xmlns="http://schemas.microsoft.com/SMI/2005/WindowsSettings">
    <dpiAware>true</dpiAware>
  </asmv3:windowsSettings>
</asmv3:application>
...

</assembly>
```

In Windows 8.1 and newer operating systems, when `true` is replaced with `false`, APL allows the operating system to do the image stretching for Windows that are drawn as if running at 100% DPI. This is the same behavior as when starting APL without a manifest file present in the same folder. Note: The DPI scaling percentage of the system is the represented by the slider setting for the "*Change the size of text, apps, and other items*" selection for the primary monitor in the Windows Display settings window.

When APL is running in `dpiAware=true` mode, most things "just work" automatically, but there are a few things that might not work as expected. In all of the examples below, the system-wide DPI scaling is set to 200%. What this means is that everything drawn by APL will be twice as tall and twice as wide (in pixel units) as it was when running at 100% DPI scaling. But all of the scale settings used on `□wi` objects automatically scale to this larger size. So, if you normally work in character units (`scalemode = 1`), you will find that 10 units of height will now correspond to twice as many pixels of height as before. But since you normally run 200% DPI scaling on a monitor with twice as many pixels as at 100% scaling, the objects you create will look about the same size as they did before, when running in a DPI unaware application. Traditionally one character unit was 16 pixels high by 8 pixels wide. So a form that was sized at 10 20 will be 10 20 x 16 8 = 160 160 pixels in size at 100% DPI. But at 200% DPI this form that is sized at 10 20 character units will be 320 320 pixels in size. If you are used to working in points (`scalemode = 3`) units then you are used to each point being 1.3333 pixels on the screen, but at 200% DPI each point will be 2.6667 pixels on the screen. If you used to work in pixels (`scalemode = 5`) these units are also affected by the DPI scaling. So if you create a `□wi` object that is sized at 200 by 400 pixels high and wide, then when running at 200% DPI this will actually be drawn at 400 x 800 pixels in size. All of the traditional scaling units scale by the DPI percentage you are running at. If you

reference the '#' `owi 'units'` property, this will show you exactly how large each unit is in terms of pixels. So for example, while in previous versions of APL you always saw the following:

```
'#' owi 'units'
16      8
2        2
1.333333333 1.333333333
0.0666666667 0.0666666667
1        1
```

When running this version at 200% DPI you'll find you now get the following reported for the units of the system:

```
'#' owi 'units'
32      16
4        4
2.666666667 2.666666667
0.1333333333 0.1333333333
2        2
1        1
```

Note carefully that in past versions, the units property returned 5 rows of data corresponding to character units, dialog units, points, twips, and pixels. But in this version units returns 6 rows corresponding to character units, dialog units, points, twips, virtual pixels, and true pixels. The last row corresponds to the new `scalemode = 6` mode setting that allows you to work in real pixels rather than virtual scaled pixels that you get from traditional `scalemode = 5`. You can override the virtualization of `scalemode = 5` so that it also refers to real pixels (the same as `scalemode = 6`) by setting `VirtualPixels=0` in the [Config] section of the INI file. However, you are advised to use caution in doing so since this will cause visual elements of your application whose layout is defined using pixel units (`scalemode = 5`) to not scale proportionally with other visual elements specified in other scale units.

Because all traditional `scalemode` units (1-5) are virtualized by the current DPI scale setting, this means that all `owi` layout calculations work exactly as they did before, relative to each other. But their correspondence to actual pixels on the screen has changed. But since fonts have scaled in exactly the same way as character units, for most applications, most layout related things work exactly as they did in the past. But there are some places where applications can experience issues. One example is if they specify a font using the LOGFONT notation (14 element argument to the font property). In that case, the LOGFONT is specified in terms of pixels, and these pixel units are not scaled. They correspond to the Windows LOGFONT data structure which expects all size measurements to be in terms of pixels. Similarly, if you use low level Windows APIs with `owcall` using window handles, any windows you access are in terms of actual pixels, not virtualized (`scalemode = 5`) pixels. So you are on your own in terms of scaling these values to the current DPI setting. Similarly, some ActiveX objects created may expect you to specify column widths, row heights, font heights in pixel units. And if they do, those are real pixels, not virtual pixels. The APL.Grid object is such an ActiveX object. It does not automatically scale the pixel units you specify for cells or font sizes. So you must compensate manually by scaling pixel setting by `('#' owi 'dpi')[2]÷100`. Even though the APL.Grid ActiveX object is distributed with APL, because we have no control over other ActiveX objects with regard to how they behave with respect to DPI scaling, the decision was made to not support automatic scaling of any ActiveX objects.

A final DPI issue involves scaling of images such as bitmaps, icons, and cursors used in the application. As with `owi` object layout units and font scaling, this too happens automatically, with rare exceptions that are noted below. Icons and cursor are the easiest to describe. Windows automatically scales them to an appropriate size as a function of system-wide DPI scaling. So they are drawn at the default icon and cursor size that Windows chooses for these UI (User Interface) elements as a function of DPI setting. However, bitmaps are more complex. In general, any bitmaps that you reference by name or DLL image resource notation are automatically scaled appropriately. If a bitmap used in a toolbar imagelist is designed as 10 pixels high and 20 pixels wide per image, and has 5 images, its total dimension will be 10 pixels high by 100 pixels wide, in the bitmap file or DLL image resource. But when this bitmap is used to in an APL application that is running at 200% DPI, it will automatically be scaled such that it is 20 pixels high by 200 pixels wide, and the width of each image will be 40 rather than 20 pixels wide. You would normally use such a bitmap in an

Imagelist object and specify its size with the `imagesize` property, to tell the Imagelist how wide each image is. The `imagesize` returns pixel size of the image as corresponding to the bitmap as it existed in the bitmap file or DLL image resource. So while `imagesize` continues to be 10 20 for this particular example, internally the image has been stretched to 20 40 in size. If you think of the pixel units used in the `imagesize` property as being in scale 5 units, you'll be correct in thinking of them as being virtualized internally to the correct number of DPI scaled real pixels required to render the image to appear as it always has in relation to other element sizes (that have correspondingly been scaled by the DPI setting by exactly the same amount).

In some rare cases you might work with bitmap handles that you obtain from other means, perhaps from some ActiveX object that creates and draws into a bitmap for you. In this case, you will often be working with a bitmap that was scaled as it should be displayed at 100% DPI. That's the default assumption APL makes for working with such bitmap handles. If you pass a bitmap handle to the `AddImages` method of an Imagelist object, APL will assume the bitmap was drawn for 100% DPI viewing and scale it up automatically to the actual DPI scale that the application is running at. The same thing happens if you use the `Draw` method with the `Bitmap` operation that uses a bitmap handle as an argument (rather than a bitmap file name or DLL resource specification). In that case, the bitmap will be scaled up to the actual DPI scale and draw onto the object you specified.

There may be times when automatic re-scaling of bitmap images is not what you want to happen. You can control this image scaling behavior by using the `imagescale` property. All Window based `Ωwi` objects have this property (such as `Form`, `Picture`, `Label`, etc.) as well as the Imagelist object. The `imagescale` property specifies the assumed scaling of the image source file, DLL image resource, or bitmap handle that you are using to draw or load images from. The default is 100 which means that APL will assume that images are scaled for 100% DPI rendering. In this case, when running at a different DPI scale than 100%, the system will automatically re-scale the image for the DPI scale you are running at.

You can disable automatic image scaling by specifying `imagescale` as 0. In that case, APL will not re-scale any bitmaps it draw or loads for that particular object that specifies `imagescale = 0`. Use this setting to suppress automatic image scaling for a particular object. If you have an image that it actually drawn for DPI viewing scale other than 100%, you can specify that percentage, and then if you are running APL at a different scale, it will be automatically scaled up or down as needed to the DPI level you are running at.

There is an important exception for the `Picture` object's image property. When you reference this property it returns a matrix of color codes corresponding to each actual pixel in the picture. If you are working in `scalemode = 6` (true pixels) the shape of the image property and the value of the `imagesize` property will be the same. But for other scale units, the `imagesize` property returns the scaled units corresponding to the actual pixel size of the image drawn on the `Picture` object. In the past this was true for `scalemode = 5` (virtual pixels) and `imagesize`. But now that scale 5 units are virtualized, the shape of the image property's return value will not match the `imagesize` property value unless you are running at 100% DPI. The only time the scale size matches the shape of the value returned by referencing the image property is when you are running with scale 6 units. The `imagescale` setting does not have any effect on the value returned by the image property because the image property is always related to the actual image pixels on the screen.

But there is a tricky issue that can arise if you use the image property to save images into a numeric matrix. Let's say you draw the image during one APL session and save it (into an integer matrix) and then in another session you set the image property on the `Picture` to restore the previously drawn image. This all works perfectly unless the DPI scaling in effect when you saved the image is different than when you restore the image. Let's say you were running at 200% DPI and created a `Picture` object that was 10 x 20 character units high and wide. This could correspond to an image value of shape 320 320 (because that's the number of pixels that correspond to 10 x 20 characters at 200% DPI). So let's say you save that image matrix into a file. Later you start APL running at 100% DPI, perhaps on a different computer or simply because you changed the system DPI scaling. When your application now runs your `Picture` object will still (most likely) be 10 x 20 character units in size (assuming you are work in character units) but because of the change in DPI scaling, it will now only be 160 x 160 pixels in size. If you set the `Picture` object's image property with the shape 320 320 matrix you saved from the previous session, you will only see the top-left quadrant of the image, and the rest of the image will be off screen. If you have enabled scrolling on the `Picture`, you'll be able to view the other

parts of the image. Or if you have set the style 4 for the Picture, the image will be stretched to fill the available display area, and you'll be able to see it completely.

So, as described above the `imagescale` doesn't have any effect on the image property insofar as referencing the property or when setting it to a numeric matrix value corresponding to image color values. However, you can also specify the image property as a bitmap resource name. In that case the image will be scaled normally by the `imagescale` property, as described earlier. However, when you then reference the image property, it will return a matrix value containing the image color matrix pixel values. So in this case, the `imagescale` does have an effect on the image property, but only when the property is set as a character vector bitmap image resource specification. Otherwise the image property ignores the `imagescale` setting.

There is a similar behavior with respect to the Picture object's bitmap property. You can specify it as a color matrix similar to the image property, and if you do that, the `imagescale` property is ignored by the bitmap property when loading the bitmap into the Picture background, or when you reference the bitmap property. But when you specify the bitmap as a character vector bitmap resource specification, in that case the `imagescale` does affect how the background bitmap is scaled when loading into the Picture. But in this case, when you reference the bitmap property, it returns the character vector you set rather than returning the color matrix.

Chapter 6: Programming the Windows Interface: Basic

There are a wide variety of tools and techniques you can use in developing the interface to a Windows application. Nobody will use all of them in any one application; indeed no one may ever use all the techniques available. This chapter displays a few of the techniques that are commonly used in a variety of applications. It assumes you know the basics of creating objects and writing an event handler to perform a useful activity. As an introduction, some terms and concepts are explained.

Working with Objects

In order to understand the techniques of dealing with user input, you must understand the basic terminology of objects in the APL+Win Windows Interface.

System Handling of Windows Objects

Windows objects are stored in a section of memory that is separate from the active workspace. You bring an object into existence with the `New` or `Create` method, and it continues to exist in APL+Win until you delete it with the `Delete` method (or use `Reset` on the system object) or until the APL session ends. Because the existence of objects is independent of the APL workspace, your forms continue to exist when you clear the workspace or load a new workspace. The callback code that a form invokes, however, **is** in the APL workspace; therefore, you will usually want to delete forms before you load a new workspace.

An object can be either *open*, meaning that it is active and capable of responding to events, or *closed*, meaning that it is inactive, not visible, and not available to respond to events. By default, a newly created object is open. You can close the object by invoking its `Close` method and reopen it with the `Open` method.

When an object is closed, it exists in APL+Win, but is not recognized by the operating system. A Windows object has a presence in the Windows operating system that is not independent of, but is separate from, its existence in APL+Win. If you delete an object from APL+Win, the system deletes it from Windows. However, you can eliminate the object from the Windows operating system while retaining its definition in APL+Win. When an object is in this state, a user cannot manipulate it, but your program can perform many of the actions on it that it can when the object does exist in the operating system. For example, you can reference or set most properties of an object while it is closed. This provides a means for you to change the state of an object without producing events or visible effects on the screen. (Some actions do require the existence of the object in Windows.) APL+Win can recreate the object in Windows from the stored definition, and the user can manipulate it as though it had continued to exist uninterruptedly. You can determine if an object is open or closed by referencing its `opened` property. You must use a method to change this state however; the `opened` property is read-only.

When an object is open, it can also be *visible* or *hidden*. The `visible` property indicates whether an object is currently visible. Objects are capable of receiving events whether they are visible or hidden, but user-generated keyboard and mouse events can occur only for visible objects. A hidden form is recognized by the operating system; for example, it might still receive events triggered by a timer. Or, your program could fire the `onClick` event handler of a hidden button by invoking the `Click` method, but the user cannot click the button with the mouse. You can make an object invisible using the `Hide` method or by setting the `visible` property to zero.

Similarly, an object can also be either *enabled* or *disabled*. When it is enabled (and open and visible), it is ready to accept user input. Setting the `enabled` property to 0 causes the image of the object to appear in a gray color; the user can see the object but not manipulate it. It is common practice to enable and disable objects dynamically during the course of execution depending on changes in context. For example, a Paste menu item is normally disabled when the Windows clipboard is empty.

Understanding the Concept of a Closed Object

Note that the properties mentioned above, `opened`, `visible`, and `enabled`, can be set or referenced independently. (You can set the `opened` property only indirectly, by using a method.) All three must be set to 1 for the object to be accessible to the user. However, there is a fundamental difference with the state of an object that is open or closed.

When an object is open, the other two properties, and indeed most of the other dozens of properties, represent the state of the object. When the object is closed, however, the other properties are only potential; they show what the object will be like when it is reopened (that is, when it is re-created in Windows). You can reference the `visible` property of an object that is closed; it may show 1 or -1 (for most objects); similarly a closed object can show that it is enabled or disabled. In the same way, you can reference the `size` of an object or its `color`. All of these properties reflect the potential state of the object. When you open it, then these properties will have an actual manifestation. You cannot see a closed object regardless of whether its `visible` property returns a positive or negative 1, because it does not really exist in Windows. Its `opened` property reflects the current state of its existence in Windows.

This difference can be detected by referencing the properties of child objects. Closing a form has the effect of closing all the children of the form; opening the form similarly opens all of its children. The `opened` property for each of them changes when the parent is closed or opened. Hiding a parent does NOT, however, change the `visible` properties of its children. When you hide a form, you cannot see any of the children of the form; yet their `visible` properties return 1 when referenced. Similarly, disabling a frame will cause all the controls that are children of the frame to be greyed out and unresponsive; yet their `enabled` properties return 1.

If you set the `visible` or `enabled` property of a child to zero, showing or re-enabling the parent does not automatically show or re-enable the child in the same manner as opening a parent reopens all the children. If you disable or hide a child either before or after disabling or hiding a parent, you must change the properties on both parent and child to reactivate the child. This fundamental difference is also appropriate for the different uses of these properties; generally, you open and close forms, or containers of objects (for example, when using the `def` property), not individual controls. It is much more common to change the `enabled` and `visible` properties of individual controls.

The `Hide`, `Print`, `Show`, and `Wait` methods invoke the `Open` method if the object is not already open.

Determining the Contents of an Object

You can query the contents of an object under program control by referencing the following read-only properties:

- Referencing the `state` property yields a 2-column nested matrix of all the properties with nondefault values, which includes event handlers. Because objects tend to have a large number of properties that are almost always left to their default value, the `state` property provides the most succinct and useful information about an object. For example,

```
ps←'fmMain.bnOK' ⍵wi 'state' ⋄ s
5 2
caption      OK
class        Button
name          bnOK
onClick      fmMainOK
where 9 30 1.5 6
```
- Referencing the `children` property yields a vector of character vectors containing the names of all the children in an object, including the top level menu items. By examining the `children` properties of each of those objects recursively, you can determine the complete hierarchy of a form. For example, to obtain a list of all the first-level descendants in form 'fmMain':

```
'fmMain' ⍵wi 'children'
fmMain.mFile fmMain.mEdit fmMain.mOptions fmMain.bnOK fmMain.btnCancel
```

You can obtain a list of all the top-level objects that currently exist by referencing the `children` property of the system object.

```
'#' ⍵wi 'children'
fmMain prPrinter tTimer
```
- Referencing the `class` property yields a character vector with the name of the object's class. For example:

```
'fmMain.bnOK' ⍵wi 'class'
Button
```
- The `properties` property is a vector of character vectors that contains the names of all the properties that an object has, whether or not they have been assigned nondefault values.
- The `events` property is a vector of character vectors that contains the names of all the events that can apply to an object, whether or not handlers have been defined for them.
- The `methods` property is a vector of character vectors that contains the names of all the methods that can be invoked on an object.

Certain classes have other read-only properties that describe characteristics appropriate only for that class; for example, an `Imagelist` object has an `imagecount` property that returns the number of images the `imagelist` contains.

In addition, the system object has a number of read-only properties that return information about the APL+Win system or your configuration.

Positioning and Sizing Controls

You position controls on a form, or other parent object, by setting the child control's `where` property. This property is a two or four-element numeric vector; the first two elements specify the row and column position of the upper left corner of the control in the scaling units of its parent. By default, this is character units, whose height is twice their width.

You can change the scaling units by changing the parent's `scale` property; you have the choices of dialog units (one-eighth character height), points, twips, pixels, or proportional scaling. You can calculate the conversions among scales by using the `units` property on the system object.

You specify the height and width of the control with the third and fourth elements of the `where` property; alternatively, you can specify these two elements in either the `size` or `extent` property. There is a slight difference between using `where` and the other two properties; the last two elements of `where` specify the outer dimensions of an object, while `size` and `extent` specify the inner dimensions. For a form, the title bar and the border are excluded from `size` and `extent`. For controls that have no border, there is no difference.

The `extent` property is measured in the scaling units of the parent, as is `where`; the `size` property is measured in scaling units of the control itself. By default these are the same; they differ only when you explicitly change the `scale` property of either the parent or the child control.

The system keeps the three properties, `where`, `extent`, and `size`, synchronized whenever an object is open. If an object is closed, you can set any of the three properties; however, if you reference them, what the system returns depends on which one was set most recently. If you set `where`, the system returns it as a four-element vector and returns an empty vector for `extent` or `size`. If you set either `size` or `extent`, the system returns their values, but returns only a two-element vector for `where`.

Positioning and Sizing Forms

You position forms on the screen in the same manner as you place controls on a form, by setting the `where` property. The scaling units for the screen are always characters; you can determine the size of the screen by referencing the `workarea` or `size` property of the system object.

Forms, and `MDIForms`, are more complicated than controls, because they have multiple states when they are open. A form can be minimized, which means it has only an icon and a title bar displayed, either on its parent or on the system task bar, or it can be maximized, which means it takes up the entire area of its parent, which is the entire screen for a top-level form. When a form is in between minimized and maximized, it is said to be restored. Moreover, forms are much more likely to change state since users can minimize, maximize, or close a form by clicking one of the system boxes, whereas the program must set the state of a control.

You can use the `where`, `extent`, and `size` properties for forms just as you do for controls. In addition, you can use the `place` property for forms and `MDI forms`. When a form is minimized or maximized, you can set the `place` property to specify the location and size you want the form to have when it is restored. In addition to numeric values, you can use the equal character (`'='`) to indicate the element should not change, or you can use the question mark (`'?'`) to indicate the system should use the default setting.

When a form is closed, or when it is in a restored state, `place` is identical to `where`; setting one changes the other. The interaction with the `size` and `extent` properties for a closed form is the same as described above for controls.

When you reference the `place` property, it always returns a four-element vector. It may show the equal character or question mark. **Warning:** In future systems, the `place` property may have more than four elements. If you reference it with the intention of using the values in your program, you should use a statement like `(r c h w) ← 4 ↑ 'fmMain' □wi 'place'` rather than the simple statement `(r c h w) ← 'fmMain' □wi 'place'`

Additional values would be catenated to the existing vector, so it is strongly suggested that you avoid statements like `(h w) ← -2 ↑ 'fmMain' □wi 'place'`

Fundamental Aspects of a User Interface

After you create a form and make it visible on the screen to the user, there are some fundamental aspects of the interaction between the user and the interface that are part of any application. Presenting the choices and directing the navigation through the interface is a basic requirement. Even though a user can theoretically do anything with a visible form, there are some practical design choices that make it easier for the user to take purposeful steps. Once the user makes a choice to take a step, your application must recognize and react to user input from either the keyboard or the mouse. The basics of these concepts are presented in this section.

Menus and Shortcuts

Menus are the traditional, and efficient, technique for presenting the primary choices to a user. You can group menu items in meaningful associations, with a top-level caption that represents the group. When the user is interested in the grouping, clicking on the top level menu drops down the list for viewing or selection. When the user makes a choice, or declines to choose, the menu closes up leaving only the menu bar visible. You can cascade more choices from second- or third-level menus, and you can tailor your menus to a current situation. In addition to captions that describe each menu choice, you can place images on menus to enhance or replace the verbal cue.

Note that in APL+Win, each item in a menu is an object of the Menu class, as is the top-level caption for the entire "menu." The menu items are children of the caption; cascading menus are children of a menu item.

The menu bar that appears at the top of a form is defined by objects of the Menu class. For example, the following statement adds two items to the top level menu bar of a form:

```
m1←'fmMain.mFile' □wi 'New' 'Menu' ('caption' '&File')
m2←'fmMain.mEdit' □wi 'New' 'Menu' ('caption' '&Edit')
```

The menus that drop down when a user selects a menu bar item are defined as children of these top-level menus. The following statement adds three items to the File menu:

```
m3←'fmMain.mFile.mOpen' □wi 'New' 'Menu' ('caption' '&Open')
m4←'fmMain.mFile.mClose' □wi 'New' 'Menu' ('caption' '&Close')
m5←'fmMain.mFile.mExit' □wi 'New' 'Menu' ('caption' 'E&xit')
```

The hierarchy of menus can be arbitrarily deep, although menus more than three levels deep are seldom used. The following statement adds a submenu of two choices to the mFile.mExit item:

```
m6←(m5, '.mSave') □wi 'New' 'Menu' ('caption' '&Save changes')
m7←(m5, '.mNoSave') □wi 'New' 'Menu' ('caption' 'Do &not save')
```

The sequence of menu objects at any point in the hierarchy is determined by the order property for those objects. By default, the order property is assigned successively as you create menu objects.

```
      m3 m4 m5 □wi''c'order'
1 2 3
```

You can modify the order of items in a menu by setting the order property. You can insert a new item into a menu by giving it a fractional order value. To insert a Print item between Close and Exit:

```
m8←'fmMain.mFile.mPrint' □wi 'New' 'Menu' ('order' 2.5) ('caption' '&Print')
```

The menu items are then automatically renumbered sequentially starting with 1:

```
      m3 m4 m8 m5 □wi''c'order'
1 2 3 4
```

Menu items that hold other menu items are not capable of producing events; their function is to collect other menu items into a group. Bottom-level menu items generate Click events when the user selects them, for which the system executes onClick handlers. For example:

```
m8 □wi 'onClick' 'PrintReport'
```

Menu Separators

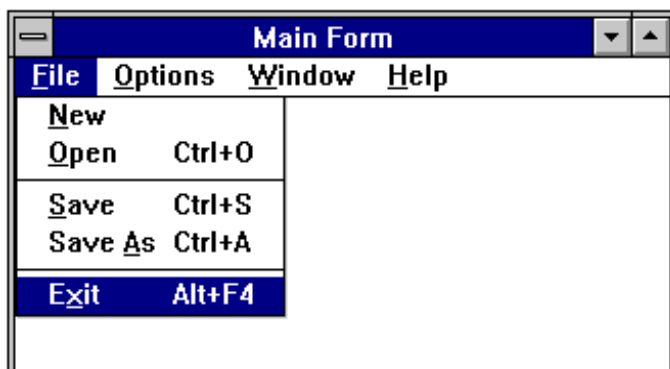
A separator is a horizontal line across the drop-down part of a menu. You use it to separate a logical grouping of menu items. Separators are Menu objects with the separator property set to 1. A menu item cannot serve as a separator if it is part of the top-level menu or if it has any children. When this property is set, the item is simply a line on the menu, independent of any of its other properties. For example, the following statements create a separator just before the Print item of the File menu:

```
OrderNum ← 'fmMain.mFile.mPrint' □wi 'order'  
'fmMain.mFile.mSep' □wi 'New' 'Menu' ('separator' 1) ('order' (OrderNum-.5))
```

Menu Access Keys

By convention, Windows programs allow users to perform almost any action with the keyboard that they can perform with the mouse. It is part of the underlying system that pressing and releasing the Alt key activates the menu bar, which allows the user to navigate among the menus using cursor keys. You can also provide the facility for typing single-character access keys that have the effect of clicking the menu item. For example, the form shown below has a menu that causes the key sequence Alt, F, X to select the Exit action from the menu.

A Menu with Access Keys



The ampersand character (&) in a menu item's caption indicates that the character following is the item's access key. This key allows a user to select the menu item without using the mouse. The key sequence works because the captions are '&File' and 'E&xit', respectively. Note that you should have at least one control (even a hidden one) other than the menu on a form for Windows to process access keys properly.

Shortcuts

It is also possible for a single keystroke (usually a keystroke combination produced with the Alt or Ctrl keys) to cause the same action as clicking a menu item or pressing an access key; these combinations are called shortcuts. Shortcuts have the effect of generating a Click event on one of the form's menu items. Some shortcut keys are common Windows style conventions, such as the use of Ctrl+X, Ctrl+C, and Ctrl+V for the Cut, Copy, and Paste editing actions, respectively.

The shortcut property of a menu item defines its shortcut key; you specify it as a two-element vector. The first element is the character key value or numeric Windows virtual keycode of the key you want to

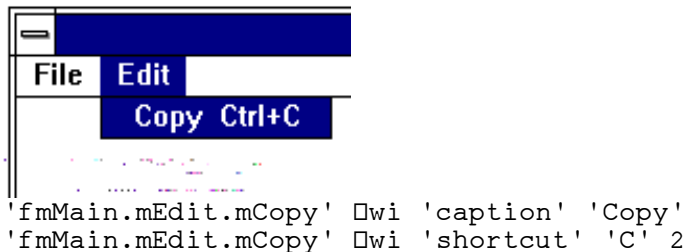
designate; the second element is the sum of the values that indicate the combination of Shift, Ctrl, and Alt keys that must be held down when the key is pressed. The values are:

Shift Key Values

Value	Description
0	Base key alone
1	Shift
2	Ctrl (the default)
4	Alt

Menu items that have a shortcut key should include a hint to that effect in their captions. When you define the shortcut property, the system automatically adjusts the caption to include a representation of the shortcut. The hint appears right-justified in the menu. You can reset the caption to contain any text you want; separate the hint from the caption with the tab character, `␣tcht`.

Menu with Shortcut Hint



You can also use this technique to provide shortcut key combinations for other controls. For example, you can devise a shortcut key combination for a button. First, create a menu item that is a child object of the button for which you want a shortcut. Then, define a shortcut key for the menu and define the `onClick` event handler with the same expression as the `onClick` event handler for the button. When a user types the shortcut key combination, the system executes the `onClick` event handler, and the effect is the same as if the user had clicked the button with the mouse.

Shortcut keys for menus work for menu items even when the menus are not dropped down. As long as the menu is open and enabled, the system executes its `onClick` event handler whenever the user types the shortcut key, regardless of which object has the focus. Shortcut keys for hidden menus on controls, however, work only when the focus is on the control.

Changing Menus

You will often want to change the state of a menu as the application runs. There are several techniques for modifying menus while a form is visible:

- Enable or disable individual menu items by setting their `enabled` property to 1 or 0; when disabled, the menu text appears gray. The user cannot select the item, and the system does not execute its `onClick` event handler.
- Hide or show the menu item by setting its `visible` property to 1 or 0. When you hide a menu item, the system automatically adjusts the position of the visible items.
- Change the complete structure of a menu all at once with the `def` property.

The most effective way to perform these actions is to define the `onDropDown` event-handler property of the form. The system runs this handler when the menu bar is activated, either by pressing Alt or by clicking any top-level menu item, before any of the form's menus drop down. This provides a convenient opportunity to

update the state of the menus before they are displayed. For example, you can enable or disable the Cut and Copy selections on an Edit menu depending on whether any text in an edit field is selected, or disable Paste if the clipboard is empty.

Popup Menus

A popup menu is a floating menu from which a user can make a selection. For a popup menu to be effective, it must have children, because only the children appear in response to the `Popup` method. You can use this method to provide a control-specific menu that is not part of the menu bar for the form. Ordinarily, you invoke this method in response to a right mouse click. That is, you can use the `MouseDown` event and check the value of the fourth element of `EventArgs` in your event handler. If the right mouse button was down when the user clicked the mouse, you use the `Popup` method to show the menu temporarily.

When the user selects a menu item from the popup menu, or clicks elsewhere, or presses Esc, the menu disappears. If a menu choice is highlighted when the user releases the mouse button, that action triggers the `OnClick` event handler for the highlighted menu item.

A popup menu can be a top-level object; that is, a child of the system object. In this case, the only way the user can access the menu is through an action that invokes the `Popup` method. The menu can also be a child of a form or another menu item that is a child of a form. If it is a child of the form, it will also appear on the menu bar; if it is the child of another menu item, the user could also access it by opening and cascading menus from the menu bar.

Alternatives to Menus

As an alternative or adjunct to using menus to provide the major navigational direction through your application, you can use command bars. Command bars occupy space outside the working area of a form, as do menus; that is, if you query the `Size` or `Extent` property of a form, it excludes the space taken by the menu bar and any command bars, which are all part of the border. The movement of command bars is handled by the system. The user can rearrange them if there is more than one, dock them on any side of the form, or float them on the screen outside the visible boundaries of the form.

Command bars have a special class of children, named `CommandButton`, on which you can place an image. You can have the image change when the user moves the mouse across the button to emphasize its function. There is a special style of command button that has an arrow, which you can use to drop down a menu or take some other action that is different than the action that results from clicking the button.

In addition to command buttons, you can place almost any other class of control on a command bar. You can use lists or combo boxes to allow choices or include an edit box for user input. Command bars provide a powerful facility for constructing a working area and toolbar that functions essentially independently of the main portion of your application.

If you have more controls than you can place on the screen at one time, you can use a selector and its pages to present groupings of controls to the user. The user can select which page to work with at any given time. You can also use event handlers to display and hide forms other than your main form; this presents a specific working area to the user based on a previous action, and allows your program to remove the working area when it is no longer relevant.

Moving through the Interface

In addition to moving through the interface by virtue of using the mouse, or otherwise causing an action that triggers an event handler, a user can move from control to control by using the keyboard; in particular the

Tab key (or Shift+Tab) moves the focus from control to control in a systematic pattern. Under some circumstances, the arrow keys move the focus within a group of controls.

The sequence of tabbing is determined by the order property of each control; the order properties for a set of siblings are a set of integers starting with 1 and having no gaps (sometimes designated "dense"). The value of the order property is set initially in the order in which you create controls; you can change the values. You can also modify the tabbing behavior by resetting the `tabstop` (as well as the `tabgroup` and `tabparent`) properties. You can set `tabstop` even on controls that do not get the focus, for example a frame. When you do this, it causes the focus to go to the first control that is a child of the frame that can get the focus; in this manner you can tab into a frame even when the children of the frame are not siblings of the control from which the focus is coming.

Groups of Objects

Option buttons should always appear in a group of at least two; only one option in the group can be selected. When the user selects one option, any previously selected option button is unselected. Check boxes can also be grouped, but any number can be selected. One effect of grouping controls is to make only one control reachable by tabbing through the form. The user can tab to the group, then move among the objects in the group using the cursor keys. With option buttons, the tab moves to the button that is selected. With a group of check boxes, the tab moves to the first one in the group.

The default behavior of the system is to treat a set of option buttons or check boxes as a group when they are all children of the same frame or form, and when they are created with successive calls to `□wi`. This order of creation is reflected in the order property of each object; the members of a group are those objects that have successive values of order.

If you want to create two groups of option buttons within a single parent object, you can create an object of a different class, such as a label, between the two groups. The intervening object can be invisible if it is not needed on the form. For example, the following statements create two groups of option buttons within a single frame, using an invisible label to separate them:

```
x←'fmMain.fPrt' □wi 'New' 'Frame' ('where' 1 2 9 14)
  ⌞ ('caption' 'Print Options')
(x, '.opPortrait') □wi 'New' 'Option' ('where' 1 1 1.5 12)
  ⌞ ('caption' 'Portrait')
(x, '.opLandscape') □wi 'New' 'Option' ('where' 2.5 1 1.5 12)
  ⌞ ('caption' 'Landscape')
(x, '.lDummy') □wi 'New' 'Label' ('visible' 0)
(x, '.opA4') □wi 'New' 'Option' ('where' 5 1 1.5 12) ('caption' 'A4 size')
(x, '.opLtr') □wi 'New' 'Option' ('where' 6.5 1 1.5 12)
  ⌞ ('caption' 'Letter size')
```

Two Groups of Options in a Frame



You can also manipulate object groupings by setting their order properties. You can insert an object between two other objects using a fractional value (for example, 4.5 to insert between 4 and 5). The system then automatically renumbers the objects to maintain a dense set of successive integers. See the descriptions of siblings, order, tabstop, tabparent, and tabgroup properties for more information.

Accommodating Keyboard and Mouse Input

Simple user interfaces do not require you to monitor directly every action that the user performs with the mouse or keyboard, because the suite of built-in events expresses the normal function of each class. In special situations, however, you may want to write code to track keyboard and mouse actions in detail. Monitoring keyboard activity allows you to filter and validate input as the user types each keystroke. It also allows you to receive keyboard input when the focus is not in an Edit control. Mouse activity is often monitored in graphics applications to let the user enter positional input with the mouse.

Keyboard Handling

When the user presses a key on the keyboard, a number of actions can occur. The first thing you might think of is that pressing a letter key can cause a letter to appear in an edit box. However, if the Alt key was pressed immediately prior, pressing a letter key might cause a menu to drop down or select a menu item. Arrow keys might make the cursor move within a control or from one control to another. Pressing the Enter key might click a button that has the focus, or it could trigger a default button for the form if another control has the focus; if the correct style value is set, pressing the Enter key could also insert a carriage return in an edit box. Function keys might invoke an action defined by the operating system, by the user, or by your application.

The keyboard handling within APL+Win can define actions that supplement these "normal" actions or it can suppress, replace, or modify them. These properties are used to process keyboard input:

- The `onKeyDown` event handler defines the action taken when a key is pressed.
- The `onKeyPress` event handler defines the action taken when a character or symbol is recognized as a result of a key being pressed.
- The `keys` property, if not empty, specifies the set of characters that trigger the `onKeyPress` handler.
- The `onKeyUp` event handler defines the action taken when a key is released.

These event handlers usually apply only to the object that has the input focus. The `onKeyDown` and `onKeyUp` event handlers provide the means to define an action based on the press or release of a specific key

on the keyboard. The `onKeyPress` event handler lets you detect whenever a user generates a specific ANSI character from the keyboard, either the shifted or unshifted version of a character the key represents, but also including extended ANSI characters generated with international keyboards or entered with the numeric keypad while pressing the Alt key; the `keys` property filters the `KeyPress` events that a control receives.

Taking Action Based on the Physical Key

Very detailed low-level keyboard processing is possible with the `KeyDown` and `KeyUp` events. This level of detailed control is rarely necessary, but can be used when you need to follow the press and release of each physical key. The `onKeyUp` property defines a handler for the release of any physical key on the Windows keyboard device. The `onKeyDown` property defines a handler for the press of a physical key on the Windows keyboard device; however, Windows reserves the `KeyDown` event for some keys. The specific keys for which you cannot use the `onKeyDown` event handler depend on the object and the circumstances. During the `onKeyDown` and `onKeyUp` callbacks, the value of `¶warg` is a 7-element vector with detailed information about the key. The table below shows the elements of `¶warg` for these event handlers.

Values of `¶warg` for `onKeyDown` and `onKeyUp`

Element	Description
<code>¶warg[1]</code>	Keyboard scan code (hardware dependent).
<code>¶warg[2]</code>	Key repeat count (<code>onKeyDown</code> only).
<code>¶warg[3]</code>	Shift state; sum of 1 for Shift, 2 for Ctrl, and 4 for Alt.
<code>¶warg[4]</code>	Caps Lock state; 1=Caps Lock on.
<code>¶warg[5]</code>	If 1, indicates that the key is an extended key such as an arrow key on a separate cursor keypad.
<code>¶warg[6]</code>	If 1, indicates that the key is being repeated.
<code>¶warg[7]</code>	The virtual keycode (hardware independent) for the key.

You can suppress the default effect of pressing a specific key by setting `¶wres` to `-1` (silent) or `-2` (beep) in your `onKeyDown` event handler. You can also set `¶wres` for a `KeyUp` event, but usually any default action has already taken place.

Taking Action Based on the Character

The `KeyPress` event occurs between the `KeyDown` and `KeyUp` events. During the `onKeyPress` callback, the value of `¶warg` is the ANSI character code of the character that the keypress generates. Note that this may or may not match the seventh element of `¶warg` for `onKeyDown`, since the same physical key can generate different characters depending on whether the Shift key is pressed.

Your handler can define an action that occurs in addition to the default system action, or you can control how the system handles each keypress by examining `¶warg` and setting `¶wres` appropriately. If you do not set `¶wres`, it has its default value of 0, which sends the keystroke through to the object. Setting `¶wres` to `-1` (silent) or `-2` (beep) rejects the keystroke, so that the current object never sees the character. In this case, any actions your `onKeyPress` handler defines beyond setting `¶warg` are the only effect of the keypress. Setting `¶wres` to a value greater than zero specifies an ANSI character, which is sent to the current object in place of the character the user typed. This provides the means, for example, to translate a character from lowercase to uppercase.

If `keys` has its default value `0`, all keypresses trigger the `onKeyPress` event handler. Setting `keys` to a list of ANSI character codes causes only those characters to generate `KeyPress` events. The system then treats other characters as though no `onKeyPress` event handler is set.

Mouse Handling

Windows defines events that give you the ability to monitor and react to mouse activity in your application. The object whose event handler the system executes is determined not by which object has the input focus, but by which object the mouse is positioned over when the event occurs. The mouse-related events include:

onMouseXx event handlers

Event-handler	Description
onMouseDouble	The user doubleclicks a mouse button.
onMouseDown	The user presses a mouse button.
onMouseDownDrag	The user moves the mouse with a mouse button pressed.
onMouseEnter	The user moves the mouse over a control.
onMouseLeave	The user moves the mouse off of (away from) a control.
onMouseMove	The user moves the mouse.
onMouseUp	The user releases a mouse button.
onMouse	A mouse action occurs over a Toolbox; this affects the handling of subsequent mouse actions.

During the callback for most of these events, `␣warg` is a five-element vector. The table below shows the elements of `␣warg` for these event handlers. The exceptions are `onMouseLeave`, for which `␣warg` is empty and `onMouse`, for which `␣warg` has values unique to a Toolbox.

Values of `␣warg` for the onMouseXx event handlers

Element	Description
<code>␣warg[1 2]</code>	The row and column coordinates for the mouse pointer.
<code>␣warg[3]</code>	For <code>MouseDown</code> , <code>MouseDownDrag</code> and <code>MouseUp</code> events, indicates which mouse buttons the user pressed; the sum of 1 (left button), 2 (right button), and 4 (middle button). Always 0 for <code>MouseEnter</code> , <code>MouseMove</code> , and <code>MouseDownDrag</code> .
<code>␣warg[4]</code>	The (down) state of all mouse buttons when the event occurred; the sum of 1 (left button), 2 (right button), and 4 (middle button).
<code>␣warg[5]</code>	The keyboard shift state when the event occurred; the sum of 1 (Shift key down), 2 (Ctrl key), and 4 (Alt key).

By setting `␣wres`, you can control whether the mouse activity is also passed through to the object for default Windows processing. The default result of 0 passes the mouse event through to the control for its own default processing. Setting `␣wres` to `-1` prevents the object from seeing the mouse action, so the only effect, if any, is that produced by the handler. Setting `␣wres` to `-2` blocks the default action and also causes a beep.

If you have lingering doubts about the whether an interpreted APL function is fast enough to keep up with Windows events, try the following simple test:

```
'fmChase' ␣wi 'New' 'Form'
'fmChase.Cat' ␣wi 'New' 'Button'
'fmChase.Cat' ␣wi 'where' 0 0 1.5 4
'fmChase' ␣wi 'onMouseMove' " 'fmChase.Cat' ␣wi 'where' (2↑␣warg) "
```

Now, move the mouse into the `fmChase` window. The Cat button follows the mouse in response to the `MouseMove` events. Note that it is not necessary for the form to have the input focus for it to receive these events.

Using the Wait Method on Forms

The Wait method activates a form for user input and gives it the input focus. In effect, the form is waiting for input events; it remains in the Wait method until the form is closed. A runtime application typically creates and waits on its main form, then terminates execution when the user closes this form. An application that runs in the development system, for example one that you build for yourself, does not require an explicit wait. There is an implicit wait loop running on the session manager.

Waiting on a form makes it application modal; in other words, any other forms in your application that were visible at the time of the Wait cannot receive input until the Wait ends. However, any forms in your application that are opened after the Wait begins are able to receive events. This leads to a subtlety when your application needs to show multiple forms at once and process events occurring in any of them. If you open two forms and then invoke Wait on one of them, only that form can receive user input.

In the example below, only `form2` can receive events:

```
'form1' □wi 'Show'  
'form2' □wi 'Wait'
```

To enable both forms, open the two forms, but use Wait only on the first form you open. One technique is to use the `onWait` event handler for one form to open the other. In the example below, both forms can receive events. Note that using either the Wait or Show method automatically invokes the Open method.

```
'form2' □wi 'onWait' " 'form1' □wi 'Show' "  
'form2' □wi 'Wait'
```

In some applications, only the main form runs the Wait method. The application displays other forms when they are needed using the Show method, which allows the user to type input in any of the application's windows. Nested calls to Wait are appropriate only when it is necessary to make a form application modal and block input to other forms that are already visible.

The style property for a Form provides some additional options. You can make a form independent of any other form that is waiting, or you can provide that a form is not disabled for a wait on a secondary form (one that otherwise is application modal). You can also Hide a form before waiting, which allows you to have a Wait on a hidden form. (Normally, the Wait method causes a form to open, and hiding an open form breaks the Wait.) Finally, you can signal the Wait event before opening a form, which allows you to process the form in the `onWait` event handler (for example, to fill in some fields that are dependent on the current state of the application) before the form becomes visible to the user.

Because the APL session manager automatically runs a wait loop, forms that you create during the process of application development are active and able to process events. This is generally useful during the development process, since you can test a form interactively while in immediate execution mode. In a finished application that is packaged with the runtime system, you must run the Wait method on at least one form in order for the application to execute callbacks. The runtime system has no implicit wait loop.

Note: If you want to experiment with the Wait method in immediate execution mode, be sure your form has a system menu or a Cancel button (style 2) before you invoke Wait.

Building an APL+Win Windows Interface Program

The flow of execution in a Windows application written with APL+Win generally follows these steps:

1. Create the form and controls.

This step does not always happen at run time. If your forms have the same layout every time the application runs, you can generate the form once and save the definition. The advantage of this approach is that the form starts faster. In other situations, you might want to generate the form dynamically after the application is running. Dynamic forms are necessary when you build applications like a data query system that lets the user decide which fields of a database to view.

2. Display the form and controls.

Use `□wi` to invoke the Show method on a form with a statement like `'fmMain' □wi 'Show'`. This causes APL to display the form and all of its control objects.

3. Enter a Wait loop.

Invoke the Wait method on your main form (`'fmMain' □wi 'Wait'`). At this point, the application is waiting for the user to do something with the form. No further application code runs until the user begins to use the keyboard or mouse.

4. Execute handlers as the user manipulates the controls.

As the user interacts with the form, Windows sends events to APL. If you have defined handlers for those specific events, the system executes the associated APL expressions. After each handler completes, the form continues waiting until it is closed. The system runs the handlers without exiting from `□wi`. When events occur, `□wi` executes the handler expression directly. In effect, your APL code calls the user interface, which then waits for input and calls APL back in response to events. Thus, the term *callback* refers to the process of handling an event with `□wi`.

5. Handle the Close event to finish processing.

A form can be closed by the user or by the action of application code. In either case, the system generates a Close event. The handler for this event is responsible for taking whatever action is required before the form disappears. For example, if the user has filled in an input form, the `onClose` event handler has the task of extracting the user's input and processing it.

How Callbacks Work

Callbacks are the mechanism by which the system executes event handlers. In a packaged application, callbacks can occur only during the execution of `□wi`, `□wgive`, or `□wcall` (unless you explicitly enable nested callbacks by calling `□wgive` with a non-zero argument). Your application code calls `□wi`; `□wi` calls your application back to run an event handler. The event handler runs to completion and control returns to `□wi`.

Callbacks can also occur while you are working in the APL session manager. When APL+Win is in immediate execution mode, you can manipulate a form and cause its callbacks to execute. This is possible because the APL session manager runs a wait loop all the time, making it easy for you to build and test user interfaces interactively. You can enable or disable this behavior by toggling the Enable Callbacks menu item on the session manager's Debug menu.

In principle, any call to `□wi`, `□wgive`, or `□wcall` is capable of producing a callback. Even if you are using `□wi` only to reference an object's property, a callback can occur if there is an event pending that needs to be processed. By default, however, the system does not check the Windows message queue during a callback, so normally each callback runs to completion before the system handles the next event. Since an application typically becomes active by running the Wait method (a call to `□wi`) on its main form, callbacks usually occur only during this Wait, or during subsequent Waits executed for other forms.

Yielding the CPU and Enabling Nested Callbacks

The default system behavior of having each event handler run to completion and return control to `□wi` before the next event is handled ensures that events are processed in a well-defined sequence. During the time that a callback is running, all other Windows events for the application are pending. For this reason, it is useful to arrange applications so that most callbacks can be completed quickly.

Many applications have periods of extended "crunching" when processing continues for a period of time without user input. Although the user could click over to another application, his APL form is essentially frozen. Even if he attempts to click back to APL, the form may not repaint while the long-running event handler is in progress.

The system function `□wgive` checks the Windows message queue even while a callback is running. You can use it with an argument of 0 to yield control of the CPU to allow other tasks to run; for example, you might want to have a Cancel button to abort the long-running calculation. Placing a `□wgive 0` at a strategic point in a loop could enable you to shut down the calculation gracefully and perhaps save the intermediate result.

You can also use `□wgive` with a positive integer argument to enable nested callbacks in your application. This approach causes the system to yield control periodically without reaching a particular point in your code. The setting is retained and effective until you reset `□wgive`. Allowing nested callbacks implies that the system can process or begin to process additional callbacks to `□wi` or `□wcall` before the first callback finishes. This is a subtle technique that you should use only with care. Callbacks can occur at any point when messages are received, and the interactions of multiple callbacks can cause unexpected consequences. See the description of `□wgive` in the System Functions, Variables, and Constants Chapter in the *System Functions Manual* for more information.

Reading Callouts and Callbacks on the Stack

The APL execution stack shows all callbacks and callouts, which are calls to the system functions capable of generating callbacks. Callouts and callbacks are given their own rows in `□si` and the debugger stack. Callouts are denoted in `□si` using the notation `<[object;action]`

For a callout using `□wi`, *object* is the fully qualified name of a Windows object, and *action* is the name of the method or property that is the subject of the call to `□wi`. For example, `<[fmMain;Wait]`

Callbacks are denoted in `□si` using the notation `>[object;event]` where *object* is the name of the object for which the event has occurred, and *event* is the name of the event. For example, `>[fmMain.bnOK;Click]`

For `□wgive` callouts, *object* and *action* are omitted: `<[ ; ]`

For `□wcall`, a callback can occur only when an event filter has been created (see the Calling Dynamic Link Libraries chapter in the *System Functions Manual*). A callout or callback from `□wcall` omits the *object*, and the *event* always displays as the question mark (?). For example, `>[ ; ?]`

The flow of control in an event-driven application can be unpredictable, since user-generated events, timer events, and DDE events can all occur in any sequence. The information in the stack is sufficient for you to be able to tell exactly what sequence of callouts and callbacks are in the process of being executed. For example,

```
WGetOpenFileName[25]
>[fmMain.mFileOpen;Click]
<[fmMain;Wait]
Startup[12]
```

The example above tells us that an application started by running the function `Startup`, which used `Wait` to invoke a `Wait` on the form `fmMain`. The user selected `Open` from the form's `File` menu (which caused a `Click` event), and this invoked the APL cover function `WGetOpenFileName`.

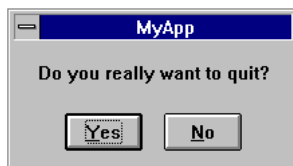
Common Dialogs and Message Boxes

Windows includes a library of standard dialog boxes called common dialogs, which are useful in a number of applications. It is wise to use the common dialogs wherever possible in your own APL applications because many Windows users are already familiar with them. APL+Win provides each of the following common dialog boxes packaged as an APL function. These functions can be found in the `WINDOWS` workspace in the `APLWIN73\TOOLS` directory:

- `WChooseColor` displays a palette of color choices and allows the user to select a color.
- `WChooseFont` displays a list of the fonts installed on the system and lets the user select the font, size, and style.
- `WChoosePrinterFont` displays a list of the fonts installed on the system and the printer and lets the user select the font, size, and style.
- `WGetOpenFileName` prompts for the name of the file the user wants to open and allows the user to browse different directories.
- `WGetSaveFileName` prompts for the name of an output file the user wants to create or write to.
- `WGetOpenLongFileName` prompts for the name of the file the user wants to open and allows the user to browse different directories. This version supports long filenames.
- `WGetSaveLongFileName` prompts for the name of an output file the user wants to create or write to. This version supports long filenames.

Another widely used style of window is the message box, an application-modal dialog box that appears in the center of the screen to notify the user of urgent information that requires an immediate acknowledgment. The utility function `Wmsg` in the `WINDOWS` workspace generates a message box. For example, the following statement pops up a message box that asks the user a yes/no question:

```
ans←Wmsg 'Do you really want to quit?' 'MyApp' 4
```



Using Printer Objects

You can generate printer output within your own applications using the `Printer` class built into APL. Each object of the class you create is an abstraction for printing one document on the Windows system printer. You can have multiple `Printer` objects at once, each controlling the printing of a single document (print job). Closing a `Printer` object causes the document to print or queues it for the Print Manager. The following series of statements opens a printer object, sets its font, and prints two variables on the default printer.

```
prt←'prDefault' ⍵wi 'New' 'Printer' ('caption' 'Print Job')
prt ⍵wi 'Set'('font' 'Arial' 2)('position' 4 8)
prt ⍵wi 'Print' Text1
prt ⍵wi 'Eject'
prt ⍵wi 'Print' Text2
prt ⍵wi 'Close'
```

Note: The caption property may be required when printing across a network.

The first statement creates a printer object; the second defines the font style and the position on the page where the printer will print the text output. The third statement uses the `Print` method, which is useful for directing an APL character vector to the printer. It begins printing at the current position, and handles newline (`⍵t⍒nl`) characters automatically, moving to the start of a new line on the printer for each line in the text and ejecting the page when you invoke the `Eject` method or when the page is full. You can call the `Print` method many times to append text to a print job, which provides a simple way to print without handling every detail of text positioning. The `Print` method updates the position and page properties automatically as the text is printed. The last statement uses the `Close` method to complete the print job, at which point APL releases it to the Windows print queue.

The table below shows some of the properties you can set when a `Printer` object is open. See the Windows Interface: Properties chapter in this manual for more information.

Printer Properties

Property	Description
<code>bin</code>	Defines the number of the current bin, or paper tray the printer will use.
<code>bins</code>	Lists the possible bins, or paper trays, you can assign for the printer.
<code>caption</code>	Defines the name of the Print Manager task. See the note below.
<code>collate</code>	Specifies whether to print multiple copies as collated documents.
<code>copies</code>	Specifies the number of copies for the current print job.
<code>default</code>	Boolean flag that specifies whether the printer is the current default printer.
<code>font</code>	Defines the current font name, size and style.
<code>hdc</code>	Device context handle for the printer (used only when you need to use low-level Windows calls with the printer).
<code>orientation</code>	Character vector that shows portrait or landscape for the current page.
<code>page</code>	The page number for the current page; updated only by the <code>Print</code> method. The system executes the <code>onPage</code> event handler whenever a new page is produced.
<code>paper</code>	Defines the number of the current paper size the printer will use.
<code>papers</code>	Lists the possible paper sizes by name and number you can assign.
<code>position</code>	Defines the position on the page where the next text output generated by the <code>Print</code> method appears. It is a two-element vector that contains the current row and column in the object's current scaling units.
<code>printrname</code>	Character vector that specifies the printer you want to use.
<code>scale</code>	Determines the scaling units that apply to the <code>Draw</code> method and the <code>position</code> property.

Note: You can use the `caption` property to define the title of the print job as it will appear in the Print Manager's print queue. This title may also print on the title page if the document is printed on a network printer. Note that changing the caption on the object while it is open has the effect of ending the current print document and starting a new one.

The table below shows the methods of Printer objects that are specific to controlling the printer:

Printer Methods

Method	Description
Setup	Displays the standard Printer Setup dialog box to allow the user to select the type of printer, page orientation, and print quality.
PageSetup	Displays the standard Page Setup dialog box to allow the user to configure the page description for a printer.
PrintDlg	Displays the standard Print Dialog dialog box to allow the user to select a printer, whether to print to a file, print range, number of copies, and whether to collate. The user can also invoke another dialog box to access other printer properties.
DevMode	Allows you to save or modify the <code>DEVMODE</code> structure used in Windows to specify settings for a printer.
DevNames	Allows you to save or modify the <code>DEVNAMES</code> structure used in Windows that contains the printer name, printer driver name, and port name.
Print <i>text</i>	Prints a character vector on the printer with the current font. New-line characters (<code>\n</code>) and form-feed characters (<code>\f</code>) cause line breaks and page breaks.
Draw <i>ops</i>	Draws graphics on the printer with a series of graphics operations.
Abort	Cancels the current print job.
Eject	Starts a new page without closing the current print job.
Close	Closes the print job and queues it for the Print Manager. If the Print Manager is not running, the system starts it.

A Printer object can produce the following events:

- **Page**
Signaled when the printer starts a new page but before it begins printing; `Page` is the new page number. By setting the `onPage` event-handler property, you can specify actions that are taken on each new page. This allows you to print the page number and any other text or graphics that you want printed on each page. It also lets you show the progress when you are printing a multiple-page document by showing how many pages of output have been produced.
- **Error**
An error occurred during printing; `Error` contains the error number. You can assign a non-zero value to `Ignore` to specify that the system should ignore the error. This allows you to use the `Draw` method to a context defined by low-level calls by setting the printer `hdc` to the (non-printer) handle of the context.

Printing in Windows is page-oriented and based on the idea that you create the image of a new page in Windows memory. You write text and draw graphics anywhere on this virtual page and then transfer the image of the page to the printer all at once. This model applies to all kinds of printers. A dot-matrix printer that prints continuous fan-fold paper is treated like a laser printer that prints on cut sheets of paper. Windows ejects the last page of a print job, whether it is full or not, to clear the printer for the next job.

Understanding printing under Windows is easier if you think in terms of pages. Page orientation means, for example, that if you write output to the printer that extends past the page margins in any direction, it is clipped to fit on the page. The `Print` method of the Printer object class provides a means for overcoming this behavior to a certain degree. It allows you to print character vectors containing embedded newline characters, moving to a new page as each is filled from top to bottom.

The Printers method, also on the system object, returns a list of the printer drivers installed on the local machine; returns or sets the current default; and searches a network for printers that are accessible.

Printing APL Characters and Objects

You can print objects that contain APL characters, such as your own functions, and the contents of APL variables using the printer object. The example below shows the technique, using the printer object created above. Note that you must make each output argument a character vector.

```
'prDefault' ⍵wi 'font' 'APLPLUS' ⍉ ⍉ 'symbol'
NumVar ← 10 12 ρ 1120
'prDefault' ⍵wi 'Print' (,('I4' ⍵fmt NumVar),⍵tcnl)

Textvar ← 309 ρ 'abcdefghijklmnopqrstuvwxyz'
y1 ← (6 50ρTextvar),⍵tcnl
y2←(-9↑Textvar),⍵tcnl
'prDefault' ⍵wi 'Print' ((,y1),y2)

'prDefault' ⍵wi 'Print' (⍵vr 'foo')
'prDefault' ⍵wi 'Close'
```

Extending the Widget Customizer

In the previous chapter, as a working example, steps were shown to build a simple form with some basic controls and event handler functions that simulated a real application. The sections below build on the previous chapter's example to employ more basic techniques and make the example somewhat more realistic, although still simple. However, you do not have to have actually executed the example in the previous chapter; the necessary functions are contained in the workspace: `Chapter6.w3`

To make it easier to execute the examples, all the steps are included in a log file that is included with APL+Win. In your APL session, click on the Session Log choice on the Options menu. **Note:** Make sure the Append to File option for the Logging Method is selected. Click the Browse button, go to the `\APLWINxx\Docs` directory, and select `Ap1_doc2.log`. Click the Begin or Restart Logging option and the OK button.

The system reads the file into your session; you can scroll up to see the entire log file. **Note:** Before you continue, return to the APL Session Log screen, select Turn Logging Off, and click OK. This will prevent you from extending the log file with your experimentations. If you wish to save your own statements in a log, assign a different file name and click Begin or Restart Logging and OK.

If you don't wish to use the log file, the steps are reproduced in this chapter and all the necessary lines of code are listed, so that you can modify the form and its event handler function. An additional event handler function is also listed.

Revising the Main Form

Unless you are extraordinarily prescient, you have to modify your original design. With the Widget Customizer, there is a need for more information on the form. In order to put more effective working area in the same space, we will replace the frame that holds the color choices with a selector that has multiple pages. However, we still want to offer the color choices, so we save the definition of the frame for later use. Then, we build the selector and place the frame on its first page. Note that the code listings in this section suppress the output that would be returned by their execution. The full record of these steps does appear in the log file mentioned above.

Building a Selector – Page One

```
)copy \aplwin7\Docs\Chapter6 fmMain_Make ShowColor Ordering
fmMain_Make
framedef ←'fmMain.fColor' □wi 'def'
'fmMain.fColor' □wi 'Close'
'fmMain.selChoices' □wi 'New' 'Selector' ('where' 0.5 14 11 18)
'fmMain.selChoices.pgColor' □wi 'New' 'Page' ('caption' 'Color')
'fmMain.selChoices.pgColor.fColor' □wi 'New' framedef
'fmMain.selChoices' □wi 'value' 1
'fmMain.selChoices.pgColor.fColor' □wi 'where' 0 0
```

Note that the way the ShowColor event handler function was written, using relative naming, allows it to work without modification even when the frame has a different parent. You should be able to click on a color choice and see the color in the bar beside the list.

Building a Selector – Page Two

Next we build the second page of the selector. On this page, we place a Picture control, on which we draw some descriptive text and then show representations of the pattern for various sizes of material and tightness of weave. The event handlers respond to single and double mouse clicks on the picture. We also use the Push and Pop methods on the picture to restore the initial selection screen. Note that you must create and save the DrawWeave function for this page to be effective.

```
□wself←'fmMain.selChoices'
□wi '.pgFabric.New' 'Page' ('caption' 'Fabric') ('value' 2)
□wself←□wi '.pgFabric.picWeave.New' 'Picture' ('where' 0.5 0.5 8.1 16.2)
□wi 'scale' 5
□wi 'Draw' 'Clear'
□wi 'Draw' ('Text' ('      Coarse Thick Light Fine') 0 0)
□wi 'Draw' ('Font' 'Arial' 10 1 'ansi' 270) ('Text'
↳ ('Loose Firm Snug Tight') 16 12)
□wi 'Draw' ('Line' (3 2 ρ 16 128 16 16 128 16))
□wi 'Push'
picEdge← □wi 'image'
□wi 'onOpen' " □wi 'image' picEdge ◇ □wi 'Push' "
□wi 'tooltip' 'Double-Click to see weave pattern'
□wi 'onMouseDouble' 'DrawWeave'
□wi 'onMouseDown' " □wi 'Pop' "
```

The DrawWeave Function

```
▽ DrawWeave;row;col;fiber;weave;x
[1] (row col)←□warg[1 2]
[2] :if row<17
[3] :orif col<17
[4] :return
[5] :end
[6] fiber←4-1col÷28
[7] weave←6-1row÷28
[8] x ←18
[9] :while x < 128
[10] □wi 'Draw' ('Pen' 1 0 fiber) ('Line' (2 2 ρ x 16 x 128))
[11] □wi 'Draw' ('Pen' 1 0 fiber) ('Line' (2 2 ρ 16 x 128 x))
[12] x←x+fiber+weave
[13] :end
▽
```

Building a Selector – Page Three

We build a third page to capture the customer information. This page consists of three edit fields, with accompanying labels, and a check box with an associated label (in lieu of using the `caption` property). The check box uses a handler to disable the credit card number edit field when it is checked and to re-enable the edit when the check box is unchecked. The credit card edit field itself uses the `onKeyPress` event handler to limit entry to numerals and the hyphen (minus sign).

```

□wself←'fmMain.selChoices.pgOrder' □wi 'New' 'Page' ('caption' 'Order')
'..' □wi 'value' 3
□wi '.lName.New' 'Label' ('caption' 'Name') ('where' 0 0)
□wi '.edName.New' 'Edit' ('where' 1 0 1 16)
□wi '.lAddr.New' 'Label' ('caption' 'Address') ('where' 2.3 0)
□wi '.edAddr.New' 'Edit' ('where' 3.3 0 2 16) ('style' 4 8192)
□wi '.lCredcd.New' 'Label' ('caption' 'Credit Card Number')
□wi '.lCredcd.where' 5.6 0 1 12
□wi '.edCredcd.New' 'Edit' ('where' 6.6 0 1 12)
□wi '.lBillme.New' 'Label' ('caption' 'Bill me') ('where' 5.6 13)
□wi '.ckBillme.New' 'Check' ('where' 6.6 13) ('caption' '')
□wi '.ckBillme.onClick' " '..' □wi '.edCredcd.enabled' (1≠□wi 'value') "
□wi '.edCredcd.onKeyPress' '□wres←-(0=□warg∈((47+110),45))'
```

Save the definition of the selector, since it isn't in the `Make` function.

```
ReplaceFrame←'fmMain.selChoices' □wi 'def'
```

You can edit `fmMain_Make` to close the frame and replace it.

```

'fmMain.fColor' □wi 'Delete'
'fmMain.selChoices' □wi 'New' ReplaceFrame
```

Modifying the Ordering Function

We will modify the `Ordering` function in three ways. First, we will add a popup menu to the `ERROR` form when the user fails to select a style, so that understanding the nature of the error, and correcting it, is easier. Then we will replace the statement "Order info to be added" with the text information from the third page of the selector. Finally, we will correct the lines that clear the color setting, since the color choice moved from the frame to the first page of the selector.

Insert these lines into `Ordering` after the `Draw` command that says you must pick a style and before the `Wait` statement on the form named `ERROR`.

```

sink←'mFix' □wi 'Create' 'Menu'
□wself←'mFix.mOne' □wi 'Create' 'Menu' ('caption' 'Aluminum')
□wi 'onClick' " 'fmMain.opLight' □wi 'value' 1 "
□wself←'mFix.mTwo' □wi 'Create' 'Menu' ('caption' 'Steel')
□wi 'onClick' " 'fmMain.opStrong' □wi 'value' 1 "
□wself←'mFix.mThree' □wi 'Create' 'Menu' ('caption' 'Titanium')
□wi 'onClick' " 'fmMain.opBoth' □wi 'value' 1 "
'ERROR' □wi 'Popup' 'mFix' 0 3 6.5
```

Make the form longer to accommodate more information. Insert the following line after you create the order acknowledgement form or edit the `where` property setting.

```
'fmOrderForm' □wi 'size' 13 30
```

Comment out the line that says Order info to be added.

```
A 'fmOrderForm' □wi 'Draw' ('Text' 'Order Info to be added' 5 5)
```

Insert the following lines to make the example more closely resemble a real ordering application.

```
    □wself ← 'fmOrderForm'
:if 0=ρ('fmMain.selChoices.pgOrder.edName' □wi 'text')
:orif 0=ρ('fmMain.selChoices.pgOrder.edAddr' □wi 'text')
    □wi 'Draw' ('Text' ('But we need your name and address') 7 1)
:else
    □wi 'Draw' ('Text' ('fmMain.selChoices.pgOrder.edName' □wi 'text') 7 1)
    □wi 'Draw' ('Text' ('fmMain.selChoices.pgOrder.edAddr' □wi 'text') 8 1)
    :if 0= ('fmMain.selChoices.pgOrder.ckBillme' □wi 'value')
        □wi 'Draw' ('Text' ('fmMain.selChoices.pgOrder.edCredcd'
↵        □wi 'text') 11 1)
    :else
        □wi 'Draw' ('Text' ('We will bill you at the above address') 11 1)
    :end
:end
```

Fix the last two lines of the function, since the frame went onto the selector. Comment out the old lines and insert the new ones or edit the object name.

```
A old line A 'fmMain.fColor.lsHue' □wi 'value' 0
A old line A 'fmMain.fColor.picDisplay' □wi 'Draw' 'Clear'
'fmMain.selChoices.pgColor.fColor.lsHue' □wi 'value' 0
'fmMain.selChoices.pgColor.fColor.picDisplay' □wi 'Draw' 'Clear'
```

While the third page of the selector is visible, experiment with the Tab key and the arrow keys. If none of the options is selected, the focus starts on the check box; if you select one of the options, the focus starts on the one that is selected and the arrow keys move through the set of options, selecting each in turn. When the focus moves to the tab of the selector, the arrow keys move from page to page. However, once you tab into the controls on the selector, you must click on a tab to select a new page.

After you enter text information in the name and address edit fields, note the difference in the up and down arrow keys between the single line name field and the multi-line address field (assuming you pressed the Enter key to generate a second address line).

Also note that checking the Bill me check box disables the credit card edit field, and prevents any further data entry, but does not erase the values already entered. You can still query the text property of the control even though it is not enabled.



Chapter 7: Programming the Windows Interface: Intermediate

This chapter explores several techniques that may be useful in your applications. There are many fundamental techniques that involve the use of a given class of object. You can read the documentation for each class in the Windows Interface: Classes chapter. The techniques described in this chapter involve the interaction of multiple classes or their properties, events, and methods working together.

Naming with Links

For any object, you can assign an arbitrary name (a link) that applies to another object (the target) anywhere in the same application, and then use the link name as though the target is a direct child of the object for which the link is defined. The name is a "link" from the pseudo-parent to the target, or you can think of it as an alias for the target. You can use the name of a link in all the ways you use an ordinary object name to identify the object that is the target of the link.

Link names are stored in the `links` property, which applies to every class in the system, including the system object. This property is a two-column matrix with the first column being the link name and the second column being the actual name of the target or another link to that name. The link name (first column) must be a single-part name, as for any child. The target name can be an absolute name or a relative name; if it is a relative name, the base name is the name of the object that defines the link.

If you are assigning only one link name, you can use a two-element vector, which the system treats as a one-row matrix. You can set the `links` property directly, or you can use the `SetLinks` method, on any object, with a two-column matrix (or a two-element vector for a single link) as the argument. You can delete a link, using `SetLinks`, by specifying the link name with an empty target (second column or element), or by resetting the `links` property without the row of the link you want to delete.

There are many possible uses for this facility:

- A link may serve as a short, direct reference to a particularly significant object. It may be convenient, for example, to use a link as a short name for a deeply-nested object.
- A link can name a semantically essential control in a way that is unaffected by changes to an application's visual and navigational structure. For example, you might define a "logical name" as a link on a form to a target object, and use the logical name in event handlers. Thus, you can avoid changing the event handlers if the target object is moved from its direct placement on the form to being within a frame on a page of a selector.
- A link may serve as a dynamically changeable identification of the current object of some special interest. For example, you could set a link to the page that was most recently hidden on a Selector, so that a GoBack button would return the user to the location of his previous action.
- A link may serve as an abstract identifier for objects to be implemented by one programmer and used by another.
- A link may be used to provide input or output parameters to reusable clusters of controls.

A simple use of this facility would be to make a short name for an object deep down the naming hierarchy. You could assign an arbitrary name as a link from the system object to a deeply nested menu object and refer to the menu item as though it were a top-level item anywhere in your code.

```
'#' □wi 'links' 'Silver' deeply-nested-name
```

```
'Silver' □wi 'Click'
```

If you had many forms, each of which had an Apply button, you could use the following code to simplify your event handlers. Note the colon in the second line.

```
□wself ← button-birth-name □wi 'New' 'Button'  
□wi ':SetLinks' 'ApplyNow' □wself
```

Once you set this up, regardless of the name you assign to the button, you don't have to change the code that sets up the link, nor do you have to change any event handler that refers to the button as

'form-name.ApplyNow'. For example, an event handler might disable the button by:

```
'some_obj' □wi 'onOpen' " 'form-name.ApplyNow' □wi 'enabled' 0 "
```

And your code could set an event handler for the button using the string:

```
'form-name.ApplyNow' □wi 'onClick' 'Go'
```

You can use links to coordinate between forms. Suppose you were creating an application where you found or created a value whose result is displayed in an edit box on one form; you then call a second form where parameters are set to process the result. With four key lines of code defined, you can develop each form with its naming independent of the other. Line [a] establishes two top-level links, aliasing the second form with the link name "Process" and the edit control with link name "value" assigned to a name on the first form that will itself be a link to the edit control (*'fm1.box'*). Line [b] establishes the link from *box* to the actual edit control.

```
[a] '#' □wi 'links' (2 2 ρ'Process' 'fm2' 'value' 'fm1.box')  
[b] 'fm1' □wi 'SetLinks' 'box' actual-name-of-edit-control
```

Line [c] occurs in an event handler on the first form. This statement invokes the second form using only the link name assigned arbitrarily at the highest level.

```
[c] 'Process' □wi 'Show'
```

Line [d] sets the second form's *onShow* event handler; the handler reads the result from a control on the first form without any direct use of the actual naming conventions used in building the form. The only connection is the link name.

```
[d] 'fm2' □wi 'onShow' "result← 'value' □wi 'text' "
```

Parsing Rules for Using Links

When you create a link name by using the *SetLinks* method or setting the *links* property of an object directly, the system checks the validity of the link name (first column) but it does not check the validity of the target, except that the target must be a character vector or scalar. This behavior has the disadvantage that you don't discover an error until you try to use the link; it has the advantage that you can create the link name at any time and create its target, the real object, later. You can even use the link name with the *New* or *Create* method to create the object.

Note that you cannot create a child with the same name as an existing link. If you want to have both a child and a link with the same name, you must create the real child first. You can create a link name that is identical to a real child name. If you use the name, the system looks for a real child first.

If you want to identify the link, precede the name with an exclamation point (!). Thus, the statement *'fmParent.bnChild'* □wi 'self' returns the full name of the actual child, whereas the statement *'fmParent.!bnChild'* □wi 'self' returns the full name of the linked object.

When you use the exclamation point, the system looks for a link only. That is, if you use the name *'fmParent.!bnChild'* when you have a child with that name but no link, the system signals an error; it disowns the real child. The system also signals an error if you use the *Delete* method or reference the *self*

property with a non-existent link name preceded by the exclamation point. (You can use `Delete` or `self` with a non-existent object name without error.)

Drag-and-Drop of Objects

Windows supports the ability to "drag-and-drop" items within or between cooperating applications. The user holds the cursor over an item, presses a mouse button down, and drags the cursor (and the item with it) to the desired location. In some cases, you can move multiple items at a time. The control on which the drag was started is called the "source;" the control where the objects are dropped is called the "target". The "source" and "target" can be the same control. The cooperation required between source and target is that both must agree on the format of the item being dropped.

When the target control receives the dropped item, it determines what to do with it. For example, an APL control could receive a text string that represents a file name. If the target control is a `Label`, it could display the file name itself. If the target control is a multiline `Edit` control and the file contains text, the control could read the file and display the contents. If the target control is a `Picture` and the file is a bitmap, the control could assign the file name to the `bitmap` property and display the image. Any APL control object can receive text, for example a selection dragged from a Word for Windows document (with drag-and-drop editing enabled).

Any APL control can be a target for a drop; if the `targetformats` property of a control is set to one of the recognized formats, the control receives the appropriate events for a target. Target objects receive a `DragEnter` event when a user drags an item over the window area occupied by the control. As the user moves the mouse, the control receives `DragOver` events. If the user drags the item outside the window, the control receives a `DragLeave` event. If the user releases the depressed mouse button over the window, the control receives a `Drop` event. The programmer implements the appropriate response in an `onDrop` event handler. The events fire on the target only if one of the formats listed in the `targetformats` property matches a format the source is willing to provide. (One exception to these general rules is the `RichEdit` control, which handles drag-and-drop at a lower level, bypassing APL drag-and-drop events and properties; for that reason, the APL drag-and-drop events and properties have been disabled on the `RichEdit` control.)

The target formats that are currently available are listed below; you specify one or more of these formats in the `targetformats` property of the control you want to be a drop target. When a user drops an item of the appropriate format on the APL control, the contents are contained in the ninth element of `⎱warg` for the `onDrop` event handler.

In this release, only `Listview` controls and `Tree` controls can be source objects. Other objects may become usable as source controls in future releases. An APL object that is a source receives `DragStart`, `DragContinue`, `DragFeedback` and `DragEnd` events when a user drags an item from it. The `sourceformats` property must be specified.

You can control the actions of the drag-and-drop operation by using `⎱warg` and `⎱wres` with the corresponding event handlers. See the *Windows Interface: Event-handler Properties* chapter in this manual for more information.

The formats that APL recognizes for drag-and-drop operations include the following:

- **FileName**
A nested vector of character vectors containing file names. The Windows Explorer provides this format if you drag one or more files from the Explorer to an APL window. The underlying Windows format is Clipboard format `CF_HDROP`.

- **FileDrop**
 A nested vector of character vectors containing file names. This format is still used by some applications that do not use OLE Drag-and-Drop.
- **Text**
 A character vector. The underlying Windows format is Clipboard format CF_TEXT.
- **ListviewItem**
 All the information about one or more entries in a Listview object. This has the same format as the result from the GetRows method of the Listview object. The underlying Windows format is Registered Clipboard format "APL_ListviewItem".
- **TreeItem**
 All the information about one node of a Tree object, including all its descendants. This has the same format as the result from the GetInfo method of the Tree object. The underlying Windows format is Registered Clipboard format "APL_TreeItem".

When the user is dragging an object, certain behaviors are inherent in the Windows operation. For example, as the item is dragged across an object that is not a valid target object, the cursor turns to an International No symbol indicating the user cannot drop the item on that object. By default, the item being dragged is represented as a box with a dotted outline. You can set the `dragimage` property for a tree or listview control to change the cursor image to represent the item in a more graphic manner.

The first example below shows how to enable drop on an Edit field, so a user can drag the name of a file into the field from Explorer or another APL+Win control as an alternative to typing it. The first statement creates an edit control wide enough to hold a simple file name; the second defines the format of objects that can be dropped on this edit control. During the `onDrop` handler, the file name is contained in the ninth element of `□warg`; in this example you are accepting only the first file name if multiple files are dragged.

```
edit←'fmSave.edFilename' □wi 'New' 'Edit' ('where' 1 1 1.5 20)
edit □wi 'targetformats' 'FileName'
edit □wi 'onDrop' '□wi ''text'' (1>9>□warg)'
```

In the next example, the edit control can accept either text or a filename. If the object dropped is text, it becomes the text property. If the object dropped is a filename, the control uses the contents of the file. During the `onDrop` handler, the eighth element of `□warg` specifies the format. As with the example above, when the format is a file name, you are handling only the first file in the argument. Note that when the format is text, the ninth element is a simple vector rather than a nested vector.

```
edit←fm □wi '.edEdit.New' 'Edit'
edit □wi 'where' 1 1 12 38
edit □wi 'style' 4 16 32 8192
edit □wi 'targetformats' 'Filename' 'Text'
edit □wi 'onDrop' 'CaptureText'
```

where `CaptureText` is the following function:

```
▽ CaptureText tie;filename;content
[1] :select t←8>□warg
[2] :case 'FileName' A Put contents of file into edit
[3]     filename←1>(□warg[9]) A Use just the first file
[4]     filename □tie tie←~1+L/0,□nnums,□xnums
[5]     content←((□nread tie 82 32000 0) ~□tclrf)
[6]     □wi 'text' content
[7] :case 'Text' A Put dragged text into edit
[8]     □wi 'text' (□warg[9])
[9] :endselect
▽
```

Note that when you drag Listview or Tree items, you are using APL formats. In these cases, you can drag-and-drop multiple items with one action, for example, by using the `AddRows` method for a Listview.

Using Graphics with the Draw Method

The Draw method provides the means to draw graphics on APL controls, such as forms, frames, and pictures, or on a printer object. It uses a set of drawing operations that form a simple graphics sublanguage. By invoking the Draw method on an object and specifying the graphics operations you want to perform, you can achieve a wide variety of graphical effects. See the section on the Draw method in this manual for details on the full range of drawing operations.

The syntax of the Draw method is:

```
'object' ⍵wi 'Draw' ('op1' arg1 arg2...) ('op2' arg1 arg2...)
```

where *op1*, *arg1*, etc are the names of the various drawing operations and their appropriate arguments. If you omit the left argument, the Draw method applies to the current object, `⍵wself`.

Arguments to the operations of the Draw method are expressed in terms of the scaling units for the object, which are defined by the first element of its `scale` property. If `scale` is 3, the operation is measured in terms of points (72 units to the logical inch), while if `scale` is 5, the units are in terms of pixels. Coordinates are relative to the upper left corner of the client area of the object, like the `size` property. The examples in this section use a square form with the scaling units set to pixels. The variable `⍵wself` is set to the name of the form to simplify the statements of the examples. The statement below performs both those actions. If you want a white background so the images display better, you can also set the `color` property. You may want to resize and move your APL session window and the form so you can see them both while you do the Draw statements.

```
⍵wself←'fmGraph' ⍵wi 'New' 'Form' ('scale' 5) ('size' 200 200)
⍵wself ('color' 255 255 255)
```

Note: Many different factors affect the size and shape of graphics on your screen. If you do not see the images after you execute the statements, or if your circles are not round and your squares are not square, you should check the following items:

- **Property settings**

The `scale` property of the object you are drawing on is crucial to the graphics. In particular, the default setting for `scale` specifies characters, which are not the same height as width. If you specify a 2-by-2 rectangle with characters as the scaling unit, you do not get a square shape; you get a rectangle that is twice as high as it is wide.

The value of `scale` also applies to the size of your object; if you specify the location of a graphical image that works with one setting of `scale`, the image may be completely off the form with another setting of `scale`. If you do not see any image resulting from a well-formed statement, check the `size` property.

- **Arguments**

The aspect ratio argument to `Circle` (the fourth element of the argument) defaults to 1. If you want to draw circles with character scaling units, use an aspect ratio of 0.5. If you do not see any image resulting from a Draw operation, make sure you are not using a transparent pen.

- **Physical settings on your monitor**

Most monitors have settings that affect the size and placement of output on the screen. You can also adjust these settings, if necessary.

To draw a square in the center of the form `fmGraph` with sides whose length is one-half that of the form, use the following statement:

```
⍵wi 'Draw' ('Rect' 50 50 100 100)
```

Note that you specify the position of the upper left corner of the rectangle. In contrast, when you draw a circle, you specify its center. To place a circle in your box, you use the following statement.

```
␣wi 'Draw' ('Circle' 100 100 50)
```

Generally, you draw a graphical image with a series of Draw method operations. These graphical images are visible immediately on the object as they are drawn; the effects are cumulative. In the examples above, if you draw the circle first and then the square, the square covers the circle. (If you are drawing on a printer, the images are not immediately visible; you must complete the whole series of operations and then close the printer object.)

If you draw on a form or a frame and the image of the object is damaged in some way, for example, by being covered by another window and then uncovered again, you must redraw the graphical image to restore the appearance you want. You may also need to do this if the size of the object changes in a situation where you want the graphics to be proportional to the size of the window. The Picture object is designed to maintain a persistent graphical image.

As you try these examples, clear the form so pieces of previous images don't clutter the screen. You can use the statement `␣wi 'Draw' 'Clear'` at any time to perform that action. If you are doing a lot of drawing, you may want to put the Clear operation on a tool. To do this, go to the Options menu, select Tools, and click on Add in the Define Tools window. In the New Tool window, name the tool with a shortcut key, and enter the Clear statement as the command. For example, you could enter the following lines:

- Menu Item: Draw Clear F7
- Command: `␣wi 'Draw' 'Clear'`

Thereafter, when you press F7 with the focus in your Session window, APL clears the current object.

The best way to redraw a graphical image is to record the sequence of Draw operations that you used to produce it. This list of operations is known as a *display list*; the system can regenerate an object's graphical image when necessary by re-executing the Draw method with this argument.

The most convenient place to store the display list for an object is in its `data` property or a user-defined data property. For example, to draw three circles on an object, you can set the object's `data` property to contain the display list, and then draw the list using `data`.

```
d←c 'Circle' 20 20 10
d←d,c'Circle' 40 80 30
d←d,c'Circle' 60 60 54
␣wi 'data' d
␣wi 'Draw' (␣wi 'data')
```

To cause the graph to redisplay itself automatically any time it is necessary, define a handler for the Paint event that redraws the display list:

```
␣wi 'onPaint' '␣wi ''Draw'' (␣wi ''data'')
```

Now, clear the form. You can trigger the `onPaint` event handler in several different ways:

- By minimizing `fmGraph` and restoring it
- By causing the system to repaint `fmGraph`. If you overlap your Session window and `fmGraph`, for example, you can observe the behavior by clicking on the Session window, pressing your shortcut key to clear the form and clicking on `fmGraph`.
- By using the `Paint` method: `␣wi 'Paint'`

The examples in this chapter show basic graphical operations. You can perform more complicated graphical operations, including those that incorporate existing bitmaps, for example, by using the Picture object. See the Windows Interface: Classes chapter in this manual for more information.

Draw Operations

Graphics operations fall into two categories: *rendering* operations, which actually draw graphics or text, and *state* operations, which define characteristics of the rendering operations. For example, in the statement below, the state operation `Color` sets the state for the rendering operation `Text`, so the system draws the text you specify in red on a white background:

```
□wi 'Draw' ('Color' (255 255 255)(255 0 0)) ('Text' 'Hello' 7 2)
```

Note that the state operation applies only to the rendering operations that come after it in the same statement. If you reverse the order of `Color` and `Text`, or if you put them in the proper order but in separate statements, the system draws the text in default black.

The first table below describes the state operations for the Draw method, and the second table describes the rendering operations.

State Operations for the Draw Method

Operation	Description
Brush	Defines the color and texture Draw uses to fill the inside of closed figures.
?Brush	Creates a predefined brush and returns its handle.
Clip	Limits the region for subsequent rendering operations.
Color	Defines the color for subsequent Text operations.
Font	Defines the font for subsequent Text operations.
Mode	Defines the drawing mode for subsequent rendering operations.
Pen	Defines the pen for subsequent rendering operations that draw lines.
?Pen	Creates a predefined pen and returns its handle.
Scale	Defines the scaling units for subsequent rendering operations.
Window	Defines a new region for subsequent rendering operations.

Rendering Operations for the Draw Method

Operation	Description
?Bitmap	Loads a bitmap from a file into Windows memory and returns a bitmap handle. The argument can be the name of a file with extension .ICO or .BMP.
Bitmap	Draws a bitmap using either a file name or a bitmap handle.
Chord	Draws a figure defined by the intersection of an ellipse with a line segment.
Circle	Draws a circle, ellipse, or pie wedge.
Clear	Clears the specified object; for Printer objects, prints and ejects the current page.
Ellipse	Draws an ellipse (see Note).
Icon	Draws an icon; the argument can be the name of a .DIB, .ICO, or .BMP file.
?Icon	Loads an icon from file (.ICO extension) and returns an icon handle.
Imagelist	Draws an image from an Imagelist object.
Line	Draws a series of connected line segments.
Point	Draws a set of points.
Poly	Draws a closed polygon.
Rect	Draws a rectangle.
Seg	Draws multiple series of connected line segments.
Text	Draws text.
?Text	Calculates the size that a certain text string will occupy when drawn in the current font.

Note: You can draw an ellipse with the `Circle` property by specifying its center, a radius, and an aspect ratio. The `Ellipse` operation takes an argument similar to `Rect`. You specify the upper left corner of the area in which you want the ellipse to fit. The height and width measurements become the axes of the ellipse.

Pens and Brushes

The Draw method adopts the concepts of pens and brushes as they are used by Windows GDI (Graphics Device Interface; used for device-independent graphics). A *pen* draws line segments and the outside borders of circles, rectangles, etc. A *brush* fills closed figures such as circles, rectangles, or closed polygons. Pens and brushes are data structures that Windows maintains; you can create a pen or brush and use it one time or many times. In a typical application, you might create several different pens, each of which you define to have specific characteristics, such as color, and select the appropriate pen for different parts of the image.

To define a pen, you specify its *style*, and optionally its *color* and *size*. Once you define these characteristics, you can draw with the pen. The size is the width of the pen in pixels; the default is 1. If you want any of styles 3 through 6, you must use a width of 1; style 2 affects only pens that are wider than 1 pixel.

You express the color as a 3-element vector of RGB values ranging from 0 to 255; it defaults to the current foreground color as set by the object's color property. The allowed values for pen styles include:

Pen Styles for the Draw Method

Style	Description
0	Transparent
1	Solid (default)
2	Solid inside (See Note)
3	Dash
4	Dot
5	Dash-Dot
6	Dash-Dot-Dot
-1	Used to select an existing pen handle

Note: When you draw closed figures with pen widths wider than 1 pixel, style 1 centers the line on the boundary, while style 2 draws with the outer edge of the line at the boundary (the line is entirely within the figure).

To define a brush, you specify its style and color. You specify the color in the same way as for a pen. The allowed values for brush styles include:

Brush Styles for the Draw Method

Style	Description
0	Transparent (do not specify a color)
1	Solid (default)
2	Horizontal hatching
3	Vertical hatching
4	Backslash diagonal hatching (upper left to lower right)
5	Slash diagonal hatching (lower left to upper right)
6	Cross hatching
7	Diagonal cross hatching
-1	Used to select an existing brush handle

Style	Description
-2	Used to select a bitmap handle or bitmap file name

You can create a temporary pen or brush as part of the argument to the Draw method; for example:

```

□wi 'Draw' ('Pen' 1 (0 255 100) 3) ('Line' (2 2097 97 200 200))

```

It is more common and efficient to define a collection of pens and brushes at the start of an application, and then use these throughout. For example, a barchart program might use one pen to draw the first bar in a barchart, a second pen to draw the second bar, and so on. In a complete application, the end user may be able to customize the graph by selecting the characteristics of each bar's pen and brush.

The following statements create vectors of pen handles and brush handles to use in subsequent Draw operations. The first pen is dashed blue, one pixel wide; the second is solid black, 2 pixels wide. The first brush is blue vertical hatching; the second is red cross hatching.

```

pens←□wi 'Draw' ('?Pen' 3 (0 0 255)) ('?Pen' 1 (0 0 0) 2)
brushes←□wi 'Draw' ('?Brush' 3 (0 0 255)) ('?Brush' 6 (255 0 0))

```

To select an existing pen or brush, use -1 and the desired handle; for example:

```

□wi 'Draw' ('Pen' -1 (pens[1])) ('Brush' -1 (brushes[2]))
  ( 'Circle' 130 120 40)

```

Scaling in Graphics

An object's scale property defines the scaling units for interpreting the position and size parameters you specify with the Draw method as well as the where property values of the object's children. When you reference it, scale is a five-element vector that defines the current scaling units and the size of the object's client area in these units. The first element of the scale property is the current scaling unit; the possible values include:

Scaling Units for the Draw Method

Scale	Description
0	Proportional units (defined in terms of 1÷scale)
1	Character units (height and average width of a character in the system font)
2	Dialog units (1/8 of system font high and 1/4 of system font wide); scaling units often used by external dialog box editors
3	Points (72 points per logical inch; 28.3 points per logical centimeter)
4	Twips (1440 twips per logical inch; 567 twips per logical centimeter)
5	Pixels

The other elements of the scale property are the origin value and the height and width of the object's client area in the current scaling units. For example, suppose you create a picture and specify its size using the where property. The scale property would be similar to what is shown below:

```

pic ← 'fmGraph.pic' □wi 'New' 'Picture'
  ('where' 25 25 50 70) ('border' 0) ('color' 235 255 150)
pic □wi 'scale'
1 0 0 2.5 7

```

The height and width are different from the specification because the where property uses the parent's scaling units (pixels), but the scale property defaults to character units. If you change the scale for the picture to match the form, the result would be as shown below.

```

pic □wi 'scale' 5
pic □wi 'scale'
5 0 0 50 70

```

The `scale` property shows that the upper left corner of the drawing area for the picture is position (0 0) in the current scaling units and the size of the picture is 50 by 70 pixels. (The upper left corner of the picture is at position 25 25 relative to the drawing area of the form.) The lower right corner of the picture is at position 50 70 relative to the picture. (The designation of that corner relative to the form is 75 95.)

You can draw a line from the center to the lower corners of the picture with the following statement:

```
pic □wi 'Draw' ('Line' (3 2ρ50 0 25 35 50 70))
```

Note that if you use a border on your picture, that is, let the `border` property default to 1, the drawing area is reduced by the amount of space the border occupies.

Changing the scaling units changes the `scale` property (but has no effect on how large the picture appears on the screen). Although `scale` is a five-element vector, you can specify just the first element. In addition to characters and pixels, you can change the `scale` to dialog units (2), points (3), or twips (4). If you step through this exercise, you may find that your results differ slightly. The units vary based on the resolution of your display and the system font in use.

```
pic □wi 'scale' 2
pic □wi 'scale'
2 0 0 20 28
pic □wi 'scale' 3
pic □wi 'scale'
3 0 0 30 42
pic □wi 'scale' 4
pic □wi 'scale'
4 0 0 600 840
```

If you set the first element of `scale` to 0, the system defines the coordinate system in terms of the object's current size instead of any fixed units. This allows you to make a drawing that is proportional to the current size of the object that contains it. You can use this technique to put a graph on a form that the user is able to resize, and write the `onResize` event handler to draw the graph using the same values. The graph will automatically scale itself to the size of the window when the application redraws it. For example, to use units that are expressed as a percentage of the current window height and width, use this statement:

```
□wi 'scale' 0 0 0 100 100
□wi 'Draw' ('Circle' (50 50 50))
```

Or, to plot points on a scale that runs from -2 -2 to 2 2:

```
□wi 'scale' 0 -2 -2 4 4
□wi 'Draw' ('Line' (2 2ρ0 0 2 2))
```

This allows you to draw points on a graph in terms of their values without calculating their positions. You can also draw a graph with the traditional display where positive values go up and to the right by reversing the direction of the vertical axis. However, APL still specifies coordinates in *row/column* order, as opposed to the typical mathematical x-y axes being horizontal/vertical.)

```
□wi 'scale' 0 2 -2 -4 4
□wi 'Draw' ('Line' (4 2ρ0 0 1 .7 1 1.3 2 2))
```

Modes of Drawing

The drawing mode defines aspects of how text or shapes that are drawn interact with the image that is already present on the object. You can specify the mode for the duration of a single Draw operation, by using a statement similar to the following:

```
□wi 'Draw' ('Mode' ⍉ ⍉ 1)
```

Or, you can define mode as a property of an object.

```
□wi 'mode' ⍉ ⍉ 1
```


The `mode` property is a three-element vector, comprising the *transparency* mode, the *raster-op* mode, and the *polygon fill* mode. You can set one of these and leave the other items unchanged by using an empty vector ($\emptyset$) for the items you do not want to change. The examples above set the polygon fill mode, leaving the transparency and raster-op modes unchanged.

The transparency mode affects whether the background color is used when drawing text with the `Text` operation. The default value of zero means the text background is transparent. You can set a value of 1 to make the background color opaque.

The raster-op mode defines the rule by which the lines, shapes, or points being drawn are combined with the pixels that are already present in the Window. The resulting color drawn in each pixel is a function of the color of the current pen and the color of the pixel that is already there. There are sixteen possible functions, each of which corresponds to one raster-op mode.

The default and most useful raster-op is 13, which simply draws in the current pen color without regard for the present value of the screen pixel. Another useful raster-op is 7, XOR, which draws the exclusive-OR of the pen color and the pixel color. This is useful because if you repeat the same operation, it restores the pixel to its original color.

If you want to outline a region of an image by drawing a rectangle temporarily, use the XOR raster-op with a black pen, which inverts the image. Repeating the same operation restores the pixels to their original colors and removes the rectangle. A third useful raster-op is 6, which simply inverts the present color independent of the current color of the pen.

The full set of the raster-op functions is shown in the description of the `mode` property in the Windows Interface: Properties chapter in this manual.

The polygon fill mode determines how `Draw` fills the interior of polygons that are drawn with intersecting lines. *Alternate* fill is the default (0); it specifies that `Draw` uses the current brush to fill only those areas that can be reached from outside the figure by crossing an odd number of lines. *Winding* fill (1) specifies that the entire interior region of the polygon is filled.

To observe the different polygon fill modes, execute the following statements.

```

□wi 'scale' 0 0 0 5 5
polygon←8 2p1 1 4 1 4 4 2 4 2 2 3 2 3 3 1 3
□wi 'Draw' ('Poly' polygon)
□wi 'Draw' ('Mode'  $\emptyset$   $\emptyset$  0) ('Brush' 2 -1) ('Poly' polygon)
□wi 'Draw' ('Mode'  $\emptyset$   $\emptyset$  1) ('Brush' 3 -1) ('Poly' polygon)

```

Bitmaps and Icons

Bitmaps and icons are similar concepts. An icon is a fixed-size rectangle of pixels that used to represent various items such as an application or a mouse cursor with a special meaning. A bitmap is a rectangle of pixels of arbitrary height and width. You can compile both icons and bitmaps into an executable Windows program (such as `aplw.exe`) or a 32-bit Dynamic Link Library (.DLL) or store them in files with extension .ICO or .BMP respectively. You can create icons and bitmaps using Windows development tools such as Microsoft AppStudio or Borland Resource Workshop. You can also create *ad hoc* bitmap images in APL+Win using a Picture object.

Icons have several special characteristics that distinguish them from bitmaps. An icon file can contain several versions of the icon in different screen resolutions. An icon can also have pixels that are transparent, meaning that the background color shows through when the icon is drawn.

The `Icon` operation draws one or more icons on the screen. Its argument consists of the name of an icon, followed by the position where you want the upper left corner of the icon. The icon name can be the name of an icon resource that has been compiled into the APL interpreter, the name of a .ICO file, or it can be the name or number of an icon in a resource file. For example:

```
␣wi 'scale' 5
␣wi 'Draw' ('Icon' ('aplw.ico' 10 10) ('bug.ico' 10 44))
␣wi 'Draw' ('Icon' 'c:\Windows\forms\configs\Mapif21.ico' 50 10)
␣wi 'Draw' ('Icon' '#15' 50 44)
```

In the last example, the resource file must be identified in the Resource DLL entry in APLW.INI. To load a resource, the system first looks in the DLL or EXE named in that entry, then in the APLW.EXE file. You can identify a resource by a number, as in the example, or by a name.

To load a file, the system uses the full path name if present. Otherwise, the system looks in directories specified in the Resources entry of the .INI file, then in the current directory.

The system can draw icons faster if you preload them using the `?Icon` operation, which loads an icon into Windows memory and returns an icon handle.

```
icons←␣wi 'Draw' ('?Icon' 'aplw.ico' 'bug.ico')
␣wi 'Draw' ('Icon' (icons[1] 10 10) (icons[2] 10 44))
```

The `Bitmap` operation draws one or more bitmaps on the screen. Its argument is the name of a bitmap, followed by the position of the upper left corner of the bitmap, followed optionally by the height and width of a rectangle into which the bitmap is clipped. The name of the bitmap can be either the name of a bitmap resource compiled into the APL interpreter, the name of any .BMP file, or it can be the name or number of a bitmap in a resource file

```
␣wi 'Draw' ('Bitmap' 'hello.bmp' 10 10)
␣wi 'Draw' ('Bitmap' 'c:\Windows\arcade.bmp' 10 10)
␣wi 'Draw' ('Bitmap' 'joe' 10 90)
```

As with icons, the system can draw bitmaps faster if you preload them using `?Bitmap`, which returns the handle of the bitmap.

```
hbmLogo←␣wi 'Draw' ('?Bitmap' 'hello.bmp')
␣wi 'Draw' ('Bitmap' hbmLogo 10 10 30 50)
```

In the above example, the bitmap is drawn starting at pixel address (10 10). If the bitmap is larger than 30 pixels high by 50 pixels wide, the system clips it so it does not exceed that size.

If you have an `Imagelist` object in your application, you can also use the `Imagelist` operation of the `Draw` method to display an image contained in the object. You can also set the `bitmap` property of a `Picture` object to automatically display a bitmap when your application opens the form containing the picture.

Freeing Bitmap, Icon, Pen, and Brush Handles

You discard pens or brushes using the `Wfree` function from the Windows workspace. This releases the Windows memory allocated to hold a loaded bitmap or icon or to represent a pen or brush. Once you finish with these objects, free them. If you do not, they continue to occupy Windows memory until the next time Windows is shut down.

To free one or more handles of any of these types, provide them as arguments to the `Wfree` function. For example:

```
      )Copy Windows Wfree
SAVED 02/06/1994 13:24:07
      Wfree pens brushes icons hbmLogo
```

Using Dynamic Data Exchange

Note: Dynamic Data Exchange (DDE) connectivity causes multiple instances of APL+Win operating as ActiveX servers to wait until they are all closed before exiting. It may also have other consequences. Therefore, DDE is disabled by default in this release. To enable DDE connectivity in your application, you must set the setting for `[Config]EnableDDE` in the `APLW.INI` file to 1.

Dynamic Data Exchange (DDE) is a mechanism that allows Windows applications to interact with each other, using a standard Windows protocol. An application that supports DDE can exchange data with another application, and it can send or receive commands, without needing to know anything about the internal structure of the other application. DDE provides a standard way of exchanging information. However, please note that since the use of the Component Object Model (COM) protocols has become widespread, DDE is considered largely obsolete for new applications.

This section provides the information you need to set up a DDE link with an APL+Win application as one or both of the interacting applications. First, it defines the basic DDE terminology; next, it explains how that extends to APL and provides an example of the basic functionality of DDE. Later sections provide more detail on how you use the full range of DDE capability in APL+Win.

Be aware that having DDE enabled causes multiple copies of APL running as ActiveX servers to wait until all of them are closed before exiting. It may also have other consequences.

Understanding DDE Terminology

Windows applications use DDE to interact with each other by participating in *conversations*. In a conversation, there is one *server* application (sometimes called the source) and one *client* application (sometimes called the destination). Typically a server supplies data to a client, either automatically or upon request; a client can also issue commands or send data to a server.

A server application makes itself available for DDE conversations by registering itself with Windows. It specifies an *application* name for itself (usually the name of its executable file, without a path or extension), and it specifies one or more *topics* on which it is prepared to open conversations. A server must also recognize a name that identifies the data it will send (or receive); you can think of this name, which is the *item*, as the location of the data in the server.

A widely-used convention is for a topic to be associated with one document (for example, a topic might be the name of a spreadsheet), while items are individual data elements within a document: a single spreadsheet cell, such as R1C1 in Microsoft Excel, is an example of a typical item.

Note: All strings used as application names, topics, and items are treated as case-insensitive: R1C1 is equivalent to r1c1.

A client application initiates a conversation by asking Windows to connect it with a server that has registered a specific application and topic. If a server registered itself with this application and topic, a conversation begins. The client then specifies *items* of data on which it wishes to be informed. A conversation may involve exchanging one or more data items; during a conversation, a data item with a specific name may be transmitted multiple times with different values. Data items may be exchanged automatically (a hot link) or manually (a cold link). In a hot link, the server informs the client every time a data item changes; in a cold link, the client must explicitly ask for a data item.

How the Terms Apply in APL+Win

For DDE communication, APL+Win treats data items, including numbers, as strings of text. In an APL application, these data are the text property of an Edit control or the caption property of a Label control. Label and Edit controls act as clients, while Form objects act as servers; on a server form, you can have either an Edit or a Label as the item.

APL+Win uses special properties and methods for DDE communication. You can also define certain event-handler properties that correspond to DDE events within the APL application; however, these event handlers are not part of the DDE protocol. The table below shows the properties, methods, and event handlers that APL uses for DDE.

Elements of DDE in APL

Properties	Methods	Event Handlers
ddeApplication	DdeExecute	onDdeConnect
ddeItem	DdePoke	onDdeDisconnect
ddeMode	DdeRequest	onDdeExecute
ddeTimeout		
ddeTopic		

Label and Edit controls have all five properties; a Form has only the last one. You use the methods on Label and Edit controls when they are clients. You can write onDdeConnect and onDdeDisconnect event handlers for a form as the server and for a control as a client, but not for a control as the item on a server form. You can write an onDdeExecute handler only for a form.

Using APL for DDE Communication

The minimum requirement to initiate a DDE conversation is that two applications agree on an application and a topic, and the client must specify the mode. To actually transfer data, the two applications must agree on an item name. If the client specifies a hot link, the server sends the data whenever the value of the item changes; if it is a cold link, the client must request the data.

Establishing APL as a Server

The application name of a server is the same as the name of the currently-executing .EXE file, without a pathname or extension. For example, if the executable file is C:\APLWINxx\aplw.exe, the application name is APLW. If you rename the file (for example, in a runtime system), the application name changes. You do not have to specify the application name within your APL application to act as a server; the server name is automatically derived from its .EXE file name. To make a form act as a server, specify a non-empty ddeTopic property for that form; for example:

```
'fmServer' ⌵wi 'ddeTopic' 'sometopic'
```

When a form with a non-empty ddeTopic opens, it registers itself with Windows as a server. If a form is already open, changing its ddeTopic from ' ' to a non-empty vector causes the form to register itself as a server. You can change a server form's ddeTopic from one non-empty value to another while it is open; this causes it to unregister the first topic and disconnect any conversations; it then registers itself as a server on the new topic. The server plays a passive role in the DDE conversation. After it registers itself with Windows, it waits for a client to establish the conversation. Whenever a client opens a conversation with a server form, the system executes the form's onDdeConnect event handler.

When a form closes (or when you set its ddeTopic to ' '), it unregisters itself with Windows, ends any conversations, and the system executes the form's onDdeDisconnect event handler.

Transferring Data Items as a Server

A client can open a conversation with the form without specifying a data item; however, there can be no transfer of data in this case. If APL will be sending data to a client, one or more Edit or Label controls on the server form must have a non-empty `ddeltem`. The value of the `ddeltem` property is the name that the client application uses to identify the data item. It is the equivalent of `r1c1` or `R1C1` in the spreadsheet example given earlier. You specify an item with a statement like the following:

```
'fmServer.edItem' ⍵wi 'ddeltem' 'linkname'
```

When a client specifies one or more data items for hot or cold links, an APL form that is acting as a server checks the `ddeltem` properties of all Edit or Label controls that are its children to see if any match the requested items. (Note that `ddeltem` is a property of the controls, not of the form). If a control exists with a matching `ddeltem` property, the form sets up a hot or cold link to the client.

The server sends data to the client automatically in APL+Win. The value of the data item that APL sends to the client application is a character vector that is the `text` property of an Edit control or the `caption` property of a Label. In a hot link, any change to the control's contents causes the parent form (the server) to immediately transmit the new data to the client. This can occur when a user types in the Edit control that is the item, or when you change the `text` or `caption` property of the item under program control. In a cold link, the server form responds to requests from the client. When the client explicitly requests data from an item, the form transmits the contents (`text` or `caption`) of the control whose `ddeltem` matches the requested item; this also happens automatically without any indication in your application.

The client can also send data to the server. In this case, APL places the data item into the control whose `ddeltem` property matches the name of the client's data item (that is, the value of the client's data item becomes the `text` property for an Edit control or the `caption` property for a Label control).

This action triggers the `onChange` event handler for the control. Generally, you use the `onChange` event handler to process the data immediately upon receipt, because you do not know when the client will transmit another data item. Note that this is the only time that APL executes the `onChange` event handler for a Label.

Receiving a Command as a Server

In addition to transmitting data through DDE, a client application can send a command to the server; when this happens, `⍵warg` is a character vector that contains the client's command. When a client issues a command to an APL form that is acting as a server, the system executes the form's `onDdeExecute` event handler. Note that the item is not involved when the client sends a DDE Execute command.

If both client and server applications are APL, you can receive a text string containing any symbols that APL can recognize. The APL application that receives the string can execute the statement through an `onDdeExecute` event handler that is as simple as `'⍵⍵warg'`.

If the sending application is not APL, it probably has no method for sending APL characters. However, you can still send some commands from a client application that does not support the APL character set. For example, you could use DDE to open APL and have APL load a specific workspace, whose latent expression (`⍵1x`) performs an action useful to the sending application.

Establishing APL as a Client

It is equally easy to define APL as a client. Other applications are usually more difficult to define as the client, although they may automatically set themselves up as a server. Many applications register themselves as servers with a topic of "SYSTEM" each time they start, allowing remote applications to access

them whenever they are running; see the documentation on each application for details on what commands they accept.

The Form object's properties are not involved when APL is the client. To make a Label or Edit control act as a client, specify a non-empty `ddeTopic` for the control and specify the `ddeMode` for the client control as

```
'hot' or 'cold'; for example:  
'fmA.edClient' □wi 'ddeTopic' 'srvrapp|srvrtopic'  
'fmA.edClient' □wi 'ddeMode' 'hot'
```

Note that for clients, the `ddeTopic` property includes both an application name and a topic name, separated by either the stile (|) or the split-stile (:) character. A space is a valid character in a topic name, so you cannot place spaces around the stile character unless the server also includes those spaces in its application and topic names. (You can specify the application and topic names separately, but the behavior differs; this alternate method is explained later in this chapter.)

The application name is normally the name of the executable file (without pathname or extension) of the application with which you want to link. With some applications, for example Quattro Pro for Windows, version 5.0, you must specify a fully-qualified path name. The topic name must be a name that the server recognizes as a topic; for example the name of a specific spreadsheet.

When a control with a non-empty `ddeMode` opens (usually, when you open its parent form), it tries to start a conversation on the application and topic names specified in `ddeTopic`. If a control is already open, setting its `ddeMode` to 'hot' or 'cold' attempts to start a conversation. The `ddeTimeout` property determines how long a client control waits when trying to establish a connection with a server. You specify this value in milliseconds; the default is 5000. If a server is registered, the two systems establish a link. When a link is established, the system executes the client control's `onDdeConnect` event handler

If you want the client control only to send commands to the server and not receive data, you can leave the control's `ddeltem` property empty. In this case, it doesn't matter whether you specify 'hot' or 'cold' as `ddeMode`; they both have the same effect.

Transferring Data Items as a Client

If you want the client control to receive data from the server, you must set the control's `ddeltem` to a non-empty character vector. The item name must be one that the server application recognizes; for example, a cell in a spreadsheet or a field in a database. You can set the item with a statement like the following:

```
'fmA.edClient' □wi 'ddeItem' 'srvritem'
```

The remote server sets up a hot or cold link on the specified data item. Once a connection is established with the item specified, the client can receive data from the server; it can also send commands or data to the server. The server application sends the contents of the specified cell or field to the APL client. In a hot link, the server sends data whenever the specified item changes; this triggers an `onChange` event in the client control. You can change an open control's `ddeMode` from 'hot' to 'cold' (or vice versa). The systems convert the link appropriately.

In a cold link, the server sends data only when requested. The APL client control requests the data with the `DdeRequest` method, which causes the server to send the data and trigger the `onChange` event handler in the client control. You request data in a cold link with a statement like the following.

```
'fmA.edClient' □wi 'DdeRequest'
```

Client controls need not be visible on your form. Especially in a complex application, it may be appropriate to hide the client controls and use the `onChange` event handler to direct the data to their ultimate destination.

When you have only one client and the control closes (or when you reset any of the relevant properties), it ends the conversation with the server. When a conversation ends with the control still open, the system executes the client control's `onDdeDisconnect` event handler.

If you have multiple client controls, APL+Win opens a single conversation for all the controls on a single form that have the same `ddeTopic`. You can turn the links for each control on and off individually, by setting their `ddeMode` properties to `'hot'`, `'cold'` or `''`; the system disconnects the conversation when the last link is turned off.

Simple Examples

You can see the effect of DDE using APL as both server and client with the following commands. Although you can create both forms in the same APL session, it is a better demonstration of DDE if you use two APL sessions, one as the server and the other as the client.

Session One

```
'fmServer' □wi 'New' 'Form' ('where' 1 1 10 20)
'fmServer' □wi 'ddeTopic' 'Minimum'
'fmServer.edItem' □wi 'New' 'Edit' ('where' 1 1 1.5 12)
'fmServer.edItem' □wi 'ddeItem' 'Needed'
```

Session Two

```
'fmClient' □wi 'New' 'Form' ('where' 1 25 10 20)
'fmClient.lblItem' □wi 'New' 'Label' ('where' 1 1 1.5 12)
'fmClient.lblItem' □wi 'ddeTopic' 'APLW|Minimum'
'fmClient.lblItem' □wi 'ddeItem' 'Needed'
'fmClient.lblItem' □wi 'ddeMode' 'Hot'
```

Now type in the edit control on the server form; you should see the same characters in the label control that is the client. You can also specify data to be transmitted; in Session one, type:

```
'fmServer.edItem' □wi 'text' 'send it now'
```

Change the mode to a cold link and observe the difference. In Session two, type:

```
'fmClient.lblItem' □wi 'ddeMode' 'Cold'
```

Change the value of the data item by typing in the edit box on the server form. Then, in Session two, type:

```
'fmClient.lblItem' □wi 'DdeRequest'
```

You can also send data from the client to the server. In Session two, type:

```
'fmClient.lblItem' □wi 'caption' 'Send back'
'fmClient.lblItem' □wi 'DdePoke'
```

Specifying the Application Name in `ddeApplication`

The usual way to specify application name and topic is to combine them in `ddeTopic`, separated by a stifle (`|`), as described above; for example:

```
'fmA.edClient' □wi 'ddeTopic' 'EXCEL|SHEET1.XLS'
```

If the application is not running, the system signals an error; you can trap this error, if desired, and execute `□wcall 'WinExec' 'name-and-path' 'window-style'`

where *name-and-path* is the name of the application and *window-style* controls how the application is shown; for example, `SW SHOWNORMAL`, to start the application. Then you can try again. As an alternate technique, you can specify the application name in the `ddeApplication` property.

```
'fmA.edClient' □wi 'ddeApplication' 'C:\EXCEL\EXCEL'
```

When `ddeApplication` is non-empty, APL makes two attempts to connect. First it tries to connect with the specified application name and topic (stripping off pathnames and extensions). If this fails, it automatically tries to execute `ddeApplication`; it then tries to connect again.

This method is convenient in some cases, but undesirable if the remote application is already running and simply unable to connect on the specified topic. In this scenario, you end up with multiple instances of the remote application, none of them connected. Using this technique, you may still need to put the application name in `ddeTopic`.

Transferring Multiple Numbers in DDE

By a common Windows convention, when more than one numeric element is transferred, tab characters (`␣`) are used to separate columns of numbers, and newlines (`␣`) separate rows. This applies not only to DDE but to data sent to and from another application via the clipboard. (The APL+Win numeric editor follows the same convention.) The `DDEFmt` and `DDEUnfmt` functions in the `WINDOWS` workspace provide for conversion between APL arrays that hold numeric values and this tab-delimited numeric-character format.

Specifying a Spreadsheet Range for the Server Item

When using APL+Win as a client to connect to a spreadsheet, you can specify a range of cells as the server's item. Values in the same row are separated by tab characters; rows are separated by newlines. The following examples show typical formats for the data item:

- Excel
Specify the starting and ending cells separated by a colon.
`'fmA.edClient' ␣wi 'ddeItem' 'R1C1:R3C4'`
- Lotus or Quattro Pro
Specify the starting and ending cells separated by two periods.
`'fmA.edClient' ␣wi 'ddeItem' 'A1..C4'`

Using DDE Methods: A Summary

There are three DDE methods in APL; only client controls use them. These methods determine the type of communication with the server when you use a cold link. When you are using a hot link, the data are transmitted automatically.

- The `DdeRequest` method requests data from the server in a cold link. You do not specify an argument; the system takes the item name from the control's `ddeltem` property. When the data item arrives, the system places it into the control's `text` or `caption` property and executes the control's `onChange` event handler.
- The `DdePoke` method sends data to the server. The method uses no argument. The system sends the value from the control's `text` or `caption` property to the item whose name is in the control's `ddeltem` property.
- The `DdeExecute` method sends a command to the server. The `Execute` command transmits a text string (contained in `␣warg` in APL) that the other application can process according to its own syntactic rules. Thus, you can use `DdeExecute` to send a command to another application in whatever macro processing language that application uses. The argument is the command that you want the other application to execute.
`'fmA.edClient' ␣wi 'DdeExecute' 'srvrcmd'`

Using DDE Event Handlers: A Summary

The `onDdeConnect` event handler applies to a Form that is a server and a Label or Edit control that is a client. The system executes this handler when a client establishes a conversation with a server. Note that the `DdeConnect` event applies to the conversation, not to each connection. If you have multiple items on a server form or multiple clients connected to a single server, the system executes the `onDdeConnect` event handler only for the first connection.

The `onDdeDisconnect` event handler applies to a Form that is a server and a Label or Edit control that is a client. The system executes this handler whenever one of the relevant properties for either the client or the server changes, or when the other end of the connection closes. Note that this event handler can be followed immediately by the `onDdeConnect` handler if a property is reset to a different valid value.

The `onDdeExecute` event handler applies to a Form that is a server. The system fires this handler when the remote client application sends a DDE Execute command. `¶warg` contains the text string that is the command.

Using DDE Properties: A Summary

The `ddeTopic` property applies to a Form that is a server and to an Edit or Label control that is a client. On a server you specify only the Topic. For a client, you can specify both the application name and the topic of the server to which you want to connect. Separate the two names with a stipe. You can specify only the topic when you use the `ddeApplication` property.

The `ddeApplication` property applies to a client control. When you specify the application name of the server in this property, the system attempts to open the remote application if it is not running. When you use this property, specify only the topic in `ddeTopic`.

The `ddeltem` property applies to an Edit or Label control that is either a client or an item on a server form. You must specify this property if you want to transmit data; if you want only to send commands from the client to the server using a DDE Execute command, you can omit it.

The `ddeMode` property applies to a client control. You specify this property as `'Hot'`, if you want automatic transmission of data from the server to the client, or `'Cold'` otherwise.

The `ddeTimeout` property applies to a client control. You specify the number of milliseconds the client control waits when trying to establish a connection with a server.

More Complex Examples

This section presents two examples of using DDE with APL. The first example uses APL as a server to receive a text string and return modified data. The second uses APL as a client to request a value from an Excel spreadsheet functioning as a server and to return a modified result to the server.

Following the examples, there is a brief description of some relevant workspaces including the `XLCHART` workspace, which uses low-level calls to set up the DDE connection. It has a complete example of using an Excel spreadsheet to draw graphs using data from APL as their basis.

Using APL as the Server and the Client

The first function in the example below defines APL as a server which receives data sent by a client application using the DDE Execute command. This client can be any application that supports DDE; in the example, the client is another APL session. When the APL server receives the command, an `onDdeExecute`

event handler runs the function `DOIT`, with the text string sent by the `Execute` command in `□warg`. The function decodes or encodes the string with a simple substitution cipher and places the result in an edit control that is defined as a DDE item. The altered text is then transmitted back to the client application through an automatic (hot) DDE link.

In this example, there are two separate transmissions using DDE. The first is the DDE `Execute` command, sent using the `DdeExecute` method. You trigger that action by clicking the `Send` button on the client form. When the server form receives the command, its `onDdeExecute` event handler runs a function; the function places its result in an edit control.

The second transmission is triggered by changing the edit control in the server application; because the applications are connected by a hot link, the new value is transmitted back to the client form as soon as the `text` property of the edit control changes.

There are five functions defined below. These functions are also in the `DEMODOE` workspace. Although you can set up both server and client from the same APL session, you should set them up from different APL sessions to truly demonstrate the DDE connection. The functions are:

- `Server` sets up a form as a DDE server application
- `Client` sets up a form as a DDE client application
- `SendMsg` transmits the DDE `Execute` command
- `DOIT` processes the data string when received
- `Reset` closes all APL objects in the session

Start one APL session and run the `Server` function; start a second APL session and run the `Client` function. If you try to run `Client` first, it will fail. The server must be registered with Windows in order for the client to make the connection. Then type `Ibvyn!` in the Edit control on the Client form, and click the `Send` button. You should see the decoded text in the Label below the Edit control on the client form and in the Edit control on the server form. You can demonstrate the hot link by itself by typing in the edit control on the Server form. Whatever you type is transmitted to the client form.

APL as the Server:

```
[0] Server;sv;sink
[1] A Creates a Server form that receives and processes text
[2]
[3] A Delete an old copy of the form if it exists
[4] 'fmServer' □wi 'Delete'
[5]
[6] A Create a form with ddeTopic set to ServerEx
[7] sv ← 'fmServer' □wi 'New' 'Form' ('where' 0 50 8 30)
[8] sv □wi 'Set' ('caption' 'Server Form') ('border' 1)
[9] sv □wi 'ddeTopic' 'ServerEx'
[10]
[11] A Define the event handler to run a function
[12] sv □wi 'onDdeExecute' 'DOIT'
[13]
[14] A Add an Edit control to display the processed result
[15] sink ← 'fmServer.edItem' □wi 'New' 'Edit' ('where' 4 1 2 25)
[16]
[17] A Set DDE Item name to ServerItem for return link
[18] 'fmServer.edItem' □wi 'ddeItem' 'ServerItem'
```

▽

APL as the Client:

```
[0] Client;cl;sink
[1] A Creates a client form, from which you send a text string
```

```
[2]  A      using the DDE Execute command, and to which the server
[3]  A      sends data via a DDE hot link.
[4]
[5]  A      Delete an old copy of the form if it exists
[6]  'fmClient' □wi 'Delete'
[7]
[8]  A      Create a client form
[9]  cl ← 'fmClient' □wi 'New' 'Form' ('where' 0 0 8 40)
[10] cl □wi 'Set' ('caption' 'Client Form') ('border' 1)
[11]
[12] A      Create an edit control to enter text to be sent to server
[13] A      Note that this control is NOT a direct part of the DDE link
[14] sink ← 'fmClient.edSource' □wi 'New' 'Edit' ('where' 1 1 2 25)
[15]
[16] A      Create a button whose onClick handler sends the message
[17] □wself ← 'fmClient.bnSend' □wi 'New' 'Button'
[18] □wi 'Set' ('caption' 'Send') ('where' 1 33 1.5 5)
[19] □wi 'onClick' 'SendMsg'
[20]
[21] A      Create a label to receive and display the processed result
[22] A      This control is one end of the DDE link. The Topic defines
[23] A      the other application and the name by which it is
[24] A      registered as a server. The Item defines the name of the
[25] A      specific control in the server that contains the data and
[26] A      to which this form is linked. The hot link specifies
[27] A      that the return transmission is automatic.
[28] □wself ← 'fmClient.lReply' □wi 'New' 'Label' ('border' 0)
[29] □wi 'Set' ('where' 4 1 2 25) ('caption' '')
[30] □wi 'ddeTopic' 'APLW\ServerEx'
[31] □wi 'ddeItem' 'ServerItem'
[32] □wi 'ddeMode' 'hot'
```

▽

```

[0] SendMsg;msg;sink
[1] A This function uses the DdeExecute method to send a DDE
[2] A Execute command to a server. Note that although the
[3] A initial text is in the Edit control, it is the Label
[4] A that is the DDE Item. The Label both sends the text
[5] A and receives the result back from the server.
[6] msg ← 'fmClient.edSource' ⌵wi 'text'
[7] 'fmClient.lReply' ⌵wi 'caption' '' A Remove previous text
[8] sink ← 'fmClient.lReply' ⌵wi 'DdeExecute' msg
▽

[0] DOIT;msg;reply;ltr;index;⌵io
[1] A This is an arbitrary function that runs in response to
[2] A the form's onDdeExecute event handler. It reads the
[3] A text string sent from the client application and returns
[4] A an enciphered (or deciphered) text string.
[5]
[6] A Read the input from ⌵warg;initialize the result and ⌵io
[7] msg ← ⌵warg
[8] reply ← ''
[9] ⌵io ← 1
[10]
[11] A Encipher the text by reversing halves of the alphabet
[12] A This portion has nothing to do with DDE; it could perform
[13] A any action useful to the client application.
[14] :for ltr :in ⌵pmsg
[15]     index ← ⌵av ⌵ msg[ltr] A Dyadic iota gives an index value
[16]     :if index ∈ (65+⌵13)
[17]     :orif index ∈ (97+⌵13)
[18]         reply ← reply,⌵av[index+13] A 'a' becomes 'n' etc.
[19]     :elseif index ∈ (78+⌵13)
[20]     :orif index ∈ (110+⌵13)
[21]         reply ← reply,⌵av[index-13] A 'n' becomes 'a' etc.
[22]     :else
[23]         reply ← reply,⌵av[index] A Other characters stay
[24]     :endif
[25] :endfor
[26]
[27] A Place the result in the edit control
[28] 'fmServer.edItem' ⌵wi 'text' reply
▽

[0] Reset
[1] A Delete all created forms
[2] '#' ⌵wi 'Reset'
▽

```

When you finish, remember to delete your forms using Reset.

Linking APL to Excel

The functions below show how you can perform a transformation on text using APL as a client to request a text string from an Excel server and return the text result to Excel. Since Excel automatically registers itself with Windows as a DDE server, you need only to open Excel, type some text in cell R2C1 (or cell A2), and press Return or click on another cell to set the contents. For example, you might type the following text: able was I

Note that the server must be registered before the client attempts to make the connection. After opening Excel, run the `Link2xl` function, which is in the `DEMOS` workspace. This function creates an APL form with an edit control as a client. It assumes that `Sheet1` is open. Click on the `Fetch` button to get the data string from Excel. Then click on the `Send Back` button to transmit the altered text back to the server.

```
[0] Link2xl;xl;sink
[1] A Creates a client form that links to an Excel spreadsheet;
[2] A Fetches the contents of a cell, reverses and returns it.
[3]
[4] A Delete an old copy of the form if it exists
[5] 'fmLink2xl' □wi 'Delete'
[6]
[7] A Create a client form
[8] xl ← 'fmLink2xl' □wi 'New' 'Form' ('where' 0 40 8 40)
[9] xl □wi 'Set' ('caption' 'Client For Excel') ('border' 1)
[10]
[11] A Create an edit control to link to the server
[12] A This time the edit control IS part of the DDE link
[13] □wself ← 'fmLink2xl.edClient' □wi 'New' 'Edit'
[14] □wi ('where' 1 1 2 25)
[15] □wi 'ddeTopic' 'EXCEL\Sheet 1'
[16] □wi 'ddeItem' 'R2C1'
[17] □wi 'ddeMode' 'cold'
[18]
[19] A Create a button to fetch the contents of the Excel cell
[20] □wself ← 'fmLink2xl.bnFetch' □wi 'New' 'Button'
[21] □wi 'Set' ('caption' 'Fetch') ('where' 1 30 1.5 9)
[22] □wi 'onClick' ' ' 'fmLink2xl.edClient' □wi 'DdeRequest' ' '
[23]
[24] A Create a button to send back the contents reversed
[25] □wself ← 'fmLink2xl.bnMirror' □wi 'New' 'Button'
[26] □wi 'Set' ('caption' 'Send Back') ('where' 4 30 1.5 9)
[27] □wi 'onClick' 'SendBack'
```

▽

The `SendBack` function is as follows:

```
[0] SendBack;mirror;sink
[1] A Reverses the text and returns it to Excel
[2] mirror ← Φ'fmLink2xl.edClient' □wi 'text'
[3] 'fmLink2xl.edClient' □wi 'text' 'mirror'
[4] sink ← 'fmLink2xl.edClient' □wi 'DdePoke'
```

▽

DDE Workspaces

The DEMODDE workspace contains the examples from this chapter.

The DEMOWIN workspace includes several functions that illustrate DDE properties, events, and methods. The examples include setting up a server and a client having a conversation in the same workspace; samples of clients and servers interacting with a spreadsheet; and a simple illustration of issuing commands for another application to execute.

The DDEII workspace includes the cover functions that version 5 of APL★PLUS II/386 used to support DDE, converted to work with the Windows interface in APL+Win. This facility allows you to continue to run existing applications that use these cover functions.

The XLCHART workspace contains a demonstration function that allows you to take APL data and draw a variety of charts in Microsoft Excel.

To run this demonstration, load the workspace and follow these steps:

1. Make sure that Microsoft Excel is accessible with your current `DOS PATH=` statement. Alternatively, you can edit the `ExcelChart` function and change the *ApplicationName* statement to point to the Excel executable file, as in this example:

```
ApplicationName ← 'c:\EXCEL\EXCEL'
```

2. Enter the following command '*type*' `ExcelChart data` where *data* is an APL numeric vector and *type* is the type of chart you want to draw. You can specify any of the following types of charts:

AREA, BAR COLUMN, LINE, PIE, SCATTER, RADAR, 3-D AREA, 3-D BAR, 3-D COLUMN,
3-D LINE, 3-D PIE, or 3-D SURFACE.

Optionally, you can specify a gallery number after *type*. See the Microsoft Excel *Users Guide* for information on gallery numbers.

For example, you can run the following:

```
'3-D COLUMN, 6' ExcelChart 3 3ρ 1 2 3 4 5 6 9 8 8
```

APL Windows Utilities Summary

This section briefly describes the purpose of the principal tools in the various Windows-related workspaces and a user command file. This is not an exhaustive list of available functions. Use the `Summary` and `Explain` functions in the workspaces to access the online documentation for these tools. Use `lname?` to see the syntax and to retrieve the documentation for a specific user command.

Windows Demonstration Workspaces

APL+Win includes a number of workspaces that demonstrate features and techniques. You can study the functions in these workspaces and experiment with variations to extend your capabilities. Functions in several of these workspaces are summarized below:

- The function `DemoCalc` in the DEMOCALC workspace displays a form that is a working calculator.
- The function `DemoDraw` in the DEMODRAW workspace displays a form on which the system draws a number of different figures with varying colors and patterns. You can select the type of figure you want the system to draw from the Demo menu. If you click on the form with the left mouse button, the system draws another figure of the same type. You can clear the window by pressing the right mouse button.

The UCMSDW User Command File

The UCMSDW file is a User Command file with tools for working with the Windows interface. These are available to you in any workspace if you used the default setup. If you cannot find these commands, use `lufile path\UCMSDW` to add it to your list of user command files. You can list the syntax and online documentation for each of these functions with `lname ?`. The table below summarizes the main commands in the workspace.

UCMSDW File Commands

Function	Description
<code>lclipcopy</code>	Copies APL data to the clipboard.
<code>lclippaste</code>	Pastes data from the clipboard into APL.
<code>lforms</code>	Reports information on all the active forms in Windows memory, all <code>def</code> variables, and all <code>_Make</code> functions in the workspace.
<code>lhandlers</code>	Reports information on all event-handler code associated with a form.
<code>lwbuid</code>	Imports forms from a migration file made with <code>1WCONVERT</code> in APL★PLUS II/386 APLGUI.
<code>lwed</code>	Forms designer and editor.
<code>lwinloc</code>	Searches Windows files for a given phrase.
<code>lwmake</code>	Builds a <code>_Make</code> function for an active form.
<code>lwtree</code>	Explorer for APL Windows interface objects.

The WINAPI Workspace

The WINAPI workspace is a conversion of the `APLWIN.WS` workspace from APL★PLUS II/386, Version 5.x. It contains the core functions for the low-level Windows API, invoked in APL+Win with `Wcall`. If you are converting (or need to develop) applications at a lower level than `Wwi` provides, use this workspace. The table below summarizes the main functions in the workspace.

WINAPI Workspace Functions and Syntax

Function (with syntax)	Description
<code>res ← type W_C name</code>	Lookup program for Windows constants.
<code>hwnd ← W_Create wspec</code>	Create one or more windows.
<code>fp ← name W_CreateFilter filter</code>	Create a callback filter procedure.
<code>W_Destroy hwnd</code>	Close one or more windows.
<code>W_DestroyFilter fp</code>	Destroy a filter procedure previously created.
<code>msg W_FP expr</code>	Build filter global var for <code>W_CreateFilter</code> .
<code>(hInst hPrev pszCmd nShow) ← W_Init</code>	Return WinMain startup parameters.
<code>data ← [newdata] W_Mem pcc</code>	Read or write Windows memory.
<code>value ← W_Ref id</code>	Retrieve value associated with an id.
<code>W_RegisterClass wc</code>	Register window class specified by a structure.
<code>[hdc] W_Sel obj</code>	Select object into device context and delete displaced object.
<code>value W_Set id</code>	Set a value associated with an id.
<code>fp ← W_SubClass parms</code>	Subclass (or unsubclass) a window.
<code>text ← W_Text parms</code>	Get or set window or dialog box text.
<code>rc ← W_Wait</code>	Equivalent to message loop in C-based Windows.

The WINDOWS Workspace

The WINDOWS workspace comprises utilities that are useful in interacting between APL+Win and Windows. The table below summarizes some of the more important functions in the workspace.

WINDOWS Workspace Functions and Syntax

Function (with syntax)	Description
<i>avchars</i> ← ANSI2AV <i>ansichars</i>	Convert APL text from ANSI to ⎱av encoding.
<i>hndl</i> ← APLFrame	Return handle of main Session Manager window.
<i>res</i> ← APLWindows	Matrix of APL window names, handles, and types.
<i>ansichars</i> ← AV2ANSI <i>avchars</i>	Convert APL text from ⎱av to ANSI encoding.
CLIPCopy <i>data</i>	Copy APL data to the clipboard (CF_TEXT format).
<i>data</i> ← CLIPPaste	Get CF_TEXT data from the Windows clipboard.
<i>dll</i> ← [call] DLLFindEntry <i>entry</i>	Locate the DLL containing a specific Windows call.
<i>ini1</i> INIcomp <i>ini2</i>	Compare 2 Windows .INI (or any [Sectioned]) files.
<i>code</i> ← [title] Msg <i>message</i>	Template function for various message box types.
<i>pos</i> ← [size] [flush] Print <i>text</i>	Send APL text to the printer.
<i>res</i> ← [flags] WChooseColor <i>init</i> [custom]	Use a common dialog box to get a color.
<i>res</i> ←[flags <i>hDC</i>] WChooseFont <i>logfont</i> [color]	Use a common dialog box to get a font.
<i>res</i> ← <i>prntername</i> [flags]	Use a common dialog box to get a font for the printer and display.
⌘ WChoosePrinterFont [logfont] [color]	
<i>res</i> ← [flags] WGetOpenLongFileName <i>specs</i>	Use a common dialog box to get a file to open.
<i>res</i> ← [flags] WGetSaveLongFileName <i>specs</i>	Use a common dialog box to get a file to save.
<i>res</i> ← [value] Wdata <i>id</i>	Set or Reference a value stored in a data property.
Wedit <i>objectname</i>	Edit a function, vector, or matrix.
<i>res</i> ← [type] Whelp [file] [keyword] [handle]	Call the Windows Help facility.
<i>rc</i> ← [sep] Wmsg [msg]	Display a message box.
<i>names</i> ← Wnames <i>form</i>	Get names of all objects that compose a form.

The AV2ANSI and ANSI2AV functions allow you to translate character vectors between the ⎱av atomic vector character encoding and the ANSI sequence used by Windows. These functions use static translate tables that convert the indices of the character positions. If you copy these functions into an application, those tables do not get updated when the atomic vector changes, such as it did most recently with the addition of the Euro symbol. You can accomplish the same tasks with the current translation table of your interpreter by using the intrinsic ⎱wcall function named W_AvAnsi.

If you simply invoke the function, it returns a 2-row, 256-column matrix of characters. But the virtue of the characters is that they represent the indices of the translation tables between the two character encodings. You can use the primitive function Index of (dyadic ⍷) to get the indices of a character vector, and then use the matrix values to retrieve the corresponding characters from the other sequence. The first row of the matrix provides the ⎱av to ANSI translation; the second row provides the ANSI to ⎱av translation.

Note that you need a companion place to display the ANSI characters; simply invoking the translation in the APL+Win session does not demonstrate the functionality. The first example uses a tooltip on a control object in APL+Win.

```
ten←⎱av[130+⍷10]
'fm.ctl' ⎱wi 'tooltip' ten      A without translation; compare to
'fm.ctl' ⎱wi 'tooltip' ((⎱wcall 'W_AvAnsi')[1;⎱av ⍷ ten]) A translated
```

Type characters into WordPad using Alt+0228 through Alt+0237. Copy these characters to the Windows clipboard. Then compare these statements using the CLIPPaste function from the Windows workspace:

```
funny←CLIPPaste ⋄ funny
```



```

ã∨⊥τφθ∧∕∇∅
      ((⊞wcall 'W_AvAnsi')[2;⊞av ⊔ funny])
ääæçèéëìí

```

Unicode Character Support

APL characters have traditionally been stored internally using an eight-bit byte and have used a mapping of characters to numeric values (called an encoding) that is proprietary. With the advent of the Unicode character set, there is now a widely accepted standard that defines a single encoding for just about every character currently in use on computers in any language, including APL characters. However, this encoding requires more numeric values (code points) than can be stored in an eight-bit byte. Allowing APL+Win programs to use this encoding for character arrays is an ongoing effort of the APL+Win development staff, and this section describes the current progress of that project.

A new internal datatype has been implemented to represent characters defined by the Unicode standard. The new datatype uses 32-bit values to store the Unicode code points for the characters. The longer-term goal is to be able to use Unicode characters and eight-bit characters interchangeably, in much the same way as integers and floating point numbers can be used together. This ideal has not yet been reached, and Unicode characters can only be used with a small number of primitives right now. You will get a NONCE ERROR if you pass Unicode characters as an argument to primitives that do not yet recognize the new datatype.

A new system function, `⊞ucs`, has been implemented to create Unicode character arrays; it is described separately. `⊞DR` has been enhanced to use 32 as the datatype for Unicode character arrays. `⊞TYPE` returns a Unicode space for each Unicode character in its argument.

The following structural primitives currently accept Unicode character arrays as arguments: `join` and `ravel` (`,`), `match` (`=`), `reshape` and `shape` (`ρ`), `pick` (`⊃`), `enclose` (`⊂`), `first` (`↑`).

The Windows Interface (`⊞wi`) has been enhanced to allow passing Unicode character arrays to and from BSTR string arguments of ActiveX controls and objects. This is described under the new `unicodebstr` property.

Advanced-to-Expert Programming Techniques

Beyond the techniques introduced in this and the preceding chapters, there are more complicated and sophisticated things you can do with the Windows interface in APL+Win. For some cases, the only limits are your imagination. Specific areas, however, are introduced in succeeding chapters.

- You can use any of the vast variety of ActiveX controls that are available to you commercially. ActiveX controls are products that conform to an interface specification that usually allows any instance of that object to have a visual representation on the screen. APL+Win can create an instance of an ActiveX control that is registered on your system. You can then use that control and its built-in properties and behaviors as though it is an APL+Win object. See the Using ActiveX in APL+Win chapter in this manual for the basic information you need to use ActiveX controls.
- You can also create ActiveX objects that are not controls; these objects also conform to the broad ActiveX interface standard, but they tend to function as server applications that you can control through your APL+Win program. Although servers often remain hidden, you may be able to make an ActiveX server visible and have the APL user work directly with the server's own user interface. Your program can then access the results or the values from the application for further use within APL. Many objects exist in families, where an application calls other objects for its constituent parts. For example, the

Excel application creates ActiveX objects for its charts and worksheets. See the Using ActiveX in APL+Win chapter for information on ActiveX server objects.

- In addition to using ActiveX objects from APL, you can create an instance of APL as an ActiveX local server. This allows you to have an APL+Win application interact with other applications through this widely accepted standard. See the APL+Win as an ActiveX Server chapter in this manual.
- APL+Win provides facilities for calling non-APL routines from APL+Win as if the routines were APL user-defined functions. These routines can be ones you write yourself, Windows system libraries, or other software packaged in a Dynamic Link Library (DLL). You can also use the system function `⌐wcall` as a low-level Windows interface to call into Windows libraries at the same level as a C programmer. See the chapter "Calling Dynamic Link Libraries" for more details.
- APL+Win implements access to communications protocols using the system function `⌐ni`, the network interface. This function is a sockets interface for the APL+Unix, APL+Win and APL64 systems; it implements a set of sockets functions common to BSD sockets and WinSock 1.1. It also adds some platform specific features. In APL+Win, `⌐ni` accepts certain *properties*, *event-handlers*, and *methods*, which are analogous to the actions you use with `⌐wi`, the windows interface function for building a graphical user interface. In particular, the asynchronous notification event handlers allow you to implement a communications application that takes advantage of the underlying event-driven architecture of Windows, and respond with actions at appropriate times without needing to rely on continual polling. See [Help | APL Language | Network Interface Reference](#) for the techniques and the descriptions of the methods, events, and properties of the network interface.
- In addition to the built-in classes and the available ActiveX controls, you can define custom classes in APL64. As a class developer, you can create your own named classes that make use of any valid combination of built-in classes and behaviors, or you can modify the properties and behaviors of existing classes. The fundamental idea is to interrupt the system's normal processing at the point of creating APL64 Windows interface (WI) objects. At that point, you define the behavior and characteristics to create the new class of object or modify the behavior and characteristics of an existing class. When the handler returns, the system resumes its normal processing. This facility gives you an almost unlimited flexibility for innovative expert programming. See the Creating User-Defined Classes chapter in this manual for the basic information you need to define your own objects.

Chapter 8: Using ActiveX in APL64

ActiveX is a term that Microsoft uses to describe a standard practice for interfacing between applications or products that are designed independently. ActiveX objects conform to the Component Object Model (COM) standard. Thus, you may also see the term COM object. APL64 recognizes two subsets of ActiveX objects: ActiveControls and ActiveObjects. ActiveControls are products that conform to a more elaborate interface specification; this usually allows such objects to be created with a visual representation on the screen and to function as though they are children of an APL64 Windows interface (WI) object, such as a form or a frame. In the APL64 documentation, the term "ActiveControl" has replaced the now-obsolete term OCX.

An ActiveObject also conforms to the broad COM standard, but an ActiveObject usually does not have a where property that allows it to be positioned on an APL64 form. Such objects tend to function as server applications that you can control through your APL64 program. Although servers often remain hidden, you may be able to make an ActiveX server visible and have the APL user work directly with the server's own user interface. Your program can then access the results or the values from the application for further use within APL.

Note: The traditional companion term for the server is "client"; the client initiates an action and the server responds to (or serves) the client. Increasingly, the term "controller" is also used to describe the client role with regard to ActiveX automation objects (servers); whether a controller demands service or a client requests it, the electrons are the same.

The terms for ActiveX objects parallel the usage for APL64 WI objects. An instance of any class is an object. Only a subset of the classes, generally those with a visible manifestation, are controls. An instance of any ActiveX class is an object; only a subset of those, which usually require a "window" in the interface, are ActiveControls. The others are designated as ActiveObject classes. In this chapter, they may also be called ActiveX server objects.

You can also access APL as an ActiveX object from other applications. As with other server objects, APL as an ActiveX object is usually not visible. See the chapter "APL+Win as an ActiveX Server" for more details.

Creating an ActiveX Control

ActiveX objects are registered with Windows when they are installed. You can see the ActiveControls registered on your system by referencing the `xclasses` property of the system object. The result is a four-column matrix that gives the descriptive name of the ActiveX control, its globally unique identifier (guid), its programmatic identifier (usually, a three-part name separated by periods), and a version number.

There are a wide variety of ActiveX controls available, from simple controls such as a gauge or a spin button to complex controls such as a graphics interface or a spreadsheet. APL64 treats an ActiveControl like any object class; it adopts its properties, methods, and events. When you query these properties, methods, and events, APL64 displays them with an "x" or "X" prefix. Consistent with APL64 practice, the properties have lowercase "x" while methods and events use uppercase "X". When you want to assign an event handler, you must also add the APL64 prefix "on", which then creates a name like: `onXClick`. However, note that, unlike APL64 WI action names, which are case sensitive, these controls may treat names as case insensitive.

That full name with the prefix may always be used, and when a name is unique the x/X prefix may be omitted. However, you must use the prefix with the names of its properties, methods, and event handlers that conflict with standard APL64 properties, methods, and event handlers.

APL allows you to create an instance of any ActiveControl that is properly registered. You create an instance of an ActiveControl using the `New` or `Create` method, followed by any of the three identifiers from the first three columns of the `xclasses` matrix:

```
'fmNew.axMyTool' □wi 'New' 'descriptive name'  
'fmNew.axMyTool' □wi 'New' '{guid}'  
'fmNew.axMyTool' □wi 'New' 'vendor.server.version'  
'fmNew.axMyTool' □wi 'New' 'OCX progid'  
'fmNew.axMyTool' □wi 'New' 'OCX simple-name'  
'fmNew.axMyTool' □wi 'New' 'non-simple-name'
```

You must include the braces around the guid. Usually, there are three parts, vendor, server and version, that make up the programmatic id. (The version may be omitted or it may have more parts; use exactly what is given.) Once you have an instance of an ActiveX control, the values from the registry are also properties of the control: description, clsid, and progid, respectively. The version number is also contained in the version property. The class property shows the term ActiveControl as well as the value with which you created the control.

A non-simple-name means a name with spaces or other non-standard class name punctuation (such as a dot "."). Simple-names mean a name without any special characters (such as "Grid"). If your ActiveX control has a simple name you must prefix it with "OCX ".

If you are distributing an application and you use an ActiveControl that has a license key, you can use any of the above three statements to create the control on your developer's machine. After you have an instance of the control, reference the class property. It will contain the guid plus the license key. For the runtime version of your application, create the ActiveX using the value from the class property:

```
'fmNew.axMyTool' □wi 'New' '{guid}license_key'  
'fmNew.axMyTool' □wi 'New' '{clsid}license_key'
```

You can also specify a license key following a programmatic identifier by using the percent sign (%) as a separator character. For example (fictitious):

```
'fmNew.axMyTool' □wi 'New' 'Footco.ToeCtrl.1%digits'
```

Properties of an ActiveControl

Every ActiveX custom control defines a unique set of properties that you can use to control its appearance or behavior; the ActiveX documentation will tell you the names of these properties, the allowable values, and their effects. You can use the properties of an ActiveX control in APL64 by prefixing their names with lowercase letter `x`. Note that ActiveX classes conventionally start their properties with an uppercase letter. Therefore, the second letter (and perhaps others) of the property name in APL64 will normally be uppercase. You can determine the names of all properties available for an object by referencing its `properties` property. You can get a short description of a property by preceeding its name with a `?`, or access the help file (if supported and available) by using two question marks.

In addition to the properties defined by the ActiveX control, APL64 automatically adds the following properties to every ActiveControl: `border`, `children`, `class`, `clsid`, `content`, `data`, `def`, `deferexit`, `description`, `edge`, `enabled`, `errorcode`, `errmsg`, `events`, `extent`, `help`, `helpcontext`, `instance`, `interface`, `links`, `methods`, `modified`, `modifystop`, `name`, `noredraw`, `obj`, `opened`, `order`, `pending`, `place`, `pointer`, `progid`, `prompt`, `properties`, `savecontent`, `scale`, `self`, `size`, `state`, `suppress`, `tabgroup`, `tabparent`, `tabstop`, `tooltipwidth`, `version`, `visible`, and `where`.

Some of these properties, for example `savecontent`, are unique to ActiveX controls in APL64; that is, they do not exist on any of the built-in classes. Some properties, for example `visible`, may duplicate similar properties inherent in the ActiveX control. In this case, it is important to use the prefix when you set or reference the controls property; for example: `'fm.MyTool' □wi 'xVisible' 1`

ActiveX Array Properties

Some ActiveX classes have one or more array properties, which allow you to access indexed lists of various sorts for an object. You may not know how many items are in the array; therefore, you cannot reference the object as a whole. You can, however, reference or set any item in the list. APL64 also allows you to set multiple items in the list with one command using strand notation.

APL64 has adopted an empty bracket notation to distinguish ActiveX array properties. If, for example, the ActiveX property of our sample control is named `ColWidth`, its name in APL64 is `xColWidth[]`. You can reference specific items in the list by following the property name with an integer vector or scalar. For a single index notation for one value, you can use the empty brackets `[]`, or you can just name the property without any additional annotation. Thus you can reference a single value with either of these statements where *col* is the number of the column:

```
width ← 'fmNew.axF1v5' □wi 'xColWidth[]' col
width ← 'fmNew.axF1v5' □wi 'xColWidth' col
```

You can set a single value by adding a second argument where *col* is the number of the column, and *size* is the value you want to assign:

```
'fmNew.axF1v5' □wi 'xColWidth[]' col size
'fmNew.axF1v5' □wi 'xColWidth' col size
```

If you want to reference or set multiple values with a single index, you must use the brackets after the name of the property.

```
width ← 'fmNew.axF1v5' □wi 'xColWidth[]' (col1 col2 col3)
'fmNew.axF1v5' □wi 'xColWidth[]' (col1 col2 col3) (size1 size2 size3)
```

You can assign the same value to multiple cells by specifying only a scalar for size; otherwise the number of values must match the number of cells.

Using Multiple Indexes:

Some arrays have multiple index notation, for example, row and column for a grid cell. If you need multiple indices to reference one value, you supply the index values in a vector or a matrix. To reference one value, you would use a vector. To set one value, you would specify a value.

```
cell ← 'fmNew.axF1v5' □wi 'xTextRC[]' (col row)
'fmNew.axF1v5' □wi 'xTextRC[]' (col row) ('12345.678')
```

If you identify multiple values in a matrix, each row must contain one set of index values; the number of columns must match the number of indices. The statements below would return the values from three cells: (1,4) (2,5) and (3,6).

```
x←(1 2 3,[1.5] 4 5 6)
cells ← 'fmNew.axF1v5' □wi 'xTextRC[]' x
```

However, if you specify two vectors of values, the index values are catenated in outer-product fashion. Thus, the following statement would return the values of a square of nine cells: (1,4) (1,5) (1,6) (2,4) (2,5) (2,6) (3,4) (3,5) and (3,6).

```
cells ← 'fmNew.axF1v5' □wi 'xTextRC[]' (1 2 3) (4 5 6)
```

If you wanted to set values in the square, you could follow the arguments by a single value or by nine values. If you want scalar extension to apply to a vector you must enclose it. Thus, the first statement below places one letter in each cell, while the second would place the nine-letter string in each cell.

```
'fmNew.axF1v5' □wi 'xTextRC[]' ((1 2 3) (4 5 6)) ('abcdefghi')
'fmNew.axF1v5' □wi 'xTextRC[]' ((1 2 3) (4 5 6)) (<('abcdefghi'))
```

ActiveControl Persistence

When you close an ActiveControl or create the control on a closed form, APL64 "asks" the control to save or restore its internal state. The content property of an ActiveControl is an integer vector containing the internal state of the properties of the control. What information gets saved into the content property depends entirely upon what the control saves into it. For example, some controls save configuration information such as color and layout, but do not save user data loaded into the control. You can reference the content property at any time. Note that this property, unlike the state property of an APL class, does not contain the event-handler bindings or the APL64 WI properties.

You can use this property as you use the def property of an ordinary control; you can save the value and use it to re-initialize the ActiveX values of your ActiveControl. You can set a new value only with one you have saved from the same class of control. If you try to put arbitrary values into this property, the results are unpredictable and possibly damaging.

A companion property, `savecontent`, is a Boolean flag that determines whether the content property is saved (1, the default) when the control is closed. For a very large and complicated control you may have already saved the contents in another way when closing the control. In this case, you may not want to also save it in the content property; therefore, you would set the `savecontent` flag to zero. The content property is not saved in the state property. Therefore, if you want to save content in a MAKE function, you need to explicitly reference it when you create the control. The content property is saved in the def property. Remember that this facility is dependent on the underlying control supporting the persistence. This may not work for every ActiveControl.

ActiveControl Methods and Events

Every ActiveX custom control defines a standard set of methods you can use to control its appearance or behavior and a unique set of events for which you can write event handlers. The ActiveX documentation that accompanies the control will tell you their names and effects. You can use the methods of an ActiveControl in APL64 by prefixing their names with the uppercase letter X. You can determine the names of all methods available for an object by referencing its `methods` property. You can get a short description of a method by preceding its name with a `?`, or access the help file (if supported and available) by using two question marks.

In addition to the methods defined by the ActiveControl, APL64 automatically adds the following methods to every ActiveX control: `Close`, `Create`, `Defer`, `Delete`, `Event`, `Exec`, `Focus`, `Help`, `Hide`, `Modify`, `New`, `Open`, `Paint`, `Popup`, `Ref`, `Resize`, `Send`, `Set`, `SetLinks`, and `Show`. Be aware that some of these APL64 WI methods may shadow the control's own methods.

You can use the event handlers of an ActiveControl in APL64 by prefixing the names of the events with the letters `onX`. Thus, if the control recognizes the `Click` event, you can write an `onXClick` event handler. You can determine the names of events for an object by referencing its `events` property.

In addition to the event handlers allowed by the ActiveControl, APL64 automatically adds the following event handlers to every ActiveControl: `onClose`, `onDelete`, `onDestroy`, `onExit`, `onExitError`, `onFocus`, `onHelp`, `onHide`, `onModified`, `onMove`, `onOpen`, `onPaint`, `onRopen`, `onResize`, `onSend`, `onShow`, and `onUnfocus`. Be aware that some of these APL64 WI events may shadow the control's own events.

There is a possibly significant difference in programming event handlers for ActiveX objects than for the built-in APL64 event handlers. For ActiveX callbacks, all the parameters from `⎱warg` are contained in `⎱wres`; in addition, there may be one leading element that is the explicit result of the event.

Creating an ActiveX Server Object

ActiveX server objects are registered with Windows just as ActiveControls are. In fact, you may have many more of them on your system. You can see how many ActiveX Controls and server objects are registered on your system by executing the statement:

```
ρ'#' □wi 'XInfo'
```

You can use a character vector argument with the XInfo method to limit the list. Using the following statement returns all the registered ActiveX objects starting with the letters MS.

```
'#' □wi 'XInfo' 'MS'
```

You can also use the asterisk (*) as a wild card symbol in the argument to XInfo. If you made the argument to XInfo in the above statement '★MS', the statement would return all the registered ActiveX objects that contain MS anywhere in their XInfo listing (case insensitive, in any column). The result of XInfo is a three-column matrix that gives the programmatic identifier of the object, whether it is an ActiveControl or an ActiveObject, and its descriptive name. One entry in the matrix may refer to another registry entry; usually the referenced entry will have a similar descriptive name with a different suffix. This technique of registering ActiveX objects allows manufacturers to update their products and have you call the most recent version without requiring you to change your code.

Many objects exist in families, where an application calls objects for its constituent parts. You can see such relationships if you have, for example, Excel, installed on your system. The Excel application actually creates ActiveX objects for its charts and worksheets.

You create an ActiveObject the same way you create an ActiveControl, using the New or Create method followed by either the programmatic identifier or the descriptive name. You can also use the ActiveObject's guid if you know what it is. You may also need to specify a license key following the guid or programmatic identifier as described above for the ActiveControl.

ActiveObject Properties, Methods, and Events

ActiveObjects also have their own properties, methods, and events. As with ActiveControls, APL64 prefixes the properties with a lowercase x and the methods and events with an uppercase X. Similarly, APL64 adds a subset of the built-in APL64 WI properties, methods, and events, to the object.

APL64 automatically adds the following properties to every ActiveObject: children, class, clsid, data, def, description, errorcode, errormessage, events, instance, interface, links, methods, modified, modifystop, name, obj, opened, pending, progid, properties, self, state, suppress, and version.

APL64 automatically adds the following methods to every ActiveObject: Close, Create, Defer, Delete, EnumEnd, EnumNext, EnumStart, Event, Exec, Modify, New, Open, Ref, Send, and Set.

APL64 automatically adds the following event handlers to every ActiveObject: onClose, onDelete, onModified, onOpen, onReopen, and onSend.

Using ActiveObjects and ActiveControls

Generally, you will know something about an ActiveX object you create and what you intend to do with it. You can experiment with such objects using their features and the facilities that APL provides.

APL Facilities that Apply to ActiveX Objects

As mentioned above, the `description`, `clsid`, and `progid` properties are character vectors that contain information about the ActiveX object; the descriptive name, `guid`, and programmatic identifier respectively. The `class` property is a character vector that shows either `ActiveControl` or `ActiveObject`, sometimes followed by one of the identifiers. If you create an `ActiveObject` from an object value, APL64 knows only that it is an object, but does not know the values of any of the identifiers. The `version` property shows the version of an `ActiveControl`; it may be empty for an `ActiveObject`.

The `interface` property is an integer that represents a pointer to the object. You can use the pointer for low-level calls to the object. Note however, that this pointer is transitory; it may be valid only for the duration of a callback. Thus, you should reference and use it in the same function; do not store it in a global variable and try to use it during a different callback. The `obj` property also returns an interface pointer, but this pointer is "typed" (see below) as an object. In some cases, you can use this pointer directly as an argument to another ActiveX action where you would have to use the `VT` function to "wrap" the normal integer returned by the `interface` property.

APL64 must keep track of these pointers in order to release the underlying Windows resources when appropriate. Thus, each pointer is invalidated when a callback returns. If you acquire a pointer to an object through a low-level call, you can use the system object's `UseObject` method to establish and retain the validity of a pointer for the duration of the current callback. You can use the system object's `ReleaseObjects` method to free all the validated pointers you have acquired, from any call that returns an ActiveX pointer to APL, during a long-running callback when your program no longer needs the underlying resources. `ReleaseObjects` does not take an argument; you cannot apply it selectively.

ActiveX Named Constants

In addition to the properties, events, and methods that are part of an ActiveX object, the object may have a number of built-in named constants. This is typical of C programming, and you may find many constants listed by name (but not value) in ActiveX documentation. In APL64, you can query the value of a constant, just as you can a property, by prefixing its name with a lowercase `x`. For example, the DAO database engine (see the example later in this chapter) has a constant named `"dbLangGeneral"`. You can query the value of this character constant, which specifies a format, by using the following statements:

```
'dao' □wi 'New' 'dao.dbEngine.35'
dao
      'dao' □wi 'xdbLangGeneral'
;LANGID=0x0409;CP=1252;COUNTRY=0
```

The named constants `"True"` and `"False"` are special values in APL64; you can use them with any ActiveX object to represent 1 and 0. If you use Excel, most of the constant names start with the letter `x`, so you must prefix an additional `x`. To find the value of the `xlBitmap` constant, which identifies the format of a picture:

```
'wb' □wi 'New' 'Excel.Sheet'
wb
      'wb' □wi 'xxlBitmap'
2
```


You can also use an equal sign in front of the constant name, INSTEAD OF the letter x, to query the value of a constant. Usually you will get the same result, but the difference is that the system looks first in the type library's ITypeInfo interface rather than in the object's ITypeInfo interface.

```
'wb' □wi '=xlBitmap'
```

2

ActiveX Help

To help you use your ActiveX controls, APL displays available help from the control itself. If you type a question mark followed by the name of any property, method, or event, you see a summary help string and description of the required arguments.

```
'fmMain.axBigGrid' □wi '?xCellType'
xCellType : Sets or returns the type of data displayed in a cell,
            a column, a row, a block of cells, or the entire spreadsheet.
Ref: INT
Set: INT
```

If you type a double question mark, followed by the name of any property, method, or event, the system invokes whatever Windows help is available for the queried action.

Error Properties on ActiveObjects and ActiveObjects

Each ActiveX class has an `errorCode` property and an `errorMessage` property. The system sets these read-only properties when an error occurs if the creator of the ActiveX object has defined error codes and error messages. They are reset to 0 and '' when the next successful call is made to that instance of the object.

ActiveX Typed Values

Many ActiveX objects use values that are typed; that is, a variable can hold a value that is tagged to represent a special interpretation of the raw value. For example, one special type is currency. The raw value might be 10.33; in the ActiveX Object, the displayed value is \$10.33. Although these typed values do not exist directly in APL, the system can create tagged values that correspond to the types.

APL64 has adopted the same values as those used by Visual Basic for purposes of compatibility. Each type has two associated codes; one for a scalar value of the type and the other for an array of the type.

Value Types for ActiveX Objects

VB Type Name	Scalar Type	Array Type
Empty	0	
Null	1	
Integer	2	8194
Long	3	8195
Single	4	8196
Double	5	8197
Currency	6	8198
Date	7	8199
String	8	8200
Object	9	8201
Error	10	8202
Boolean	11	8203

VB Type Name	Scalar Type	Array Type
Variant	12	8204
DataObject	13	8205
Byte	17	8209

Note 1: Unlike APL, you cannot have arrays of empty or null values.

Note 2: Variant type, codes 12 and 8204 are not types of actual values. An array of variant type is similar to the APL heterogeneous array. It can contain values of other types mixed together. The scalar variant type is a place holder for a value that may take any one of several types. You cannot have an actual scalar value of variant type 12, but you can have a variant array type 8204.

Note 3: When the type code is one of the object types, scalar values 9 or 13, the corresponding value is a pointer to the object.

Using Typed Values in APL

The system object has two methods for reading and creating typed values. You use the VT (Variant Type) method to query the type of an object; for instance, a cell or a range of cells in a spreadsheet. You can also use the VT method to create a typed value in APL, which you can then use to put into an object such as the cell of a spreadsheet. You can use the VV (Variant Value) method to query the value of an object that contains typed values. However, you do not need to use typed values for the most common forms of data. Character strings are automatically transferred between APL64 and the ActiveX object without programmer intervention. If queried, the object shows the "string" type. Similarly, integers and floating point numbers are automatically transferred between APL64 and the ActiveX object. Floating point numbers are the "double" type and integers are "long" type, but some ActiveX objects treat integers as "double." Depending on the ActiveX object, you may not be able to specify "integer" or "single" typed values. If such values exist in the object, you will see the codes when you query the type.

If you query a range of values from an ActiveX object, you will see an array type code. However, if you want to specify one of the formatted values, such as "currency" or "date," the object may require you to create an array of individually typed values. In addition to the APL64 WI methods on the system object, there are two cover functions, named VT and VV, that you can use in lieu of making an explicit `⌵wi` statement. You can find these functions in the VT_VV workspace in the `Tools` subdirectory where you installed APL.

You should also note formatting information is persistent in some ActiveX objects, such as Excel. Thus, once you set one or more cells, for example, to currency type, the application remembers that setting. If you reassign numerical values, the visual representation will still be currency. You must clear the cells, using the `xClear` method (in Excel) or reset the `xNumberFormat` property to the default setting (in Excel, 'General'). This behavior is not inherent in using typed values; it represents the vendor's idea of how the object should work.

Alternatively, to reset an Excel cell to its pristine state, you can use the APL property `empty`. This property of the system object is a typed value. Similarly, the system object has a property you can use when you wish to leave out an optional ActiveX argument. This property, also a typed value, is named `missing`. You can reference `missing`, assign the returned value to a variable, and use that variable in place of an argument.

```
vacuum ← '#' ⌵wi 'missing'  
'object' ⌵wi 'xMethod' vacuum
```

Creating a Hierarchy of Objects and the Redirection Syntax

Many ActiveX servers are designed as a hierarchy of objects. That is, the application itself is an ActiveX object, and it may contain multiple ActiveX sub-objects down through many levels. You can create each of these objects in APL to use the application. Note that "Child" is a property of the app object, and "Item" is a property of the offspring object.

```
'app' □wi 'New' '{guid}'  
A Using the Guid allows you to use runtime components  
'offspring' □wi 'Create' ('app' □wi 'xChild')  
'thing' □wi 'Create' ('offspring' □wi 'xItem' 1)
```

Rather than nesting □wi statements, you can use a redirection syntax, similar to that used in DOS or Unix shell commands, to create new objects from objects. If you follow the property or method action name with a greater than sign (>), you can assign the APL64 WI name within the single statement. The following statements are equivalent to those above.

```
'app' □wi 'New' '{guid}'  
'app' □wi 'xChild > offspring'  
'offspring' □wi 'xItem > thing' 1
```

It is not always obvious what objects do in a server. You must study the application's documentation. In Excel, for example, a Worksheets object is a collection (or container) that may contain one or more Worksheet objects, which are individual worksheets. A worksheet cell is also an object, as is a range of cells.

An Example Using the Excel Spreadsheet

The following example shows how to set up a single Excel spreadsheet and assign some values. This example merely touches the surface of how you could use Excel as a server, but it may be enough to get you started doing something useful. The interspersed comments are helpful hints, not a thorough explication.

You should also be aware that Microsoft's Component Object Model is an ever-changing "standard." Their own implementations change from version to version. What works in Excel97 may not work in Excel2000.

You can create the initial server object by using an entry found in the registry.

```
'xl' □wi 'New' 'Excel.Application'  
xl
```

If you want to see the results of your efforts, you can make it visible.

```
'xl' □wi 'xVisible' 1
```

In order to progress, you must know that the application consists of a cascade of objects. There is a container (known as a "collection" in standard ActiveX terms) named Workbooks that contains Items, each of which is a Workbook. The Microsoft help file says: "In Microsoft Excel, a workbook is the file in which you work and store your data. Because each workbook can contain many sheets, you can organize various kinds of related information in a single file." You create an instance of the collection using the xWorkbooks property, and you can create one item in the container using the XAdd method. Some containers (see below) automatically populate themselves with one or more items; you can add different types of items to a Workbook by changing the argument to XAdd. By default, this method adds one Workbook.

```
'xl' □wi 'xWorkbooks > xl.wbs'  
'xl.wbs' □wi 'XAdd > xl.wbs.wb'
```

Note that it is not necessary to make the Workbook object a child of the Workbooks object in APL64. It is useful to do so both for understanding and to make it easier to delete objects.

Analogously, a Workbook contains a collection named Worksheets that contains Items, each of which is a Worksheet. A workbook has a property named "xWorksheets". You can use this property to create the container object. A collection has items that can be referred to by their numbers or their names; this object creates one or more items automatically when you create the container. Note that you can use the redirection syntax even when there is an argument to the Item property (the number 1).

```
'xl.wbs.wb' ⌐wi 'xWorksheets > xl.wbs.wb.wss'
'xl.wbs.wb.wss' ⌐wi 'xCount'
3
'xl.wbs.wb.wss' ⌐wi 'xItem > .ws' 1
```

Now that you have connected to an actual sheet, you can work in Excel. You can identify a range of cells and treat them as a single object. You can address individual cells within or outside the range as well.

```
'xl.wbs.wb.wss.ws' ⌐wi 'xRange > rng' 'b2' 'e3'
data ← 4 2 ρ 11 12 21 22 31 32 41 42
'rng' ⌐wi 'xValue' data
```

The matrix is oriented unexpectedly, because Excel, unlike APL, specifies its ranges in column major order. Thus, what looks like a two-row, four-column matrix in Excel is a four-row, two-column matrix in APL.

You can retrieve data from a range like this:

```
values ← 'rng' ⌐wi 'xValue'
values
11 12
21 22
31 32
41 42
```

The following statements load data into an Excel worksheet using typed values:

```
int←'#' ⌐wi 'VT' 43 3
flt←'#' ⌐wi 'VT' 1.7 5
string ← '#' ⌐wi 'VT' 'hello' 8
buck← '#' ⌐wi 'VT' 1.25 6
date← '#' ⌐wi 'VT' 23456 7
null← '#' ⌐wi 'VT' 0 1
one← '#' ⌐wi 'VT' 1 11
zero←'#' ⌐wi 'VT' 0 11
matrix← 4 2ρ int flt string buck date null one zero
'rng' ⌐wi 'xValue' matrix
```

In Excel, the result looks like the following:

	B	C	D	E
2	43	hello	3/20/1964	TRUE
3	1.7	\$1.25		FALSE

In APL, the result looks like this:

```
values ← 'rng' ⌐wi 'xValue'
values
43 1.70
hello 1.25
23456
1 0.00
```

Note that the APL representation of the date 3/20/1964 from cell D2 is 23456. This is the value used by Excel to store the date. However, there is nothing in this APL value to tell us that it represents a date. It is possible to display the typed or tagged values that an ActiveX object returns in its result. This can be done using the at sign (@) as a suffix following a method or property name; for example:

```
types ← 'rng' ⍵wi 'xValue @'
types
      43      1.7
      hello   VT 6 1.25
VT 7 23456    VT 0 0
VT 11 1       VT 11 0
```

As mentioned earlier, integers, floating point values, and character strings can be passed between APL and an ActiveX object without typing. These values may not be treated as typed values when returned to APL. The offset of three values in the display above results from those values not being nested with their type information as the other values are. You can still determine the type using the VT or VV method.

You can put text directly into the spreadsheet:

```
msg ← 'You' 'can' 'put' 'text' 'right' 'into' 'these' 'cells'
      'rng' ⍵wi 'xValue' (⊞2 4ρmsg)
```

	B	C	D	E
2	You	can	put	text
3	right	into	these	cells

```
'xl' ⍵wi 'Delete'
'rng' ⍵wi 'Delete'
```

Special APL Syntax Rules for ActiveX Objects

Not all ActiveX objects are constructed according to the same rules. ActiveX objects that are component parts of a larger ActiveX server do not necessarily have to be named; when they are named, they do not necessarily have to be hierarchical. This leads to some special cases when ActiveX objects are used by APL64.

Specifying the Difference in Setting and Referencing for ActiveX Objects

Certain ActiveX objects do not support type library information. (Which means, among other things, that you will see none of the built-in methods or properties when you query the object with the `⍵wi` statement.) When this is the case, it is necessary to use the greater than (`>`) or less than (`<`) signs in an ActiveX invocation to clarify what kind of invocation you want to make. Under these circumstances, you can use the greater than sign when you are invoking a method or referencing a property, and you can use the less than sign when you are setting a property.

If you use the greater than sign but you do not follow it with a name, this signals the system that you are referencing a property or invoking a method. If you want to set a property, you must specify the less than sign follow it with a value or an argument and a value. For example:

```
'prefs' ⍵wi 'LongPref <' 27 999
```

sets the 27th instance of LongPref to the value 999. In Visual Basic, the equivalent statement would be:

```
prefs.LongPref(27) = 999
```

Enumerating Collections in ActiveX Objects

Three new methods allow you to enumerate collections in ActiveX Objects that don't have a built-in property that allows you to access members of the collection. If your collection has an `Item` property and a `Count` property (that show up as `xItem` and `xCount` in the properties list), then you probably don't need this facility.

Using the EnumStart Method

This method requires no argument: `'obj' ⌵wi 'EnumStart'`

This method begins enumerating a collection from the start. You must call this function before you can use `EnumNext`. The result is zero on a failure and a non-zero interface pointer on success. You do not need to capture this pointer unless you want to access the collection using `⌵wcall`, but you do need to use the method any time the collection has been refreshed.

Using the EnumNext Method

This method allows a non-zero argument; if you do not supply an argument, it defaults to 1:

```
interface ← 'obj' ⌵wi 'EnumNext' n
```

This method iterates over the next `n` or fewer items from a collection and returns a vector of interface pointers, which you can use to create APL objects that represent the items of the collection. Then you can, for example, reference the `name` property of each item to identify it. When the method has enumerated all items in the collection, it does an implicit `EnumEnd`, and returns an empty result until you call `EnumStart` again.

Using the EnumEnd Method

This method requires no argument: `'obj' ⌵wi 'EnumEnd'`

This method ends the enumeration and frees the internally cached interface that was returned by the `EnumStart` method. In general, you do not need to invoke this explicitly, because the system invokes it implicitly when you reach the end of enumerating a collection or when you release the collection.

An Example Function

The function below enumerates the tables and fields in tables in a Visual Basic or Access database file (.mdb file) found at the path which is the argument to the function. System tables from Access have the prefix `MSys`.

There are three steps to creating the collection: 1) the function creates an instance, named `dao`, of the DAO DB Engine ActiveX Object; 2) it creates a connection to the database, named `dao.db`, using the `xOpenDatabase` property; and 3) it creates a `TableDefs` collections object, named `dao.db.tableDefs`. All three of these statements use the redirection syntax (the greater than sign ">") to create the child object. However, note that you cannot use the redirection syntax with `EnumNext`:

```

▽ mdbtables dbpath;j;k;tobj;fobj;fdefs
[1] :IF 0=ρ,dbpath
[2] dbpath←'c:\Program Files\Microsoft Visual Basic\report\empdata.mdb'
[3] :ENDIF
[4]
[5] 0 0ρ'dao' ρwi 'Create' 'DAO.DBEngine.35'
[6] 'dao' ρwi 'xOpenDatabase > dao.db' dbpath
[7] 'dao.db' ρwi 'xTableDefs > dao.db.tdefs'
[8]
[9] A enumerate the tables within the TableDefs collection
[10] j←0
[11] interface←'dao.db.tdefs' ρwi 'EnumStart' A position at beginning
[12] :IF 0=interface
[13] '***failed to establish a new interface on:',dao.db.tdefs,' ***'
[14] :ENDIF
[15]
[16] :WHILE 1=ρitem←'dao.db.tdefs' ρwi 'EnumNext'
[17] r←(tobj←'dao.db.tdefs','tbl',⌈j) ρwi 'Create' (⌈FIRST item)
[18] ρ←'Table: ',r,' Pos: ',(⌈j),' Name: ',⌈tobj ρwi 'xName'
[19] j←j+1 A increment table position number
[20]
[21] A enumerate fields within Fields Collection of this table object
[22] A first create Fields collection
[23] tobj ρwi 'xFields > ',fdefs←tobj,'.fdefs'
[24] k←0
[25] interface←fdefs ρwi 'EnumStart'
[26] :IF 0=interface
[27] '***failed to establish a new interface on: ',fdefs,' ***'
[28] :ENDIF
[29]
[30] :WHILE 1=ρitem←fdefs ρwi 'EnumNext'
[31] r←(fobj←fdefs,'fobj',⌈k) ρwi 'Create' (⌈FIRST item)
[32] ρ←'--Field:',r,' Pos: ',(⌈k),' Name:',⌈fobj ρwi 'xName'
[33] k←k+1 A increment field position number
[34] fobj ρwi 'Delete' A delete field object
[35] :ENDWHILE
[36]
[37] fdefs ρwi 'EnumEnd' A remove internal enum collection object
[38] tobj ρwi 'Delete' A delete table object
[39] :ENDWHILE
[40]
[41] 'dao.db.tdefs' ρwi 'EnumEnd'
[42] 'dao' ρwi 'Delete' A delete top object and all children
▽

```

The result of running this function is shown below:

```

mdbtables 'd:\Excel\db1.mdb'
Table: dao.db.tdefs.tbl0 Pos: 0 Name: MSysACEs
Table: dao.db.tdefs.tbl1 Pos: 1 Name: MSysModules
Table: dao.db.tdefs.tbl2 Pos: 2 Name: MSysModules2
--Field:dao.db.tdefs.tbl2.fdefs.fobj0 Pos: 0 Name:Flags
--Field:dao.db.tdefs.tbl2.fdefs.fobj1 Pos: 1 Name:Form
--Field:dao.db.tdefs.tbl2.fdefs.fobj2 Pos: 2 Name:Module
--Field:dao.db.tdefs.tbl2.fdefs.fobj3 Pos: 3 Name:Name
--Field:dao.db.tdefs.tbl2.fdefs.fobj4 Pos: 4 Name:ReplicationVersion
--Field:dao.db.tdefs.tbl2.fdefs.fobj5 Pos: 5 Name:Type
--Field:dao.db.tdefs.tbl2.fdefs.fobj6 Pos: 6 Name:TypeInfo
--Field:dao.db.tdefs.tbl2.fdefs.fobj7 Pos: 7 Name:Version
Table: dao.db.tdefs.tbl3 Pos: 3 Name: MSysObjects
Table: dao.db.tdefs.tbl4 Pos: 4 Name: MSysQueries
Table: dao.db.tdefs.tbl5 Pos: 5 Name: MSysRelationships

```

Chaining Actions and Specifying Parameters

In order to use the hierarchy of objects in complicated ActiveX servers, and to accommodate some of the syntax you may find in examples of their documentation, APL64 provides some alternate syntax possibilities for use with these objects. (The statements in this section are a shortening of the sequential steps in the example above that use redirection.) To see the examples, invoke Excel and make it visible:

```
x1 'x1' □wi 'Create' 'Excel.Application'
x1 'x1' □wi 'xVisible' 1
```

Chaining Actions

Instead of using multiple steps or calling □wi recursively in the same statement, you can chain a series of actions separated by periods as the right argument to □wi. Each action except the last must result in an ActiveX object, either one you create or one you identify. The first action is taken on the named object identified (either explicitly or implicitly) as □wself. Each subsequent action is taken on the object resulting from its immediate predecessor action.

The first statement below demonstrates two steps explicitly by using two invocations of □wi. The first invocation, inside the parentheses, creates a Workbooks collection, which becomes the left argument to the second, which adds a single Workbook. The subsequent statement is equivalent, doing the same thing by chaining actions.

```
A ('x1' □wi 'xWorkbooks') □wi 'XAdd'
  'x1' □wi 'xWorkbooks.XAdd'
1685900
```

For purposes of this ActiveX discussion, an action can be either a property or a method. While it may not be obvious that referencing a property is an "action," simply invoking the property name can have the effect of creating (or identifying) an object, in this example, a Workbooks collection. The method that follows is an action upon that collection, but the new object, the one being added, is a single member of that collection, a Workbook. When you are chaining actions, there can be any number (so long as there is a meaningful sequence) of properties and/or methods, but they must be properties or methods of the ActiveX objects themselves, not the APL64 properties or methods that are added to ActiveX objects.

As mentioned earlier, Microsoft built the Excel application as a cascade of ActiveX objects. You can identify an object in various ways; most collections (of which Workbooks is an example) have a property named Item that identifies an instance of a member of the collection (a single Workbook in this case). Each instance has a name, not surprisingly contained in a property called Name. These three statements are equivalent:

```
'x1' □wi 'xWorkbooks.xItem(1).xName'
Book1 'x1' □wi 'xWorkbooks(1).xName'
Book1 'x1' □wi 'xWorkbooks().xName' 1
Book1
```

Specifying Parameter Values

Although a range of cells is really a part of a Worksheet, which is an item of a Worksheets collection object, which in turn is part of the Workbook item, you can identify a range as a property of the application. You can reference or set the values of a range using its Value2 property. The next three statements show equivalent ways of identifying a range, by nesting □wi calls, by specifying the range with the property name, and by specifying the range parameters as a separate argument.

```
('x1' □wi 'xRange' 'A1:C4') □wi 'xValue2' (⊘4 3⊘12)
```



```
'xl' □wi 'xRange("A1:C4").xValue2' (⊔4 3 ρ10+⌈12)
'xl' □wi 'xRange().xValue2' "A1:C4" (⊔4 3ρ100+⌈12)
```

In some cases, you can use parameters with the property or method and associate the arguments with them. Here are two examples of saving the workbook. In the first, you save with the name `XXX.XLS` and add the name to the "most recently used" files list. In the second, you save with the name `FOO.XLS`, but do not add the name to the MRU list. Note that when you specify the names and the values, you can specify them in any order. (As a reminder, the prefatory "x" is not necessary unless the name would conflict with a built-in APL WI action name.)

```
'xl' □wi 'DisplayAlerts' 0
'xl' □wi 'ActiveWorkbook.SaveAs(FileName,AddToMru)' 'xxx.xls' 1
1
'xl' □wi 'ActiveWorkbook>.wkbk'
'xl.wkbk' □wi 'SaveAs(AddtoMru,FileName)' 0 'foo.xls'
1
```

When you are specifying a parameter within parentheses, you can also use the "property=value" notation. In this case, *property* is the name of the parameter to be assigned, and *value* can be any of a numeric value, a named constant, or a string. For example:

```
'xl' □wi 'ActiveWorkbook.SaveAs(FileName="xxx.xls",AddToMru=True)'
```

As an alternative to the equal sign as a separator between the name and the value, you can use a colon (name:number); or a colon and equal sign (name:=string). Note that the colon used in separating a property name from the value is a different kind of separator than in the earlier example of specifying the range, where it is distinguishing the parts of a two-part value. You can mix inline and separate argument values in the same statement; for example,

```
'xl' □wi 'ActiveWorkbook.SaveAs(FileName,AddToMru:True)' 'xxx.xls'
'xl' □wi 'ActiveWorkbook.SaveAs(FileName:="xxx.xls",AddToMru)' 1
```

In the last example, you must specify the second value as a number; a named constant can be used only as an inline parameter value. However, you can use named constants without the name prefix, so long as you use one of the separator characters before the constant name. When the action has a default sequence of arguments, you can simply supply the values, in the established order, without specifying the names at all.

In Microsoft Word, the application has a Documents collection that can contain one or more documents. Having opened an existing document, the first statement below defines a range by specifying the starting character and the number of characters in the range (as in an Edit control selection property). The second statement moves the end of the range by specifying what units to use and how many to move. In this case, the units are by word and the repetition value is 4. The final statement turns the range into a selection. (These two statements are equivalent to holding down the Ctrl and Shift keys and pressing the right arrow four times if you were the user working in the application).

```
'word.docs.doc' □wi 'XRange >word.docs.doc.range' 2 4
'word.docs.doc.range' □wi 'XMoveEnd(=wdWord,4)'
4 2 4
'word.docs.doc.range' □wi 'XSelect'
```

The next two statements show alternate forms of the middle statement above. You could also code the last example without the parentheses, simply supplying two arguments to the `XMoveEnd` method; however the empty placeholder notation can be useful when working with chained actions that accept parameters.

```
'word.document.range' □wi 'XMove(=wdWord,)' 4
'word.document.range' □wi 'XMove(,)' 2 4
```

Chaining and Specifying Parameter Values

You can also use the "property=value" notation in chaining actions; in these cases, you must use the equal sign as the separator between the property name and its value. (These examples use the Excel range.)

```
'x1' ⍋wi 'xRange("A1:C4").xColumnWidth=10.5'
'x1' ⍋wi 'xRange("D1:E2").xWrapText=True'
'x1' ⍋wi 'xWorksheets(1).xName="Examples"'
```

Chaining with Multiple Names on Both Sides of ⍋wi

When you have multiple names in both the left and right arguments to ⍋wi, there are more possibilities with ActiveX objects than with the built-in APL64 classes. The differences hinge on the nature of the first name in the right argument (whether it is an object name or an action, such as a property or method) and whether there is a leading period. Note that it is the first *name* in the right argument; that is, whether it is established as the APL64 name of an object. There is a difference between the 'xItem' property of a collection that represents the current "Item" of a collection, which *item* is, in fact, an ActiveX object, and the name 'item' which can be the APL WI name of an object, which may or may not be an item of a collection.

Consider the following statements, where *item* may have varying meanings; the action of the interpreter is explained in the three conditional steps below each statement.

With leading period in the right argument	Without leading period in right argument
'app.collection' ⍋wi '.item.action'	'app.collection' ⍋wi 'item.action'
Search for object named <i>app.collection.item</i> ; apply <i>action</i> .	Search for object named <i>app.item</i> ; apply <i>action</i> (note 1).
If <i>app.collection.item</i> is not defined, see if <i>action</i> is an "existence" type (note 2); if so, apply <i>action</i> on <i>app.collection.item</i> treating <i>item</i> as a pending object name.	If <i>app.item</i> is not defined, see if <i>action</i> is an "existence" type (note 2); if so, apply <i>action</i> on <i>app.item</i> treating <i>item</i> as a pending object name.
Otherwise, see if <i>app.collection</i> is an ActiveX object; apply <i>item</i> as an action on <i>app.collection</i> , followed by <i>action</i> on the result.	Otherwise, see if <i>app.collection</i> is an ActiveX object; apply <i>item</i> as an action on <i>app.collection</i> , followed by <i>action</i> on the result.

Note 1. If you do not recognize that an APL64 statement of this form, where *item* is an object name, applies to the object 'app.item', review the "Using Relative Object Naming" section in the Programming the Windows Interface: Introduction chapter in this manual.

Note 2. An "existence" action is one of the built-in methods, New, Create, or Delete, or built-in properties, def or self. These actions do not signal an error if the object to which they are applied does not exist.

Note 3. When you are chaining actions on an ActiveX object, you may not use multiple consecutive periods in the right argument.

Note 4. You may supply more than two actions in the right argument, or you may supply multiple object names in a parentage chain followed by one or more actions. However, each action prior to the last must return an ActiveX object as a result; that is, it must either create or identify an ActiveX object upon which the next action is valid.

Passing ActiveX Objects as Arguments to Other ActiveX Methods

You can also pass ActiveX objects as arguments to other ActiveX methods in a more convenient and efficient manner. In the past, passing an ActiveX object as an argument to another ActiveX method required using a

second, separate invocation of (`⊞wi 'obj'`) on each argument object. For example, the last line of code below passes the 'xsl' object as an argument to the 'transformNode' method:

```
'xml' ⊞wi 'Create' 'Msxml2.DOMDocument'
'xml' ⊞wi 'Xload' 'C:\mydoc.xml'
'xsl' ⊞wi 'Create' 'Msxml2.DOMDocument'
'xsl' ⊞wi 'Xload' 'C:\stylesheet.xslt'
'xml' ⊞wi 'XtransformNode' ('xsl' ⊞wi 'obj')
```

It is now possible to pass the 'xsl' object directly as an argument using the following alternative notation:

```
'xml' ⊞wi 'XtransformNode(<xsl)'
```

This notation can also be used with named parameters such as this:

```
'xml' ⊞wi 'XtransformNode(stylesheet:<xsl)'
```

The < notation tells `⊞wi` that the name following it is another ActiveX object that should be directly passed as an argument.

The < notation also has the side effect of telling `⊞wi` that the argument should be "passed by value" rather than "passed by reference" (`⊞wi` normally passes object arguments by reference including the VT_BYREF internal flag for the argument). This is important for working around implementation limitations of some ActiveX objects that only accept object argument "by value" and fail when passed their arguments "by reference". Most ActiveX objects understand "by reference" and "by value" arguments equally well, but a few objects that use non-standard implementations do not. This enhancement provides a work around for the limitations of such objects. The < notation can be used without an object name immediately following it to denote only the "pass by value" attribute while using the traditional notation for accessing the object's pointer. In other words, you can also code the previous two lines like this using traditional `⊞wi 'obj'` notation:

```
'xml' ⊞wi 'XtransformNode(<)' ('xsl' ⊞wi 'obj')
```

or:

```
'xml' ⊞wi 'XtransformNode(stylesheet:<)' ('xsl' ⊞wi 'obj')
```

In this case, the only affect of < notation is to denote "pass by value" argument semantics. However, in cases where the object being passed as an argument is already bound to a named ActiveX object, it is more convenient and efficient to use the object name following the < prefix. The traditional capability of passing a object argument is retained for cases where "pass by value" semantics are required but an unnamed dynamic result of some other method is being passed as the argument.

Another side effect of using < notation is that it suppresses returning a copy of the argument's pointer in the result of `⊞wi`. The examples above that use < notation only return the transformed XML as their result whereas the first example that does not use < notation returns both the transformed XML and a reference to the object's pointer in its result.

Chapter 9: Creating User-Defined Classes

Creating a User-Defined Class

In addition to the built-in classes and the available ActiveX controls, you can define custom classes in APL+Win using the `onNew` and `onAction` event handlers, the `Event` and `Defer` methods, and the `newclasses` property. As a class developer, you can create your own named classes that make use of any valid combination of built-in classes and behaviors, or you can modify the properties and behaviors of existing classes. Please note that the word "user" in the term user-defined classes refers to a developer using APL. The term "user" applied otherwise still means an end user of an application program written by either the class developer or an application programmer using the new class.

The fundamental idea is to interrupt the system's normal processing at the point of creating APL+Win Windows interface (WI) objects. The mechanism for the interruption is the `onNew` event handler, which exists on the system object. In the `onNew` event handler, you define the behavior and characteristics to create the new class of object or modify the behavior and characteristics of an existing class. When the handler returns, the system resumes its normal processing.

The `onNew` event handler is triggered for any `⌵wi` statement that invokes the `New` or `Create` method on a class name (the first argument following the `New` or `Create` action) that is not one of the built-in APL+Win classes and also not a registered ActiveX control or server object. It is also triggered for any built-in class named in the `newclasses` property, except that the event handler can be suppressed by putting an asterisk in front of the class name in the `⌵wi` statement. Thus, in the case where the class name is listed in the `newclasses` property, the application programmer chooses each time he or she creates an object whether to invoke the standard behavior (`⌵wi 'New' '★Class'`) or the behavior defined in the `onNew` event-handler function (`⌵wi 'New' 'Class'`).

This is a very powerful facility, but you must take extra care when you use it. Since you are modifying the existing behavior of the `⌵wi` system function, APL+Win provides you with much more flexibility, but also much more responsibility. APL2000 recommends that you use the following form of the statement. This protects you when you load a workspace that does not contain the function you are using to define new behavior.

```
'#' ⌵wi 'onNew' "⌵(0≠⌵nc 'UDC_Fn')/'UDC_Fn' "
```

If you want to create a class with a new name (as opposed to modifying an existing class), define an `onNew` event handler function that creates an instance of your class; presumably, your handler code will detect the name of the class to be created. You must create at least one object recognized by APL+Win that has the name stored in `⌵wself`. When the function returns, the system processes everything in `⌵wres` using the object you created in the event handler function. If you have not modified `⌵wres` in your `onNew` event handler, that will consist of the rest of the arguments to the right of the class name. You can modify items in `⌵wres` or add items to `⌵wres` that further define your class.

Creating an Event Handler for `onNew`

The basic form of the `onNew` event handler statement is: `'#' ⌵wi 'onNew' 'UDC_Fn'` where you can use any name you choose for the function, `UDC_Fn`, that defines the special behavior. During

the `onNew` event handler, `□wself` contains the name of the object being created, which is normally the left argument to `□wi`.

The variable `□warg` contains the class name, and `□wres` contains the normalized right argument to `□wi`, including the method name, the class name, and any settings. For example, if an object is created with a statement like:

```
'form' □wi 'cked.New' 'CheckEdit' ('caption' 'Check it out')
```

- `□wself` is normalized as: `'form.cked'`
- `□wevent` is always: `'New'` for either the `New` or `Create` methods
- `□warg` is normalized as: `'CheckEdit'`
- `□wres` is normalized as: `'New' 'CheckEdit' ('caption' 'Check it out')`

The following rudimentary example shows how to create a compound control consisting of a check box and an edit control working together, both of which are children of a frame. The `UDC_Fn` function, displayed below, creates the `CheckEdit`, assuming that the `onNew` event handler is set for the system object as shown above.

The class name is `CheckEdit`. The frame is assigned the name that the system uses to create the `CheckEdit` control. The control will exhibit one unique behavior; when the check box is unchecked, the edit control is disabled. Lines 4 and 6 identify two functions to be triggered as event handlers. The `CheckEdit_Events` function houses the built-in behavior. The `CheckEdit_Actions` function defines the system response for special properties and methods, and it calls the events function.

```
▽ UDC_Fn;obj
[1] :select □warg
[2] :case 'CheckEdit'
[3]   obj ← □wi 'Create' 'Frame' ('style' 3) ('extent' 4 12)
[4]   □wi 'onAction' 'CheckEdit_Actions'
[5]   obj ← □wi '.ck.Create' 'Check' ('where' 0.5 0.5) ('caption' '')
[6]   obj □wi 'onClick' 'CheckEdit_Events'
[7]   obj ← □wi '.ed.Create' 'Edit' ('where' 2 .5 1 10) ('enabled' 0)
[8] :else
[9]   ⑈ Reserved for modifying a built-in class and other UDCs
[10] :end
```

Creating an Event Handler for onAction

APL+Win WI objects, including user-defined controls, have an event-handler property named `onAction`. After a control is created, every `⊞wi` statement that applies to the control triggers the Action event on that object. The class developer can specify any special behavior in the function that is called by the event handler, for this example, `CheckEdit_Actions`. During execution of an `onAction` event handler, the windows system variables have values similar to those described above for `onNew`, except that `⊞warg` contains the entire normalized right argument of `⊞wi`, and `⊞wres` is initialized. On return from the handler, any value you have set in `⊞wres` is returned as the explicit result of `⊞wi`. For example, if an object with an `onAction` handler is called with a statement like:

```
'form' ⊞wi 'cked.SomeAction' 'Arg1' 'Arg2'
```

- `⊞wself` is normalized to the name of the object; in this case: `'form.cked'`
- `⊞wevent` is always: `'Action'`
- `⊞warg` is normalized as: `'SomeAction' 'Arg1' 'Arg2'`
- `⊞wres` is initialized as: `0 0 ρ 0`

Note that `⊞warg` is always normalized as a nested *vector* whose first item is a character *vector*. This is true even for cases such as `⊞wi 'DoIt'` (where `⊞warg` would be `1ρ<'DoIt'`) and `⊞wi 'N'` (where `⊞warg` would be `1ρ<1ρ'N'`).

The body of an `onAction` handler function will typically be structured as a `:select` statement with a `:case` clause for each action you want to handle. The default `:else` clause (`⊞wres ← ⊞wi '★'`) passes all other actions to the system for their normal built-in handling. It is important to have a statement of this kind to handle default processing; otherwise you must account for every conceivable possibility in your `onAction` handler function.

The properties and methods that normally belong to the frame, for example where, can simply be passed through for normal processing. Properties and methods that normally belong to children of the frame must be specified. This procedure can apply to a property such as `caption`, which exists on the frame, but is more appropriate in this instance for the checkbox. It can also apply to properties, such as `text` and `value` that belong to the edit and checkbox controls respectively, but not the frame.

The first line of `CheckEdit_Actions` is the overall `:select` statement. It works on the first element of `⊞warg`, which is the action name following `⊞wi`. Lines 3-5 define the `caption` property for the `CheckEdit` control to be the caption of the checkbox rather than the frame. This positions the identifier inside the apparent boundaries of the control. This case also associates a `value` property with the `CheckEdit`; this is the value of the checkbox, 1 if checked, zero otherwise.

Lines 7-9 make a special case of the read-only class property. It identifies the object as `CheckEdit`. To be thorough, the lists of items in the properties, methods, and events properties should be adjusted to reflect the changes specified in this handler. You would do that by restructuring each list and assigning it to `⊞wres`, as is done for `class`.

Lines 11-13 associate selected properties of the `CheckEdit` control with the corresponding properties of edit box. Note that some of them exist for the edit but not the container (frame), while others replace the frame's property. Lines 15-17 expose the methods we wish to show for the `CheckEdit` control. These methods allow the application programmer to use the `Copy` method, for example, on the `CheckEdit` control to copy the contents of the edit box to the clipboard. Other methods normally associated with an edit box, such as `CharToLine` and `LineToChar`, are not included because there is no provision for a multi-line edit control. If you allowed different styles on the edit control, you might wish to include these methods on the `CheckEdit`.

In a similar fashion, you could associate selected event handler properties of the `CheckEdit` with the event handlers of its child controls; however, this is not totally satisfactory, because the value of `□wself` during the event would be the child of the `CheckEdit` rather than the `CheckEdit` itself where the handler is assigned. A better technique is shown starting at line 19. This stores the handlers in user-defined properties that are executed using the `Event` (or `Defer`) method with `□wself` set to be the `CheckEdit`. See below for more explanation. Line 38 provides default processing for every other action.

```
▽ CheckEdit_Actions
[1]   :select ↑□warg
[2]
[3]   A   Put any caption on the checkbox; use value property for ckedit
[4]       :caselist 'caption' 'value'
[5]           □wres ← '.ck' □wi □warg
[6]
[7]   A   Identify the defined class
[8]       :case 'class'
[9]           □wres ← 'CheckEdit'
[10]
[11]  A   Reassign selected properties from the Edit to the CheckEdit
[12]      :caselist 'color' 'font' 'keys' 'limit' 'range' 'selection'
[13]      'seltext' 'text' 'translate'
[14]      □wres ← '.ed' □wi □warg
[15]  A   Reassign these methods to the CheckEdit control similarly
[16]      :caselist 'Copy' 'Cut' 'Focus' 'Paste'
[17]      □wres ← '.ed' □wi □warg
[18]
[19]  A   Store the programmer's onClick handler in a user-defined
[20]  A   property on the checkbox. It is executed by CheckEdit_Events
[21]      :case 'onClick'
[22]          :if 1=ρ□warg
[23]              □wres ← '.ck' □wi 'ΔonClick'
[24]          :else
[25]              '.ck' □wi 'ΔonClick' (↑1↓□warg)
[26]          :endif
[27]
[28]  A   Store Edit event handlers; call CheckEdit_Events when defined
[29]      :caselist 'onChange' 'onFocus' 'onKeyDown' 'onKeyPress'
[30]      'onKeyUp' 'onLimit' 'onMemory' 'onScroll' 'onUnfocus'
[31]      :if 1=ρ□warg
[32]          □wres ← '.ed' □wi 'Δ', ↑□warg
[33]      :else
[34]          '.ed' □wi ('Δ',↑□warg) (↑1↓□warg)
[35]          '.ed' □wi (↑□warg) 'CheckEdit_Events'
[36]      :endif
[37]      :else
[38]          □wres ← □wi '★' A Default handling of other actions
[39]      :end
```

Creating Events for the User-Defined Class

Handling events on an instance of a user-defined class presents a different problem than handling properties and methods. When the application program sets or references a property, or invokes a method, a statement invokes `□wi` on the user-defined object. This triggers the `Action` event, and the `onAction` event handler responds. The result is determined by the code in the event handler.

Events, by contrast, can be triggered other than by a program statement, frequently by a user action. In order for a user-defined object to respond, the system must, in essence, transfer the event trigger to the user-defined object so that the appropriate object responds. The technique used is to invoke a function in

which `□wself`, `□wevent`, `□warg`, and `□wres` are localized to values you specify. When the function runs, it appears as though it is an event handler with the chosen name occurring for the user-defined object. Two methods, `Event` and `Defer`, invoke such a function.

As the class developer, you invoke one of these two methods in response to the actual trigger (that is, the specific user action when it will occur) as part of defining the behavior of your special class. When you, or an application programmer, uses your special class in an application, the specific behavior that occurs in response to the event is defined as an argument to the method.

In using these methods while defining your class, you specify the event name, which can be the name of an event that is recognized by APL+Win or a name that you create. You provide a user-defined property to hold the definition of the event handler that is specified as part of the application to respond to the event.

Suppose you want to trigger a special event, for example, when the end user of an application using your specially defined control presses a key when the focus is on a particular component of your user-defined object. You can use the `onKeyDown` event handler to invoke the `Event` method that localizes the event name as `Whack` ("Hit any key to continue"). The `Event` method, applied on your object, runs the statement or function that appears to be the `onWhack` event handler. Conceptually, this technique is similar to using the `Send` method on a built-in class, which triggers the `onSend` handler on a named object.

You can use either the `Event` method or the `Defer` method on your object. You follow the method name with the handler and the event name (which localizes `□wevent`), followed optionally by any values you want to have in `□warg` and `□wres`. The syntax is:

```
object_name □wi 'Event' handler event_name arguments result allow
```

where *object_name* is the name of your user-defined control and *handler* is the expression or function name that is the desired event handler; although the name of the handler function is specified once in this definition of the event, the expression it represents can be different for each instance of your user-defined class. The remaining arguments are optional. The last one is a flag that determines under which circumstances the handler executes; it is explained below. The syntax for the `Defer` method is the same. The difference between these methods is that `Event` executes the handler immediately, while `Defer` executes the handler when the system becomes "idle."

Becoming idle often means that the system must return from the function that invoked the `Defer` statement before the deferred handler can run. The deferred statement can run before the calling function returns, if the calling function contains some action that allows the system to become idle, such as `□wgive`, `□dl`, or a `Wait`. Note that simply returning from the calling function is not sufficient to trigger the deferred action. All immediately impending actions must complete before the deferred statement is executed. (These methods also have uses beyond what is described here; in particular, you are likely to find the `Defer` method useful other than with user-defined classes.)

The *allow* argument is a flag that you can use to allow an event to be fired even when an object is closed or the system is suspended. Normally, events are suppressed on closed objects or when the system is suspended, and this is the default behavior for `Event` and `Defer`. If you don't specify the *allow* argument or set it to zero, then the event handler is not triggered under these conditions. The `allow` flag must be an integer; it can have any combination of the following values:

- 0 Don't allow events if the object is closed or the system is suspended; this is the default value
- 1 Allow events if the object is closed
- 2 Allow events if the object is opening (in transition between closed and open states)
- 4 Allow events if the system is suspended.

The function below named `CheckEdit_Events` is the code that invokes the event handlers defined on the `CheckEdit`. The `onClick` event handler, which is triggered by clicking the checkbox, includes the default

built-in behavior for the Click event for the CheckEdit control (toggling the enabled property of the edit box) as well as any behavior the application programmer defines. The user-defined property `ΔonClick` was created above to store the handler function. The event handlers that are triggered on the edit box simply execute the expressions contained in the corresponding user-defined properties.

An important feature of the Event and Defer methods is that they accept a zero as an argument. When they encounter zero, they essentially ignore the call. This makes it possible for an undefined user-defined property (`Δudp`), which would nominally hold the event handler, to be referenced without generating an error.

```
▽ CheckEdit_Events
[1] :select 0wevent
[2]   A Define the default behavior for a click on the checkbox
[3]   A Then execute the onClick handler defined for the UDC
[4] :case 'Click'
[5]   '..' 0wi '.ed.enabled' (0wi 'value')
[6]   handler ← 0wi 'ΔonClick'
[7]   0wres ← '..' 0wi '*Event' handler 'Click'
[8]
[9]   A Execute any edit box event handlers defined on the checkedit
[10] :caselist 'Change' 'Focus' 'KeyDown' 'KeyPress' 'KeyUp' 'Limit'
    'Memory' 'Scroll'
[11]   handler ← 0wi ('Δon',0wevent)
[12]   0wres ← '..' 0wi '*Event' handler 0wevent 0warg 0wres
[13]   A Defer onFocus to avoid Windows conflicts
[14] :case 'Unfocus'
[15]   handler ← 0wi ('Δon',0wevent)
[16]   0wres ← '..' 0wi '*Defer' handler 0wevent 0warg 0wres
[17] :end
```

Note that the statements that use the Event and Defer methods use the asterisk as a prefix to avoid unwanted recursion. See the section "Controlling Default versus Special Behavior," below.

The function `ProgHndl` shows a trivial example of arbitrary event handlers that a programmer could write using the CheckEdit control. These handlers are part of building the application, not building the user-defined control. These handlers change the background color of the edit box that is part of the CheckEdit control. Note that the Focus and Unfocus events that fire on the edit control call the `CheckEdit_Events` function; this is part of the CheckEdit behavior. The `CheckEdit_Events` function invokes the Event or Defer method to trigger the `onFocus` event handler defined by the programmer, in this case `ProgHndl`.

```
▽ ProgHndl
[1] :select 0wevent
[2] :case 'Focus'
[3]   0wi 'color' 255 255 255
[4] :case 'Unfocus'
[5]   :if (0wi 'self')='g.cked'
[6]     0wi 'color' 255 255 0
[7]   :else
[8]     0wi 'color' 0 255 0
[9]   :endif
[10] :end
```

In the application, you need to specify the `onFocus` and `onUnfocus` event handlers.

```
'g.cked' 0wi 'Set' ('onFocus' 'ProgHndl') ('onUnfocus' 'ProgHndl')
'g.duce' 0wi 'Set' ('onFocus' 'ProgHndl') ('onUnfocus' 'ProgHndl')
```

To demonstrate the validity of this user-defined control, create a form and place one or more instances of the class on the form. The function below establishes the necessary `onNew` handler on the system object and creates two CheckEdit controls. For the second one, the `style` property has been altered arbitrarily. This overrides the style setting for the frame in the `UDC_Fn` function.

```

▽   FormG;t
[1] '#' □wi 'onNew' "ⓧ(0≠□nc 'UDC_Fn')/'UDC_Fn' "
[2] t←'g' □wi 'Create' 'Form' ('where' 1 60 15 40) ('caption' 'Form G')
[3] t←'g.cked' □wi 'Create' 'CheckEdit' ('where' 1 1) ('caption' 'Test UDC')
[4] t ← 'g.duce' □wi 'Create' 'CheckEdit' ('where' 1 14)
    ( 'caption' 'UDC Clone') ('style' 4)

```

To see the full effect of the example, be sure to set the `onFocus` and `onUnfocus` event handlers.

Programming Style for User-Defined Classes

The functions, `CheckEdit_Actions` and `CheckEdit_Events`, used above were separated for the purpose of describing the different techniques. A class developer would typically combine these into a single function with a name such as `UDC_CheckEdit`. Furthermore, when these are combined into a single `UDC_CheckEdit` function, you could also include the `onNew` handling logic in the same function. When your code is structured in that fashion, it is possible to create a completely generic `onNew` handler function that uses code such as `(3=□nc 'UDC_', □warg)` to check for the existence of a class handler function and call it to create an object (if the handler for that name exists).

Modifying a Built-in Class

You can also define specific modifications in characteristics or behavior to one of the classes built in to APL+Win. If you do it by using the `onNew` event handler, the fundamental idea is the same: You intercept the processing of the system at the point of creating an APL+Win WI object, insert the behavior you want to define, and return to the system's processing.

Specifying the Class to be Modified

Conceptually, this technique is more difficult, but in practice it is simpler. You use the same statement to set the `onNew` event handler on the system object that you used or would use for a user-defined class with a different name. You establish your design criteria in the same event-handler function.

There is an additional step, however; you must name the class you want to modify in the `newclasses` property of the system object. When a built-in class is named in `newclasses`, a statement that attempts to create an instance of that class, using either the `New` method or the `Create` method, triggers the `onNew` event handler.

If you create the object in your handler, the system resumes processing with the arguments that followed the class name. If you do not create the object in your event handler, for example, you may just modify `□wres`, the system attempts to create an object of the appropriate class, using the results of the changes you made in the `onNew` event handler. (You may also want to include your user defined class names in the `newclasses` list to avoid problems if one of your user defined class names gets added as a built-in class in a future release of the APL system.)

Using the Event Handler for onNew

The following example shows how to modify an instance of a `RichEdit` control, changing some of the default settings when an instance is created. The `UDC_Fn` function, which was shown above, establishes the settings assuming that the proper `onNew` event handler is set for the system object and you have already set the `newclasses` property of the system object.

```

'#' □wi 'onNew' "ⓧ(0≠□nc 'UDC_Fn')/'UDC_Fn' "
'#' □wi 'newclasses' 'RichEdit'

```

Suppose you want to change the defaults for your RichEdit controls so they are multiline controls that allow Enter characters, have a vertical scroll bar when needed, do not select text when tabbing back to the control, do not hide the selection when tabbing out of the field, and have a tinted background to distinguish them from edit controls. The `onNew` event handler was shown above when creating a user-defined class. It is extended below to modify RichEdit.

The `:if` statement in the `RichEdit` case adds a style setting to the arguments used when the system creates the object. The five values establish the characteristics mentioned above. The `:else` block of statements occur if the programmer sets `style` in the same statement that creates the object. That locates the style argument, and adds any of the five values that are not included.

```
▽ UDC_Fn
[1] :select Ɑwarg
[2] :case 'CheckEdit'
[3]   A The lines for CheckEdit are snipped here

[8] :case 'RichEdit'
[9]   index ← (↑Ɑwres) Ɑ c'style'   A Find if prog specifies style
[10]  :if index ≤ ρ Ɑwres
[11]    scalar_style ← +/ 1Ɑ index > Ɑwres
[12]    vector_style ← 2 * ((15ρ2)↑scalar_style)/(15-115)
[13]    A Append any of the default styles the prog didn't specify
[14]    newstyle ← vector_style,(4 16 1024 8192 16384 ~ vector_style)
[15]    (index > Ɑwres) ← (c'style') , newstyle
[16]  :else
[17]    Ɑwres ← Ɑwres,c'style' 4 16 1024 8192 16384
[18]  :endif
[19]  Ɑwres ← Ɑwres,c'color' 220 220 220   A Specify color regardless
[20] :end
```

This function does not preclude the application programmer changing the value of the `style` property or the `color` property after the object is created.

When you have a built-in class named in the `newclasses` property and an `onNew` event handler that modifies it, you can still create an instance of the class with all the default characteristics. Simply prefix the class name with an asterisk to bypass the special processing. Thus, the following statement suppresses any special handling when creating an instance of a RichEdit control.

```
object Ɑwi 'New' '★RichEdit' (additional settings. . .)
```

Modifying Built-in Classes without using onNew

You can also modify the behavior of built-in classes without using the `onNew` event handler technique. For example, you might want to change only the behavior of certain specific instances of a class rather than all occurrences. To do this, you can simply set an appropriate `onAction` handler for the object when it is created.

Note that you can also use the `def` property of any control (or its container) without regard for the techniques discussed in this section. Using the `def` property in either of these two forms:

```
object Ɑwi 'New' def
object Ɑwi 'def' def
```

does not invoke the `onNew` event handler. You will, however, need to define a handler for any properties, methods and events that are not standard.

Also note that "Make" functions created using the `Ɑwmake` user command are constructed so that the `onAction` event handler property is set last, after all other properties have been set for an object. This results from the fact that the state property returns `onAction` as the last row of its result, and the `Ɑwmake` command

sets properties in the order returned by `state`. Thus, Make functions behave like the `def` property, since the `onAction` handler is not executed when the objects are being created and initialized in the function.

Controlling Default versus Special Behavior

In working with a complex facility like creating user-defined classes or using them in an application, there are times that you want to invoke the special processing and other times that you may want the normal system behavior. You can control this by using the asterisk to suppress the special behavior of user-defined classes. You use the asterisk by itself as an argument or as a prefix to a class name or an action name. There are a number of different situations when you can effectively specify this syntax.

- When you are defining the behavior of a class that you name or modifying the behavior of a built-in class, you should have a default statement that processes any action with normal system behavior that you have not explicitly modified. You do this by setting `□wres` in your `onAction` event handler:
`(□wres ← □wi '★')`
- In some cases you may want to use a `□wi '★'` statement when you are modifying the behavior of some action. For example, when implementing a custom version of the `methods` property, you might call `□wi '★'` to get the built-in set of methods for the object. You could then add or delete methods from this list before returning from your handler, so it represents all the applicable methods of the object, both built-in and user-defined.
- When you are defining the behavior of a class that you name or modifying the behavior of a built-in class and you call a function that uses a `□wi` statement, you must invoke normal behavior on the nested statement or you may find yourself in an infinite recursion. For example, see the definition of events on the `CheckEdit` object above. The statement used is: `'..' □wi '★Event' handler 'Click'`
 If there were no asterisk preceding the `Event` action word, the system would try to process the statement through the `onAction` event handler.
- When you are programming using a modification to a built-in class, that is, one that is named in the `newclasses` property, you can create a standard instance of the built-in class by prefixing the class name with the asterisk. In the example used earlier in the chapter, the statement
`object □wi 'New' '★RichEdit' (additional settings. . .)`
 suppresses any special definition of the class and creates a "normal" `RichEdit` control.
- When you are programming an application using specially defined classes, you can cause any `□wi` statement, other than one that creates an object, to bypass the special handling by prefixing the action name, that is the first argument after `□wi`, with the asterisk. The statement
`object □wi '★Action' arguments`
 causes the system to bypass `onAction` for any property, event-handler or method name (excluding `New` and `Create`). If the Action name is prefixed by an object name, such as
`'a' □wi 'b.★Resize'`
 the asterisk must come after the object name prefix, immediately before the action name

Caution: Use with Care



As you were warned at the beginning of this topic, this is a very powerful facility, but you must take extra care when you use it. Since you are modifying the existing behavior of the `□wi` system function, APL+Win provides you with much more flexibility, but also much more responsibility.

If you set the `onAction` event handler on the system object for example, and you do not provide for default processing of most statements, you can trap yourself. You can reference the event-handler property and get an empty result. But the `onAction` handler will still exist.

```
▽ Braveheart
[1] arg ← ↑□warg
[2] :select arg
[3] :case 'dragsource'
[4]   'Do something funny to my system during Drag-and-Drop'
[5] :end

'#' □wi 'onAction' 'Braveheart'
'#' □wi 'onAction'
      ⌘ Despite the empty result, onAction still is set.
'#' □wi '★onAction'
Braveheart
'#' □wi 'dragsource'
Do something funny to my system during Drag-and-Drop
'#' □wi 'Reset'
      ⌘ Because only dragsource is handled, this has no effect.
```

Remember Your Escape Hatch

```
'#' □wi '★onAction' '' ⌘ This is your escape hatch.
'#' □wi '★Reset'      ⌘ You may also find this useful.
```

Chapter 10: Using the APL Registry Object

Introduction

The APL Registry Object is an ActiveX object that you can use in an APL application or session to read from and write to the Windows Registry. Although this tool makes it easy to do so, it does not protect you from yourself.

WARNING: IT IS INHERENTLY DANGEROUS TO MODIFY THE WINDOWS REGISTRY DIRECTLY!

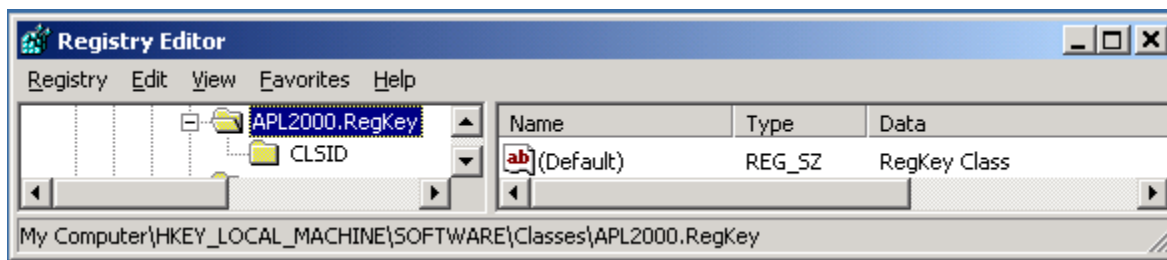
At the very least, you should make a complete backup of your Registry before you experiment with or use this facility, and you should be sure you know how to reset the Registry from your backup. Consult the Help menu on the Registry Editor for guidance.

The Windows Registry is a hierarchical database of values, names, settings, locations and other information about various users, preferences, hardware, and applications installed on a machine using a Windows operating system. Although its scope is much broader, one important use for the Registry is to hold information about an application that formerly might have been contained in a .INI file. Thus, to adhere to the current Microsoft preferred practice, you may have a need to read from and write to the Registry.

You can view the Registry by invoking `regedit` from the Start/Run menu; exactly what you see depends on which version of Windows you are running. The Registry has a tree structure analogous to the file system. It consists of "keys" and "subkeys" that compare to directories and subdirectories. The first level keys, sometimes called the "root keys" are fixed by Microsoft, although they vary slightly by system. There are some standard second level keys, some depend on your configuration, and you may create others. A key or subkey can contain "values" as a directory holds files. A "value" in this special Registry sense of the word is made up of three parts: its name (usually referred to in this document as "value-name"), a type (usually referred to in this document as "value-type"), and data (usually referred to in this document as "data-value," but also called the "content" of the value). Each key has one entry for a value which appears in the registry with "(Default)" as its name, but in actuality the name is empty; initially it has no data-value. In any key, you can assign a data-value to the default and you can add other named values with or without setting the default. You cannot rename the (Default) value.

Note that the word "value" is overloaded in this context. It has a special meaning with regard to the Registry and this can cause confusion. The value-type is one part of a value, but it really refers to the "type" of the data (such as `DWORD`, the Registry term for integer), which is another part of the "value." We refer to the data, or data-value, interpreted according to its value-type, as the "content" to avoid using the awkward phrase "value of the value."

This is a picture of the default Registry Editor on Windows 2000/XP. On a different version of Windows, this may look somewhat different. In particular, you may not see the Type column on some systems.



In this example, the root key is `HKEY_LOCAL_MACHINE`; three levels down, the highlighted key is `SOFTWARE\Classes\APL2000.RegKey`. It has one subkey and one value.

The Registry Editor always shows at least one entry under Name; the "(Default)" value-name often shows "(value not set)" under Data. When there is no data-value set, the entry effectively does not exist; there is only a placeholder. In this case, the data-value, or content of the default is the character vector: `'RegKey Class'`

Note that in the Registry Editor, the subkeys show up only on the left-hand side of the tree display and the values in the right-hand pane, unlike the Windows Explorer where the subdirectories also show up on the right-hand side above the file names.

Getting Started

The `RegKey` object is contained in the file `APLUtil.DLL`; install this file in the directory with `APLW.EXE` and register it by running `REGSVR32` with the full path name as the argument.

You invoke an instance of `RegKey` using a `␣wi` statement with the `New` method and the `progid`. Initially, the object exists, but it has a null key handle. When you assign a value to the `xKeyName` property, the name you assigned in the creation statement is associated with that particular Registry key; we call this the current key. Almost every other property and method you invoke operates in relation to the current key.

```
'rk' ␣wi 'New' 'APL2000.RegKey'
rk
      'rk' ␣wi 'xHKey'
0
```

The `xHKey` property returns the Windows handle of the current key.

```
'rk' ␣wi 'xKeyName' 'HKLM\Software\Classes\APL2000.RegKey'
      'rk' ␣wi 'xHKey'
244
```

The key has one subkey, and, although it appears to have no values, the default value is present. You query the default value by using `' '` or `0` as the argument for its name. The picture above shows that value in the Registry.

```
'rk' ␣wi 'xSubKeys'
CLSID
      'rk' ␣wi 'xValueNames'
ρ'rk' ␣wi 'xValueNames'
1
      'rk' ␣wi 'XGetValue' ''
RegKey Class
```

Programming Alert: While you are programming `RegKey`, you can invoke quick reference help on any ActiveX property, method, and event handler by executing a `␣wi` statement with a question mark in front of the name of the action you are interested in. The help shows the syntax in APL format, the names of the arguments, and a short description of the relevant codes, values, or uses.

```
␣wi '?xAccess'
xAccess property:
  Result ← ␣WI 'xAccess'
ax
  ␣WI 'xAccess' Code
Code specifying type of access desired
Code: Character vector or integer MS code; valid simple entries are:
  'read', 'write', or 'all' (default); integers are a mask for security acce
ss
```

A	Help string header
A	Help string reference synt
A	Help string setting syntax
A	Description


```

bits as interpreted by the RegOpenKeyEx Windows API call
Result: Integer representing the mask of access bits that are set; the three
that correspond to the simple entries are: 131097, 131078, 983103 (default
)
Read access allows you to get Registry values and to create subkeys;
Write access allows you to create or set a Registry value with XSetTypedValue;
All access allows use of all the methods and properties
If you set xAccess, it reopens the key; the setting continues to apply to
the named APL+Win object even when the key name changes

```

You can change the key name to the named subkey and query its default value.

```

'rk' □wi 'xKeyName' 'HKLM\Software\Classes\APL2000.RegKey\CLSID'
'rk' □wi 'xGetValue' ''
{CBA0AD4C-1217-4895-B4C3-83E3B48D0B86}

```

Note that this is the GUID for the RegKey object. Using this Registry entry is how Windows finds the GUID to create this object from the progid or the description. In general, this document makes no attempt to explain to you how the Registry works. If you use the APL2000 RegKey object, you are responsible for your own actions.

Using RegKey Methods

In the interests of safety, we will create a meaningless key and all further experimentation will be made on that key and any subkeys and the values created for them.

```

'rk' □wi 'xKeyName' 'HKLM\Software'
'rk' □wi 'XCreateSubkey' 'APLexpt'
19940216 1
'rk' □wi 'xKeyName' 'HKLM\Software\APLexpt'
'rk' □wi 'XCreateSubkey>rk2' 'Values'
'rk2' □wi 'XSetValue' 'mynum' 1234
'rk2' □wi 'XSetValue' 'myvec' 'What's up Doc?'

```

The XCreateSubkey method allows you to create a subkey of the current key. If you invoke it with a "normal" syntax, it returns a pointer to the subkey and a code that indicates if the key was created (1, as above), or alternatively (2), that a subkey with that name already existed and it was opened. If you know the subkey exists, you can obtain a pointer by using the XOpenSubkey method.

The existence of the pointer confirms the success of the action. If you are programming a function, you could use that pointer as an argument to a call within an APL+Win callback. You can also use the redirection syntax as shown with the sub-subkey example above. Using the greater than sign (>) followed by a name associates the pointer with an APL+Win Windows interface object. This object is another instance of RegKey with the named subkey as its current key.

```

'rk2' □wi 'xKeyName'
\HKLM\Software\APLexpt\Values

```

The XSetValue method works to create a value if you want to use the simple integer or character vector value-types. There was an example of the counterpart method XGetValue used above. If you want to use other value-types, you must use the XGetTypedValue and XSetTypedValue methods. In order to demonstrate those, it is first useful to look at the possible value-types.

If you make a mistake, you can use either the XDeleteSubkey or the XDeleteValue method followed by the name of the subkey or value you wish to delete. There is no undo functionality in the Registry.

Value-Types in the Registry

RegKey recognizes five value-types by name; three of them are character vectors. As shown above, you can set a simple character vector, (value-type "string") without specifying the type explicitly. To retrieve the data-value:

```
'rk2' ⌐wi 'XGetTypedValue' 'myvec' 'string' 'xxx'
What's up Doc?
```

This method returns a value if its value-type matches the specified value-type (in this example, string) or an error if it is of another value-type. The last argument, which is optional, is returned if the named value does not exist. If you omit the last argument, a name mismatch returns an error indication.

```
'rk2' ⌐wi 'XGetTypedValue' 'myvec0' 'string' 'xxx'
xxx
'rk2' ⌐wi 'XGetTypedValue' 'myvec0' 'string'
-2147352572
```

You can also set and reference a nested vector of character vectors which is a value-type named 'multistring'; in the registry, it is stored as a series of null-delimited vectors. RegKey handles the conversion in both directions.

```
'rk2' ⌐wi 'XGetTypedValue' 'myMulti' 'multistring' 'yyy'
alpha bravo charlie dog echo
⌐'rk2' ⌐wi 'XGetTypedValue' 'myMulti' 'multistring'
5
```

Another simple string value with a special characteristic is named 'expandstring'; this is a character vector that includes one or more named environment variables, enclosed in percent signs. When this data-value is used, the value of the variable replaces its name in the string. You cannot assign this value-type with XSetValue; you must use the XSetTypedValue method.

```
'rk2' ⌐wi 'XSetTypedValue' 'myXvec' 'expandstring' 'This is %Os2LibPath%h%'
'rk2' ⌐wi 'XGetTypedValue' 'myXvec' 'expandstring'
This is %Os2LibPath%
'rk2' ⌐wi 'XGetValue' 'myXvec'
This is C:\WINNT\system32\os2\dll;
```

You can view some environment variables by clicking on Settings\Control Panel\System; you can set your own, among other places, in your autoexec file.

You can set integer values simply as shown earlier. Integers comprise four bytes; they are stored in the reverse order (little-endian) of the way you would ordinarily read them. The hexadecimal value equates to the decimal 1234 (210 + 256 × 4).

```
'rk2' ⌐wi 'XGetValue' 'mynum'
1234
'rk2' ⌐wi 'XGetTypedValue' 'mynum' 'int'
1234
(⌐av 1 2> 'rk2' ⌐wi 'XGetTypedValue' 'mynum' 'raw' )-⌐io
210 4 0 0
```

Note the use of the 'raw' argument to XGetTypedValue. You can retrieve the data-value of any value-type using raw. The method returns a two-element vector with an integer code (see below) designating the value-type as the first element and an APL character vector representing the raw bytes of the data-value as the second element. The indices to ⌐av are the decimal equivalent of the hexadecimal byte values.

The other named value-type is 'binary'. You can set an arbitrary string of hexadecimal bytes by assigning a character vector whose positions in ⌐av are the equivalent of the value you want to store.

```
'rk2' ⌐wi 'XSetTypedValue' 'myBval' 'binary' (⌐av[⌐io+170 187 204 221]
)
```

```

      'rk2' ␣wi 'XGetTypedValue' 'myBval' 'binary'
õ+İÝ
      (␣av ␣ 'rk2' ␣wi 'XGetTypedValue' 'myBval' 'binary') -␣io
170 187 204 221
      (␣av ␣ 2 ␣ 'rk2' ␣wi 'XGetTypedValue' 'myBval' 'raw') -␣io
170 187 204 221

```

The value-types recognized by RegKey are summarized in the table below.

Name	Code	Windows Name	Nature of the data-value
string	1	REG_SZ	Character vector
multistring	7	REG_MULTI_SZ	Nested vector of character vectors
expandstring	2	REG_EXPAND_SZ	Character vector with embedded environment variables
int	4	REG_DWORD	Four-byte integers
binary	3	REG_BINARY	Hexadecimal bytes

The Registry also recognizes some other value-types. For purposes of RegKey you can read or create these value-types using XSetTypedValue with the 'raw' argument and a two-element vector, the numeric code for the value-type and the ␣av indexed character vector for the content. (See XSetTypedValue for options).

Code	Windows Name	Nature of the data-value
5	REG_DWORD_BIG_ENDIAN	Four-byte values stored in assigned order
6	REG_LINK	A Unicode symbolic link
8	REG_RESOURCE_LIST	See below
9	REG_FULL_RESOURCE_DESCRIPTOR	
10	REG_RESOURCE_REQUIREMENTS_LIST	

There are a number of caveats regarding these value-types. Value-type 5 on a Windows 2000/XP system displays its value-type in the Registry editor as REG_DWORD_LITTLE_ENDIAN. It seems clear this is a Microsoft mistake, but the actual values are simply the assigned bytes in order.

One item of Microsoft documentation says of value-type 6: "Used internally; applications should not use this type." In another place MS says: "Indicates a symbolic link between system or application data and a registry value. You can use Unicode characters in a REG_LINK entry." Use at your own risk.

Microsoft describes value-type 9 as: "A series of nested arrays designed to store a resource list for a hardware component or driver. For example, in Regedt32, double-click ConfigurationData (in HKEY_LOCAL_MACHINE\Hardware\Description\System\MultifunctionAdapter\0\ControllerName\0). Presumably, this is a literal path down to the point of ControllerName. Note that Microsoft labels this "nested arrays"; these are not APL nested arrays.

Not all value-types are present or recognized on all Windows systems. You may also see a listing for value-type 0, REG_NONE. You can create other numbered value-types and assign their data-values with a string of hexadecimal values.

You can query the value-type of any named value using the xValueType property with the name as its argument. If the value-type is one of the five that RegKey recognizes by name, the character vector name is returned. Otherwise, the integer code for the value-type is returned.

Some versions of the Registry do not recognize Unicode characters. You can query the flag in the xUnicodeRegistry property, which returns a 1 if the registry is Unicode. This may be important, for

example, if you are using `XSetTypedValue` or `XGetTypedValue` for strings. In a Unicode registry, each character requires two bytes, so for simple character vectors, every other byte is null.

```
'rk2' 0wi 'xUnicodeRegistry'
1
(0av 1 2 0'rk2' 0wi 'XGetTypedValue' 'myvec' 'raw' )-0io
87 0 104 0 97 0 116 0 39 0 115 0 32 0 117 0 112 0 32 0 68 0 111 0 99 0 63 0
0 0
W h a t ' s space u p space D o c ? N
ull
```

Using the RegKey Event

There is one event handler associated with `RegKey`; it is named `onXKeyChange`. The event fires when certain items in the registry change; the conditions that determine the change are identified in three properties, `xChangeValue`, `xChangeName`, and `xChangeAttr`. If `xChangeValue` is set to 1, setting, resetting, or deleting any value-name or data-value on the current key triggers the event. If this property is set to 2, changing a value-name or data-value on any subkey descendant of the current key also triggers the event. If it has its default value of zero, changing any value does not trigger the event.

The `xChangeName` property is similarly ternary valued, but its scope is slightly different. If `xChangeName` is set to 1, deleting the current key or setting or deleting any of its first-level subkeys triggers the event. If its value is 2, changing any subkey descendant also triggers the event. The default value of zero means changing any key does not trigger the event. Note that this "name" change does not refer to the entry in the Name column of the Registry editor, where "name" is the name of a "value." This name refers to (the name of) a key or subkey.

The `xChangeAttr` property specifies that changes in attributes trigger the event. Although attributes can be quite complicated, as you can discover if you delve into the `xAccess` property numeric values, it basically means the permissions for the key. If this property is set to 1, changing attributes on the current key triggers the event, and if it is set to 2, changing attributes on any subkey of the current key also triggers the event.

When the event fires, `0warg` contains two elements. The first element is the name of the property, excluding the `x` prefix, that set the condition that caused the event to fire. The second element matches the value of that property: 1 if the conditions are limited to the current key for values and attributes and to first-level subkeys for name; 2 if the conditions for triggering the event extend to all levels of subkey descendants of the current key. Note that the second element does not tell you where the change occurred; it tells you only the setting.

Properties Reference

xAccess

Description:

This property is a code that specifies the type of access you want for the current key. The bits are interpreted

by the `RegOpenKeyEx` Windows API call. `RegKey` recognizes three values (read access, write access, and all, meaning both read and write access) by character vector name. The default value is all.

Syntax:

You reference this property without an argument. It returns an integer showing the security access mask values. You can set this property with an integer value if you know the meanings of the masks. You can set it simply with one of three character vectors: 'read', 'write', or 'all'

Remarks:

Read access allows you to get Registry values and to create subkeys; write access allows you to create or set a Registry value with `XSetTypedValue`; all access allows use of all the methods and properties. The setting continues to apply to the named APL+Win object even when the key name changes. If you set `xAccess`, it reopens the key. The three values that correspond to the simple entries are: 131097 (read), 131078 (write), and 983103 (all).

xChangeAttr

Description:

This property is a ternary-valued code that specifies whether changing the attributes of the current key or one of its subkeys triggers the `onXKeyChange` event.

0	Changing attributes does not trigger the event (this is the default value)
1	Changing attributes of the current key triggers the event
2	Changing attributes of any subkey descendant also triggers the event

Syntax:

You reference this property without an argument. It returns an integer scalar with one of the three possible values. You set this property with an integer in the range 0-2.

Remarks:

The setting continues to apply to the named APL+Win object even when the key name changes. Attributes, for practical purposes, means access privileges.

xChangeName

Description:

This property is a ternary-valued code that specifies whether deleting the current key or setting or deleting one of its subkeys triggers the `onXKeyChange` event.

0	Setting or deleting a key does not trigger the event (this is the default value)
1	Deleting the current key or setting or deleting a first-level subkey triggers the event
2	Setting or deleting any subkey descendant also triggers the event

Syntax:

You reference this property without an argument. It returns an integer scalar with one of the three possible values. You set this property with an integer in the range 0-2.

Remarks:

There is no way of literally changing the name of a key in `RegKey`. The Registry editor allows you to change a name, but it performs that action by creating a new key, copying the associated values, and deleting the old key.

The setting of `xChangeName` continues to apply to the named APL+Win object even when you change which key is the current key.

The "name" to which this property refers is not the entry under the Name column in the Registry editor; that is the name of the so-called "value." This "name" refers to the (name of) a key or subkey.

xChangeValue

Description:

This property is a ternary-valued code that specifies whether setting, resetting, or deleting a value-name or data-value on the current key or one of its subkeys triggers the onXKeyChange event.

- 0 Changing a value does not trigger the event (this is the default setting)
- 1 Changing a value of the current key triggers the event
- 2 Changing a value of any subkey descendant also triggers the event

Syntax:

You reference this property without an argument. It returns an integer scalar with one of the three possible values. You set this property with an integer in the range 0-2.

Remarks:

The setting continues to apply to the named APL+Win object even when the key name changes.

xHKey

Description:

This property is a read-only integer that is the Windows handle of the current key.

Syntax:

You reference this property without an argument.

Remarks:

This handle changes whenever xKeyName or xAccess changes. You can use the handle as an argument for low-level Windows calls to the Registry.

xKeyName

Description:

This property is a character vector that specifies the name of the current key. Typically, when working on a single machine, it is like a path name that starts with an abbreviation for the root key name followed by a backslash-delimited tree hierarchy of subkeys. You can also use the full name of the root key. Key names are case-insensitive in the Registry. The Windows root keys and the abbreviations you can use include the following:

```
HKLM=HKEY_LOCAL_MACHINE
HKCU=HKEY_CURRENT_USER
HKCC=HKEY_CURRENT_CONFIG
HKCR=HKEY_CLASSES_ROOT
HKU=HKEY_USERS
HKDD=HKEY_DYN_DATA (Windows 9x only)
HKPD=HKEY_PERFORMANCE_DATA (Windows NT, 2000, XP)
```

If you are working on a network and have access privileges to another machine, you can precede the registry path with a UNC network path to the machine name.

```
'rk' □wi 'xKeyName' '\\Wprichizeveb-2k\HKCR\WebPost.CWebPostWizard'
'rk' □wi 'XGetValue' 0
Web Publishing Wizard
```

Syntax:

You reference this property without an argument; it returns a character vector. You set it with a character vector giving the path through the Registry to the key you want to have as the current key.

Remarks:

The current key is the focus of RegKey; most other properties and methods apply to this key without explicitly naming it in each statement. Each time it is set, the system opens the key and creates a new handle.

A key can have named subkeys and/or named "Registry Values"; a Registry value consists of three parts: value-name, value-type, and data-value (its content)

xSubkeys**Description:**

This property is a read-only list of all the subkeys of the current key.

Syntax:

You reference this property without an argument; it returns a nested vector of character vectors with the last part of the key-name of all first-level subkeys of the current key.

xUnicodeRegistry**Description:**

This property is a read-only flag that tells if underlying content of this registry is Unicode: 1=yes; 0=no

Syntax:

You reference this property without an argument.

Remarks:

You may need to query this flag if you are using the 'raw' value-type argument, or if you are using XSetTypedValue for strings, because Unicode uses two bytes per character.

xValueNames

Description:

This property is a read-only list of all the value-names of the current key.

Syntax:

You reference this property without an argument; it returns a nested vector of character vectors with the names of all values created on the current key.

Remarks:

This list may or may not include the unnamed default value that is automatically created by Windows on each key below the root keys. If the (Default) entry has a data-value assigned, the result of xValueNames includes an empty vector as one element. If no data-value is assigned, there is no such entry in this list. The default value placeholder shows up in the Windows registry as "(Default)" in the Name column, but it is not so named. You can reference or assign a data-value to the default using 0 or ' ' as the value-name.

xValueType

Description:

This property is a read-only code giving the value-type of the content of a value whose value-name you specify.

Syntax:

You reference this property with a character vector that is the value-name of the value for which you want the value-type. You can use 0 or ' ' for the value-name that appears in each key as (Default). The method returns a character vector with the name of the value-type if it is one of the five value-types handled by RegKey. Otherwise it returns a scalar integer with the code for the value-type.

Remarks:

The standard Windows Registry value-types are listed in the table below. You can create your own ValueTypes by assigning an arbitrary integer number; in this case, RegKey returns the number assigned.

Name	Code	Windows Name	Nature of the data-value
	0	REG_NONE	
string	1	REG_SZ	Character vector
expandstring	2	REG_EXPAND_SZ	Character vector with embedded environment variables (see note 1)
binary	3	REG_BINARY	Hexadecimal bytes
int	4	REG_DWORD	Four-byte integers
	5	REG_DWORD_BIG_ENDIAN	(see note 2)
	6	REG_LINK	(see note 3)
multistring	7	REG_MULTI_SZ	Nested vector of character vectors
	8	REG_RESOURCE_LIST	
	9	REG_FULL_RESOURCE_DESCRIPTOR	
	10	REG_RESOURCE_REQUIREMENTS_LIST	

Note 1: An expand string has placeholders for environment variables. The `XGetTypedValue` method returns the stored value; the `XGetValue` method returns the expanded string.

Note 2: MS documentation defines value-type 5 as shown above; the Windows 2000/XP Registry Editor stupidly labels this as `REG_DWORD_LITTLE_ENDIAN`; since you enter the data as raw hexadecimal bytes, it could be anything, but value-type 4, `int`, is really little-endian.

Note 3: MS documentation says: "A Unicode symbolic link. Used internally; applications should not use this type." MS documentation also says: "Indicates a symbolic link between system or application data and a registry value. You can use Unicode characters in a `REG_LINK` entry."

Note 4: In the registry, a multistring contains multiple null-delimited strings ending with a double null; in APL+Win, it is returned as a nested vector.

Methods Reference

XCreateSubkey

Effect:

This method creates a subkey of the current key and returns a creation code and a pointer. You can use the pointer as the argument to a low-level call to the Registry within the same callback, or you can redirect the output to the name of a new APL+Win object. If you use the redirection syntax, the method returns only the code. The values of the code are: 1=A new key was created; 2=An existing key was opened.

Syntax:

You invoke this method using a character vector argument with the name of the subkey you want to create.

```
'rk' □wi 'XCreateSubkey' 'newKeyName'
19941704 1
'rk' □wi 'XCreateSubkey>rk3' 'newKeyName'
2
```

Remarks:

When you use the redirection syntax, you can use the name of the APL object as the left argument to any □wi statement in RegKey. Initially, the current key is the name you specified to the method.

XDeleteSubkey

Effect:

This method deletes a named subkey of the current key and the subkey's values; depending on the argument you specify, it deletes the subkey only if it has no descendant sub-subkeys, or it attempts to delete all descendants.

Syntax:

You invoke this method with a character vector that is the name of the key you want to delete and an integer Recurse flag. If you set Recurse=0, the subkey is deleted only if it has no descendants. If you set Recurse=1, the method attempts to delete the subkey and all its descendants. The user must have read/write access to all subkeys for this method to complete successfully with Recurse=1.

The method returns a zero if the subkey was deleted; otherwise, it returns an error code.

Remarks:

When you delete a subkey, you delete its name and all the values assigned to that name. The result may be non-zero if Recurse=1 and some descendants were not deleted. The result will pertain to the RegOpenKeyEx, RegEnumKeyEx, or RegDeleteKey call for the descendant that could not be deleted. If such an error occurs, the system stops processing and deletes no more subkeys.

XDeleteValue

Effect:

This method deletes a named value of the current key.

Syntax:

You invoke this method with a character vector that is the name of the value you want to delete.

Remarks:

When you delete a value, you delete its value-name and the data-value assigned to it. If you apply this method to the default value associated with a key when it has a data-value assigned, the system resets the

data-value so the Registry displays "(value not set)" but it does not get rid of the entry. If you invoke this method when the data-value is not set, it generates an error.

XGetTypedValue

Effect:

This method returns the data-value for a specified value of the current key. For a named value-type, it returns the correspondingly interpreted value; for any value-type, it can return the raw byte values.

Syntax:

You invoke this method with a character vector that is the name of the value you want to reference, a character vector that is the name of the value-type or 'raw', and, optionally, a default datum to be returned if the named value is not present.

For the first argument, use 0 or ' ' for the value-name that appears in each key as (Default).

The valid values for the second argument are: 'binary', 'int', 'string', 'expandstring', 'multistring', and 'raw'.

If you omit the optional argument and the named value is not present, the method signals an error.

When the data-value is a one of the named value-types, the method returns an integer for value-type int. For the other named value types, it returns an appropriate character vector. The character vector is:

- the unexpanded string for the expandstring value-type,
- a simple character vector for string,
- a nested vector of character vectors for multistring,
- a character vector with the $\square av$ APL representation of the hexadecimal data-value content for binary.

If the specified value is present but its value-type does not match the specified value-type, the method signals an error.

For any value-type, you can specify the character vector 'raw' in lieu of a type. If the named value is present, the method returns a two-element vector consisting of an integer code for the value-type (see the `xValueType` property for the list of codes) and a character vector with the $\square av$ APL representation of the hexadecimal data-value content of the raw bytes of the data-value.

Remarks:

When you want to use the result of specifying 'binary' or 'raw', capture the character vector rather than displaying it in the session and copying it; the display font is not a one-to-one mapping with $\square av$. You can use the indices of the result for an accurate (decimal) representation of the hexadecimal bytes.

For binary, use: ($\square av$ 1 *Result*)- $\square io$

For raw, use: ($\square av$ 1 2 $\triangleright$ *Result*)- $\square io$

```
'rk'  $\square wi$  'xGetTypedValue' 'GTVexample' 'raw' 1
3 APL+Win Version5
'rk'  $\square wi$  'xGetTypedValue' 'GTVexample' 'binary' 1
APL+Win Version5
decimal  $\leftarrow$  ( $\square av$  1 'rk'  $\square wi$  'xGetTypedValue' 'GTVexample' 'binary' 1)- $\square i$ 
o
base16 $\leftarrow$ 16 16  $\tau$  decimal
hex $\leftarrow$ '0123456789abcdef'
, hex[ $\square io$ +  $\&$  base16] , ' '
41 50 4c 2b 57 69 6e 20 56 65 72 73 69 6f 6e 35
```

XGetValue

Effect:

This method returns the data-value for a specified value-name of the current key for certain named value-types.

Syntax:

You invoke this method with a character vector that is the name of the value you want to reference and, optionally, a default datum to be returned if the named value is not present. For the first argument, use 0 or ' ' for the value-name that appears in each key as (Default).

If you omit the second argument and the named value is not present, the method signals an error. You can use a decimal value as the second argument, especially for the default value-name. Since the Registry does not recognize decimal values, if this method returns the decimal, no data-value exists for that value-name.

Remarks:

This method works for int and the three string value-types. Use the XGetTypedValue method for binary and all unnamed value-types.

This method returns the expanded string for the expandstring value-type. That is, the value of an environment variable replaces the name of the environment variable that is part of the string. You can retrieve the unexpanded string with XGetTypedValue.

```
'rk2' □wi 'XGetValue' 'myXvec'  
This expandstring shows C:\WINNT\system32\os2\dll;  
'rk2' □wi 'XGetTypedValue' 'myXvec' 'expandstring'  
This expandstring shows %Os2LibPath%
```

XOpenSubkey

Effect:

This method creates an ActiveX object that represents a specified subkey of the current key and returns a pointer to the object; if the specified subkey or its index does not exist, the method returns a zero.

Syntax:

You invoke this method using a character vector argument with the path, or a scalar integer with the origin-zero index, to the subkey you want to open.

Remarks:

You can use the pointer as the argument to a low-level call to the Registry within the same callback, or you can redirect the output to the name of a new APL+Win object.

```
'rk' □wi 'xKeyName' 'HKLM'  
'rk' □wi 'XOpenSubkey' 'Software\APLexpt'  
19940976  
( 'rk' □wi 'XOpenSubkey' 'Software\APLexpt' ) □wi 'XOpenSubkey > rk2' 2  
'rk2' □wi 'xKeyName'  
\HKLM\Software\APLexpt\Values
```

XSetTypedValue

Effect:

This method sets the data-value for a specified value-name of the current key for a specified value-type, providing the data-value is of the value-type you specify; optionally you can allow the system to change the value-type as you reset the data-value. If the value-name does not exist, it is created as the data-value is set.

Syntax:

You invoke this method with a character vector that is the name of the value you want to set, a character vector that is the name of the value-type, the appropriate data-value, and, optionally, a flag that specifies whether you want to be able to change the value-type.

For the first argument, use 0 or ' ' for the value-name that appears in each key as (Default).

The valid values for the second argument are: 'binary', 'int', 'string', 'expandstring', 'multistring', or 'raw'.

If you specify 'raw' for the second argument, the data-value argument must be a two-element vector with the integer code for the value-type (see `xValueType` for the list of codes) and the data.

The data for binary or the second element of data for raw can be any of the following:

- a numeric vector with decimal values in the range 0-255 (independent of `⌈io`),
- a vector of ActiveX typed values (see below) in the range 0-255 (independent of `⌈io`),
- a character vector with the `⌈av` APL representation (indices to `⌈av` dependent on `⌈io`) of the hexadecimal data-value content of the raw bytes of the data-value.

If you omit the optional argument or specify 0, and a data-value already exists, the value-type of the data-value you specify and the type argument must match the value-type of the existing data-value.

Remarks:

When you want to set raw data bytes using 'binary' or 'raw' with a character vector, specify the characters using `⌈av` rather than typing the characters directly or copying them from the session; the display font is not a one-to-one mapping with `⌈av`. You can use the (decimal) indices of `⌈av` for an accurate representation of the hexadecimal bytes.

Use: (`⌈av[DataValue+⌈IO]`) to represent (in decimals) the hexadecimal range 00-FF

```
'rk2' ⌈wi 'xKeyName'
\HKLM\Software\APLxpt\Values
indices←⌈av ⌈ 'APL2000.RegKey'
indices
66 81 77 51 49 49 49 47 83 102 104 76 102 122  A Note ⌈io+1
'rk2' ⌈wi 'XSetTypedValue' 'STVexample' 'raw' (3 (⌈av[indices])) 0
'rk2' ⌈wi 'xValueType' 'STVexample'
binary
```

You can set any Registry value-type using the 'raw' argument and the appropriate integer value-type code; however, you must be careful to define the bytes properly. The `REG_DWORD(int)` value-type requires four bytes to specify one integer value, and each character in a Unicode registry entry requires two bytes.

```
'rk' ⌈wi 'XSetTypedValue' 'STVexclaim' 'raw' (1 (,⌈exclaim ,''⌈av[⌈io]))
```

As a second option for defining the data-value, you can set raw bytes or binary by specifying an integer vector of values in the range 0-255.

As a third option you can set raw bytes or binary by specifying a vector of ActiveX typed values using any one of the types "Long" (4-byte integer, APL scalar code 3, array code 8195), "Integer" (2-byte integer, APL scalar code 2, array code 8194), or "Byte" (1-byte not-necessarily character, APL scalar code 17, array code 8209).

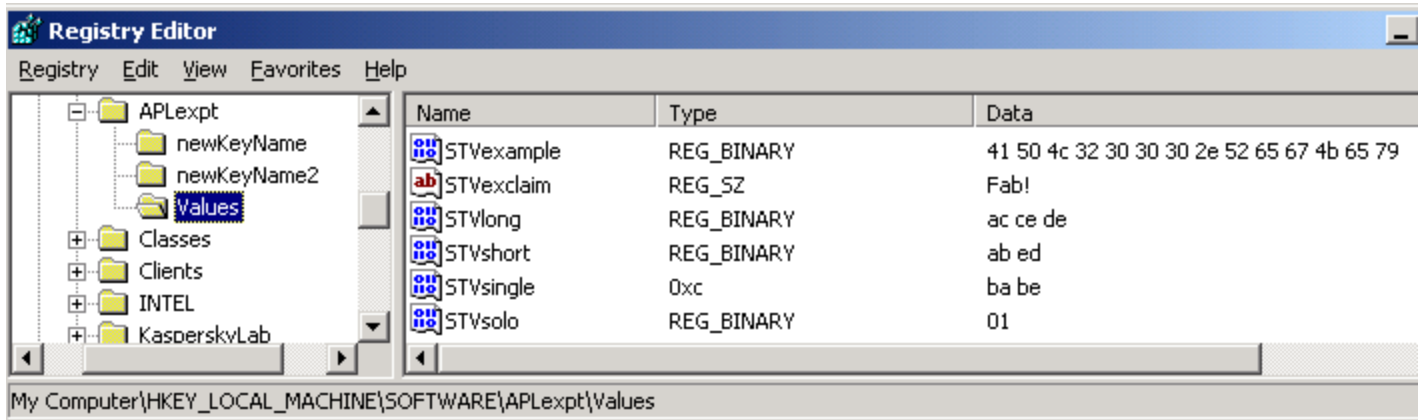
If you use these values, you most likely would get them from an ActiveX object; you can create such values using the VT method on the system object.

```
longint←'#' ⎕wi 'VT' (172 206 222) 3
longint
VT 8195 172 206 222
shortint←'#' ⎕wi 'VT' (171 237) 2
shortint
VT 8194 171 237

singlebyte←'#' ⎕wi 'VT' (⎕av[⎕io+186 190]) 17
singlebyte
VT 8209 1+
'rk' ⎕wi 'XSetTypedValue' 'STVlong' 'raw' (3 longint)
'rk' ⎕wi 'XSetTypedValue' 'STVshort' 'binary' shortint
'rk' ⎕wi 'XSetTypedValue' 'STVsingle' 'raw' (12 singlebyte)
```

Note also that these typed values must be vectors. If you want a single byte in the registry, and you want to use a typed value, you must use the four-digit code to type it as an array.

```
nogood←'#' ⎕wi 'VT' 1 2
okay ← '#' ⎕wi 'VT' 1 8194
nogood okay
VT 2 1 VT 8194 1
'rk' ⎕wi 'XSetTypedValue' 'STVsolo' 'binary' nogood
⎕WI ERROR: APL2000.RegKey exception 80004005 Nonce error: conversion not .
. .
'rk' ⎕wi 'XSetTypedValue' 'STVsolo' 'binary' nogood
^
'rk' ⎕wi 'XSetTypedValue' 'STVsolo' 'binary' okay
```



XSetValue

Effect:

This method sets the data-value for a specified value-name of the current key for certain value-types. If a data-value already exists, the value-type of the data-value you specify must match the existing value-type. If

the value-name does not exist, it is created as the data-value is set. The value-types that are allowed by this method are int, string, and multistring.

Syntax:

You invoke this method with a character vector that is the name of the value you want to set and the appropriate data-value, a scalar integer, a character vector, or a nested vector of character vectors.

For the first argument, use 0 or ' ' for the value-name that appears in each key as (Default).

Remarks:

You cannot set an expandstring with this method, because the APL system does not distinguish a character vector containing the name of an environment variable from any other character vector.

Event Reference

onXKeyChange

Trigger:

This event is triggered when certain settings change on the current key; optionally it can be triggered by the same type of change on a subkey to any level; the conditions of change are controlled by the settings of the xChangeName, xChangeValue, and xChangeAttr properties.

Behavior:

If xChangeValue=1, setting, resetting, or deleting a data-value on the current key triggers the event handler; if xChangeValue=2, a data-value change on a subkey descendant also triggers the event handler.

If xChangeName=1, setting or deleting the name of a first-level subkey of the current key or deleting the current key triggers the event handler; if xChangeName=2, a name change on a lower-level subkey descendant also triggers the event handler.

If xChangeAttr=1, a change in certain attributes of the current key triggers the event; if xChangeAttr=2, changing the attributes on a subkey descendant also triggers the event handler.

Syntax:

$\square$ warg is a two-element vector: the first element contains the name of the property (without the x prefix) whose setting allowed the event handler to fire; the second element contains the property setting (1 or 2).

Remarks:

Note that when the second element of $\square$ warg is 2, it does not tell you whether the change occurred on a subkey; it tells only that the property setting extends the triggering conditions to all subkeys.

Chapter 11: Calling Dynamic Link Libraries

APL+Win provides two facilities for calling a Dynamic Link Library (DLL). The system function `□na` allows you to call non-APL routines from APL+Win as if the routines were APL user-defined functions. These routines can be ones you write yourself, Windows system libraries, or other software packaged in a DLL. The system function `□wcall` also is used for calling dynamic link libraries, including the wide variety of useful third-party Windows software that is packaged in DLL format. Using the low-level Windows interface provided in `□wcall` and `□na`, you can call into Windows libraries at the same level as a C programmer.

The system functions `□na` and `□wcall` each provide a low-level interface from APL+Win to the Windows application programming interface (API) environment. Their uses overlap in that you can use either to make a subroutine call into a DLL. There are several significant distinctions.

- The `□na` interface uses a separate associated APL function for each entry point to be called, while `□wcall` takes the name of the API function as an item of its argument.
- `□na` provides only a call-and-return protocol, so the only action possible is to make a subroutine call. `□wcall` provides call-and-return, but it also is capable of making callbacks into APL, similar to the event handler facility of `□wi`. This provides a way for you to turn control over to an external code library and make callbacks into your APL workspace in response to Windows messages.
- `□na` creates an associated APL function that is a wrapper for one DLL-resident routine. You provide the information about the syntax of the routine in the argument to `□na`, and the associated function stores this information. `□wcall`, on the other hand, relies on syntax descriptions in `.INI` / `.ADF` files that APL loads at startup.

If the information already exists in these files, `□wcall` may be easier to use. If you need to provide the information, the initial overhead may influence you to use `□na`. The `.ADF` file provided with APL+Win contains some entry points that are used by functions in utility workspaces. You can add definitions to the `.ADF` file to access any Windows routine.

- The `□wcall` interface allows APL code to use names for constants, in the manner of those provided by C header files; you must convert constants to numerics for `□na`.

To use either `□na` or `□wcall`, it helps if you have a comfortable knowledge of C programming and how the Windows system delivers messages to programs. External libraries for Windows are documented in terms of their C language interface, and you should have this documentation for any DLL that you intend to call. You can find documentation on the Win32 API in the Microsoft Win32 Software Development Kit. It is also helpful to read any of the books on Windows programming for C programmers. Using `□wcall` or `□na` to bridge to Windows requires that you specify in APL the same kind of details as a C programmer writing Windows software. This chapter contains these sections.

“Using the Name Association Function” explains how you create and use an associated function, which is the function created by `□na` that calls an external routine in a DLL. You also learn more about the arguments and results of associated functions.

“The Windows Call Function” details the syntax for `□wcall`, explains how to interact with the Windows API, and explains filters and callbacks. The section also provides examples of various calls, options, and specifications.

“Argument Conversion for the Windows Call Function” describes how APL+Win converts the parameters and results of the Windows function calls specified in the `.INI` and `.ADF` files.

Using the Name Association Function

This section refers to a program written in a language other than APL that has been compiled and linked into a Windows DLL as an external routine. This chapter assumes that you already know how to write, compile, and link programs in other languages, or that you have them available.

`⌈na` creates an associated function in the workspace that defines an interface between APL+Win and an external routine. You can use `⌈na` to make utility routines available to APL applications. External routines may be useful runtime library routines or fast implementations of algorithms that are hard to express efficiently in APL. The associated function behaves like a user-defined, locked, monadic function in the workspace. You can save and load it with the workspace or copy it into another workspace.

When you use an associated function, APL+Win transfers control to the external routine and supplies parameters in the form that the external routine expects. The external routine then behaves as if it had been called from its native environment. When the external routine finishes execution, it passes any results back to APL+Win through the associated function.

Creating and Using Associated Functions

To access the routines in a DLL, you must create an associated function for each routine in the library you plan to call. The system function `⌈na` creates this function using a specific definition you supply. The right argument to an associated function is a vector; typically, a nested array. Each item in the vector corresponds to one of the routine's parameters.

The explicit result is also a vector; typically, a nested array. The first item is the routine's explicit result (if defined), and the remaining elements are the output arrays, if any (arguments marked with the left arrow in the parameter list). The routine may modify these arguments.

Syntax of `⌈na`

`⌈na` has three possible syntax forms:

```
status ← 'defn'      ⌈na 'name'
status ← ' '         ⌈na 'name'
defn   ←             ⌈na 'name'
```

status

The explicit result. A 1 indicates the system created the function successfully; a 0 indicates the system could not create the function.

name

The name you want to use for the associated function.

defn

The definition of the external routine, including the language, type of result, name of the library, name of the routine, and the parameters expected by the routine. You can place `⌈tcnl` characters between fields. You can include trailing comments in each line of the definition by preceding the comment with a lamp (`⌈`) symbol.

Using `⌈na` dyadically with a left argument of `defn` creates an associated function. To find out if an associated function exists, you can use `⌈na` with an empty left argument:

```
status ← ' ' ⌈na 'name'
```

The result is either a 1 or a 0. A 1 indicates that the right argument is the name of an associated function in the active workspace; a 0 indicates that it is not. You can also use `⊞na` monadically to find out the definition used to create the associated function:

```
defn ← ⊞na 'name'
```

Constructing a Definition

The definition of the library routine has the structure:

```
'language result← library.routine (parm1,parm2,...)'
```

language

Defines the APL linkage scheme that matches the calling convention of the language and compiler in which the DLL-based subroutine was written. The value **DLL** specifies that you want to call a Windows subroutine in a 32-bit DLL. The subroutine can use a linkage with a fixed parameter list or a variable number of arguments. It must use either the `_cdecl` or `_stdcall` Microsoft-specific keywords, or their equivalents in other compilers; `_fastcall` is not supported.

result←

(Optional) The result (if any) the external routine returns. The result uses the same datatype notation as *parm*. If the result has exactly one item, you can use a disclose symbol (`⊃`) as a prefix to have automatically disclose the result.

library

The name of a Win32 dynamic link library with extension `.DLL`. You do not supply the path or extension. APL+Win searches for the DLL by looking in the directory where APL is installed, in `\WINDOWS\SYSTEM`, `\WINDOWS` and each directory in the `DOS PATH`. The order of search may vary by platform.

routine

The name of the routine in the library. The distinction between uppercase and lowercase is important.

Note: Some Windows library functions that have character arguments have two forms: they use 'A' and 'W' suffixes for 1-byte and 2-byte characters. You must supply the 'A' or 'W' suffix in the name of *routine*.

When specifying the *routine* portion of *defn*, the spelling of the routine name must match the spelling as it is exported by the DLL. Some compilers may modify the name as given in the original C source code.

parm1, parm2, ...

The form of arguments to be passed to the routine. Enclose the list of argument specifications in parentheses and separate them with commas. If the routine does not require parameters, you must still supply an empty list enclosed in parentheses. Each *parm* describes the datatype of an argument, how it is passed, and whether the routine modifies it. If a *parm* is marked with a left arrow (`←`) as being modifiable, it is an output array that the routine returns as part of its explicit result, regardless of whether the routine actually modified it. If the argument consists of exactly one item, you can precede the specification with an enclose symbol (`⊃`) to have `⊞na` automatically enclose the item.

Datatypes Recognized by `⊞na`

Datatypes are declared explicitly in most compiled languages. Routines work correctly only if you pass parameters of the declared type. When you describe the result and parameters of the external routine in the

syntax of `⌈na`, you must specify the correct datatypes and tell `⌈na` how to pass the arguments. The datatypes that `⌈na` recognizes include:

Datatypes for `⌈na`

Datatype	Description
B1	Boolean (1 bit per element in byte-wise order; 8 bits per byte, most significant bit first).
C1	Character (1 byte per element) or C parameter declared as <code>char</code> (unsigned or signed).
F4	Four-byte single precision IEEE floating point or C parameter declared as <code>float</code> .
F8	Eight-byte double precision IEEE floating point or C parameter declared as <code>double</code> .
G0	General object. A variable in the internal form used by APL. Passed by reference with the address of the variable in the workspace.
I2	Two-byte signed integer or C parameter declared as <code>short</code> .
I4	Four-byte signed integer or C parameter declared as <code>long</code> .
U2	Two-byte unsigned integer or C parameter declared as <code>unsigned short</code> .
U4	Four-byte unsigned integer or C parameter declared as <code>unsigned long</code> .

Note: APL+Win does not use two-byte integers in ordinary variables in the workspace. The system creates such values temporarily to provide them to the associated function, but it converts any returned results back to four-byte integers.

Coercing the Data

When you call an associated function, APL+Win ensures that each of the arrays being passed has the appropriate datatype. If necessary, the system changes the datatype of an array to make it conform to that expected. This process is called coercion. The system coerces datatypes by making a copy of the data and changing the internal representation in such a way that the value stays the same. It then passes the copy to the external routine.

For example, if the variable `A` contains a Boolean array, each number uses one bit per element in the computer.

```

      A
1 0 1 1

```

Suppose the external routine you want to use works with four-byte integer data; that is, each number uses four bytes per element in the computer. Its parameter must be described in the `⌈na` argument as `★I4`. If this routine were to receive `A` in Boolean form as an argument, it would fail because it expects each element of the array to be in 32-bit integer form. When you call the associated function, APL+Win makes an integer copy of `A` by changing each one-bit element into a 32-bit 0 or 1, and passes the copy to the external routine.

A `DOMAIN ERROR` occurs if the system cannot coerce an array to the correct datatype without changing its values. The system can always promote Boolean and integer data to floating point, but it can convert floating-point values to integer only if they are exact integers or within APL+Win's system fuzz tolerance of whole integers. (See the System Limits and Characteristics appendix in the [User Manual](#).)

Specifying Parameters in `⌈na`

In addition to specifying the datatypes of the parameters, you must specify how to pass them to the external routine. Compiled languages usually provide several distinct methods for passing parameters, such as by name, by value, and by reference. Pass-by-value and pass-by-reference are the most common. Pass-by-value is used to pass a single scalar element. Pass-by-reference is used to pass the memory address of the data rather than the actual data. This method is appropriate for passing arrays of arbitrary size, because only the starting address is passed. Pass-by-reference also allows a subroutine to modify the values stored at that address. `⌈na` allows you to pass scalars by value. You must pass arrays of more than one element by

reference. To indicate how you want APL to pass the arguments, add a prefix to the datatype. The possible prefixes are:

Prefixes for Parameter Passing

Parameter	Description
=	Default for most datatypes. Passed by value. The actual value is passed to the external routine. Only one-element, simple arrays can be passed by value.
★	Passed by reference. The machine address of the value is passed to the external routine. (Default for datatype G0, which cannot be passed by value.)
⌘	Passed by reference in column-major order as expected by FORTRAN; must be used instead of ★. For arrays with ranks > 2, only the last two coordinates are transposed.
⊗	Forced coercion. Forces a modifiable copy of the argument item to be made. The modified argument is not returned in the result.

APL represents arrays in workspaces with data structures that contain a variety of descriptive and housekeeping data about the array, such as the shape and datatype, in addition to the actual data. Generally, only the data items themselves are meaningful to the external routine. If APL passes the argument by reference, it passes the address of the first data value of the corresponding item of the argument. For example, if a routine has two arguments, both passed by reference (★I4, ★C1), it requires a two-item nested array as its argument:

```
ARG ← (99 100 101 102) ('ABCDEF')
```

APL passes the external routine two arguments: the address in the workspace of the value 99 in 1⊃ARG, and the address of the 'A' in 2⊃ARG.

Examples Using ⌘na to Create an Associated Function

The `GetSystemMetrics` function is defined in the Windows API; it retrieves various system metrics (widths and heights of display elements) and system configuration settings. The first block of code shows the definition; it comes from the `winuser.h` file distributed by Microsoft as part of your system. The line below that block is the APL `⌘na` call that creates the associated function.

```
WINUSERAPI
int
WINAPI
GetSystemMetrics(
    int nIndex);

'DLL I4← USER32.GetSystemMetrics(I4)' ⌘na 'FOO_GSM'
1
```

The uses of "int" in the definition correspond to the I4 datatype. The first is the result, and the second is the argument. The `nIndex` parameter specifies which system metric or configuration setting to retrieve. You could make the function name identical to the called function.

This example describes how you use `⌘na` to create an associated function that calls a routine you may have written yourself and compiled into a 32-bit DLL. The sample external routine shown below is a Microsoft C routine, named `isum`, that adds a vector of numbers and returns the sum.

```
#define DllExport32 __declspec(dllexport)
DllExport32 int isum ( input, length )
int *input;          /*address of first number in vector */
int length;          /*length of the vector */
{
    int i;
    int sum = 0;
```

```

    for (i=0; i<length; i++ ) {
        sum += input[i];
    }
    return sum;
}

```

This routine requires two parameters: `input` and `length`. The parameter named `input` contains the machine address of the first number in the vector, which is passed by reference. The parameter named `length` contains the length of the vector, which is passed by value. Before you can run `isum`, you must compile the routine and link it into a DLL. The distribution disks contain an example DLL named `DLLEXP32`, which contains the function `isum` shown above. The example below creates an associated function named `SUMIT`. It passes two parameters: a four-byte integer, passed by reference; and a four-byte integer, passed by value. The result is a four-byte integer.

```

'DLL I4← DLLEXP32.isum(★I4,I4)' ⌵na 'SUMIT'
1

```

Running Associated Functions

Once you create an associated function, you can run the non-APL function from APL; for example, calling the `GetSystemMetrics` function:

```

FOO_GSM 0  A nIndex 0 gets the screen width in pixels
1024
FOO_GSM 1  A nIndex 1 gets the screen height in pixels
768

```

Running the Microsoft C routine detailed above:

```

X←14
SUMIT (X) (ρX)
10

```

When you call an external routine, APL+Win first checks the right argument to make sure that it is a scalar or vector whose length matches the number of parameters that you specified with `⌵na`. It then locates the entry point in the library. If the library does not exist, APL produces a `MODULE NOT FOUND` error. If the library does not contain the entry point, APL produces an `ENTRY POINT NOT FOUND` error.

Next, the system creates temporary copies of the parameters marked as modifiable and any array where the data in the APL array are not already of the type expected by the routine but can be coerced to that type. It then passes the parameter list; parameters are passed as specified in the associated function. Finally, it calls the external routine. When the external routine returns, it builds an APL variable and returns it as the explicit result of the associated function.

Explicit Results and Output Arrays

The explicit result, if any, of an associated function is a vector, typically a nested array. The number of items depends on the description used with `⌵na` to define the associated function. If the routine returns a result, that result is the first item of the array. The remaining items of the explicit result consist of any of the arrays marked as modifiable output arrays.

APL+Win makes temporary copies of all the parameters marked as modifiable, and passes the copies to the routine. An external routine can safely modify the values in these copies without unexpected side effects. When the external routine completes, APL forms the associated function's result from the explicit result of the routine followed by the parameters marked as output arrays. If any output arrays are passed by reference, they are returned as enclosed items of the result. If you do not specify a result or output arrays, the associated function returns an empty numeric matrix.

Output arrays allow an external routine to return one or more entire arrays to APL, rather than a single scalar. For example, the following external routine reverses the text in a character vector.

```
DllExport32 void REVTXT(char *s1, char *s2, int len)
{
    int i;
    s1 +=len;
    for (i=0; i<len;i++){
        *s2++=*--s1;
    }
}
```

It expects three parameters — the addresses of the input vector and the output vector, and the length.

```
'DLL DLLEXP32.REVTXT(*C1,*C1←,I4)' ⌐na 'REVERSE'
1
```

The second parameter is marked as modifiable. APL creates a temporary copy and passes the copy to the external routine. When APL calls REVERSE, the second parameter must be the array that it will modify to contain the reversed character vector. The simplest way to call this specific routine is to pass the same array as both arguments:

```
TXTR ← 'THE INPUT UNMODIFIED'
TXTR ← REVERSE (TXTR)(TXTR)(ρTXTR)
TXTR
DEIFIDOMNU TUPNI EHT
TXTR
THE INPUT UNMODIFIED
```

Because the second argument to the routine is marked as an output array, a copy of TXTR is created and passed. REVERSE then places the output in this copy. This copy is the same size as TXTR, which is exactly what the routine requires.

Alternatively, you can create the output array explicitly:

```
TXTR ← REVERSE (TXTR) ((ρTXTR)ρ' ') (ρTXTR)
```

When the second parameter is not specified as modifiable, a serious problem occurs. APL passes the address of the original array rather than making a copy.

```
'DLL ⤵DLLEXP32.REVTXT(*C1,*C1,I4)' ⌐na 'REVERSE'
1
SPARETXT ← TXTR ← 'THE INPUT UNMODIFIED'
REVERSE (TXTR)(TXTR)(ρTXTR)
SPARETXT
DEIFIDOMNU TUPNI EHT
TXTR
DEIFIDOMNU TUPNI EHT
```

Since no copy was made of the input variable, TXTR and SPARETXT refer to the same array in the workspace and the modifications done by REVERSE are to this shared array and possibly other arrays in the workspace.

Caution: Be sure to indicate modifiable arrays when you define associated functions. APL+Win uses an array representation strategy based on reference counts, which means that a single array can have multiple references; for example:

```
A←B←C←ι1000
```

The array ι1000 is created only once. The variables A, B, and C point to that one array. If you pass A by reference and do not define it as an output variable, an external routine could modify the contents of A. That also modifies B and C.

As a rule, you should never modify the value of an array unless you mark it as modifiable or you are certain that it is referenced only as the variable you intend to modify. If you fail to follow this guideline, you can cause system crashes, data loss, or other unpredictable behavior. The `*qualifier` lets you pass a copy of an array that you can modify but that is not returned in the result.

As the APL+Win System evolves, the internal array representation strategies may change and invalidate your code if you modify arrays that are not marked as modifiable.

This example shows the definition of `CharUpper` as defined in the Windows API. The `CharUpper` function converts a character string to uppercase.

```
WINUSERAPI LPSTR WINAPI CharUpperA(LPSTR lpsz);
WINUSERAPI LPWSTR WINAPI CharUpperW(LPWSTR lpsz);

#ifdef UNICODE
#define CharUpper CharUpperW
#else
#define CharUpper CharUpperA
#endif // !UNICODE
```

Note that Windows actually defines two functions: `CharUpperA` (ASCII version) and `CharUpperW` (wide or double-byte version). It then defines `CharUpper` to point to one of the other two functions. Most API documentation will only refer to the function with no suffix. In APL+Win, you should generally use the version with the “A” suffix.

`CharUpper` takes a single argument of a character string which the call will modify and returns no explicit result. We mark the argument with the enclose symbol (`<`) so that we will not need to enclose it when we make the call. Since the argument is marked as modifiable, it will be returned as part of the result. We use the disclose symbol (`>`) so that it comes back as a simple vector instead of nested.

```
'DLL >USER32.CharUpperA(<★C1<)' ⍵na 'CharUpper'
1
CharUpper 'Text string'
TEXT STRING
```

Using the Windows Call Function

You can also use the system function `⍵wcall` to access a function in a DLL; in particular, you can use `⍵wcall` to access the Windows API. You can use `⍵wcall` to call virtual functions in a custom COM interface or call functions using dynamically obtained function pointers. In addition, `⍵wcall` is capable of defining callbacks into APL, similar to the event-handler facility of `⍵wi`.

Using `⍵wcall`

The argument of `⍵wcall` contains the name and arguments of a function to be called. Its syntax is

```
result ← ⍵wcall 'function' arg1 arg2 ...
```

where `'function'` is the name of an entry point in a DLL, spelled exactly as it would be used in a C call. Note that uppercase and lowercase are distinct in function names. For example, the Windows function `GetActiveWindow()` does not take any arguments, so `⍵wcall` has only the function name in its argument. It returns the window handle of the active window.

```
h ← ⍵wcall 'GetActiveWindow'
```

The Windows function `Ellipse()`, on the other hand, requires several parameters. Its syntax, as described in the *Windows Programmer's Reference*, is:

```
BOOL Ellipse(hdc,nLeftRect,nTopRect,nRightRect,nBottomRect)
HDC hdc; /* handle of device context */
int nLeftRect; /* x-coordinate of upper-left corner bounding rectangle */
int nTopRect; /* y-coordinate of upper-left corner bounding rectangle */
int nRightRect; /* x-coordinate of lower-right corner bounding rectangle */
int nBottomRect; /* y-coordinate of lower-right corner bounding rectangle */
```

You could call it with an APL statement such as the following. (Do not try this unless you have already obtained a device context for it.)

```
rc ← ⌐wcall 'Ellipse' hdc 10 10 300 200
```

This passes five parameters to `Ellipse()`, which correspond to the five items following 'Ellipse' in the argument. All of the argument are scalars; all five are passed by value as `Ellipse()` expects.

When a routine requires a string or other block of variable-sized data as a parameter, you supply an enclosed APL array as the argument. For example, one of the parameters of the Windows function `TextOut()` is a string of text you want to display.

```
BOOL TextOut(hdc, nXstart, nYstart, lpszString, cbString)
HDC hdc; /* handle of device context */
int nXstart; /* x-coordinate of starting position */
int nYstart; /* y-coordinate of starting position */
LPCSTR lpszString /* address of string */
int cbString /* number of bytes in string */
```

You could call this function (again, `hdc` must be a valid device context) with the statement:

```
rc ← ⌐wcall 'TextOut' hdc 40 40 'Hello' 5
```

Using Names for Constants

A C programmer who is developing an application using the Windows API uses names for constants that have arbitrary values but very specific meanings. For example, a C programmer would use the macro `WM_PAINT` in code that requires the special number that identifies the windows paint message. This special number happens to be 15, but a C programmer would never write code that explicitly uses this number. When the source file is compiled, the C preprocessor looks up the constant definitions in one of the header files included by `windows.h` and substitutes 15 for `WM_PAINT`.

The `⌐wcall` interface provides a similar capability, so that your APL code can contain the same names for these constants that a C programmer would use. Instead of using `windows.h`, APL+Win uses the [Constant] section of the `APLW.ADF` file to associate a name with each Windows constant. These constants are expressed by using character vectors in `⌐wcall` arguments for parameters that Windows expects to be numeric scalars. `⌐wcall` automatically substitutes the correct numeric parameter in place of the character value. For example, the following APL statement posts a `WM_PAINT` message to a window whose window handle is stored in `hwndGraph`:

```
⌐wcall 'PostMessage' hwndGraph 'WM_PAINT' 0 0
```

The APL utility function `W_C` in the `WINAPI` workspace returns the number represented by each Windows constant. For example,

```
W_C 'WM_PAINT'
15
```


When several named constants are written together in a single character vector, the system converts them to numbers and combines them with a logical OR operation. For example, a C programmer might select the flags `CS_HREDRAW` and `CS_VREDRAW` in defining a window class with a statement like the following:

```
flag = CS_HREDRAW | CS_VREDRAW;
```

In an argument to `⌵wcall`, the logical OR of these two constants would be expressed by putting the two names in the same character vector, separated by one or more spaces.

```
flag ← 'CS_HREDRAW CS_VREDRAW'
```

The APL utility `W_C` evaluates a list of one or more character vectors and returns the numeric values that they encode. For example:

```
W_C 'CS_HREDRAW' 'CS_VREDRAW'
1 2
W_C 'CS_HREDRAW CD_VREDRAW'
3
```

As a comparison with the Windows API call shown earlier using `⌵na`, you could make the same call using the defined constant with either of the following statements:

```
⌵wcall 'GetSystemMetrics' 'SM_CXSCREEN'
1024
⌵wcall 'GetSystemMetrics' (W_C 'SM_CXSCREEN')
1024
```

However, the first call is more convenient and more efficient. Note that you can also use the `W_C` function to look up constants when calling a function via `⌵na`. However, `⌵na` does not directly accept constant names as arguments.

Startup Files for `⌵wcall` (.ADF and .INI)

The syntax of each function you want to call using `⌵wcall` must be defined to APL at the start of the APL session, or, if you change the `.INI` file dynamically, before the specified item is used by `⌵wcall`. The primary file used for this initialization is `APLW.ADF`, which APL+Win reads during initialization. The `.ADF` file by default is assumed to be in the same directory as the APL interpreter with the name *exename*.ADF, where *exename* is the name of the APL interpreter. If you rename the APL interpreter in a runtime-based application, and your application uses `⌵wcall`, you should provide a similarly named `.ADF` file in the same directory. You can also override this default behavior by placing the line `ADF=path\file` in the `[Config]` section of APL's `.INI` file.

You can query the location of the `.INI` or `.ADF` file using `⌵wcall 'W_IniFile'` or `'W_AdfFile'`. The system returns the full path and file name.

An `.ADF` file is a compiled version of the API descriptions found in a `.INI` file. The `.ADF` format is more compact and requires much less processing at startup time; it is produced by compiling one or more source `.INI` files. APL+Win provides the file `APLWADF.INI`, which is the source code for `APLW.ADF`.

The utility function `W_MakeADF` in the `WINAPI` workspace is used to compile `.INI` files into a `.ADF` file. If you want to extend `APLW.ADF`, for example, you create a `.INI` file that contains the entries you want to add, and then recompile `APLW.ADF` to merge your extensions with the `APLW.INI` file. Rename the resulting file `APLW.ADF`.

```
'NEWAPLW.ADF' W_MakeADF 'APLWADF.INI MYCODE.INI'
```

While you are developing the contents of a new `.INI` file, you can test new additions and modifications by putting them in the `APLW.INI` file (that is, the interpreter's startup file, *not* `APLWADF.INI`). `APLW.INI` can contain the same `[Call]`, `[Constant]`, `[Pattern]`, and `[Type]` sections as those in `APLW.ADF`, and any definitions there override those in the `.ADF` file.

The default `APLW.ADF` file contains an extensive but not exhaustively complete list of Win32 calls. The Win32 libraries contain two separate sets of functions that take character arguments or structures containing character arguments. Functions with the suffix 'A' expect characters to be passed as 8-bit ANSI characters, and a parallel set of functions with the 'W' suffix treat characters as 16-bit Unicode. `APLW.ADF` selects the 'A' variant, and aliases the name so your code does not need to supply the 'A' suffix. Functions that don't have any character arguments don't have either an 'A' or 'W' suffix. The `WINAPI` workspace contains a variable named `PortingHints` that contains further information useful for moving applications from earlier versions of APL★PLUS.

The primary challenge in using `⌵wcall`, and the primary purpose of this chapter, is to ensure the agreement among the arguments to the function call in APL, the specification of the syntax in the `.ADF` or `.INI` file, and the parameters required by the function in the DLL. Much of the material below is so similar it can be confusing. Specifications described for the `[Call]` section apply to APL function calls to a DLL; specifications described for the `[Pattern]` section apply to callbacks from Windows to APL. Specifications for the `[Type]` section describe abstractions of datatypes that can be used by any of the sections. Be aware that the square brackets have multiple meanings in the text below. They are used to designate sections of the startup files, show that elements of parameter or argument specifications are optional, and are literals in some of the argument syntax, for example when showing repeat count.

Defining Function Call Syntax [Call]

The `[Call]` section of `APLW.ADF` contains specifications for many of the functions of the Win32 API. Each function's syntax is described with a shorthand notation that APL uses to define the mapping between the arguments that your code uses for `⌵wcall` and the parameters that are passed to the Windows function. New functions for other entry points in other DLLs can be added, either to a `[Call]` section of `APLW.INI`, or to the `.ADF` file's source file (`APLWADF.INI`), which is then recompiled to build a new `APLW.ADF` file.

The general format for a function call specification is the following.

fname = *rtype* (*pspec*, *pspec* ...) *options*

fname

The name of the entry point in the DLL, as it would be used in the argument to `⌵wcall`.

rtype

The result type, which describes the datatype returned by the function. The *rtype* should be omitted for functions that don't return an explicit result (that is, functions declared with result type `void` in C/C++). Please note that some functions that actually do return an explicit result are declared in the `ADF` file without a result. For example, `GetClientRect=(HW,>RECT)`. This is done for historical compatibility with code that was originally written for the Win16 API which did not return an explicit result for the corresponding function. To avoid causing existing application code to stop working, the old definitions are retained rather than using the actual definitions for the Win32 API. In most such cases, the result type is `BOOL` and there is an alternate definition in the `ADF` file with a `B` suffix. For example, there is a definition for `GetClientRectB=B(HW,>RECT)` that includes an *rtype* declaration for the explicit result.

pspec

The parameter specification, which describes a formal parameter to the function. The specification provides information about the datatype of the parameter, whether it is passed by value or by reference, and whether it can be modified by the called routine and should therefore be returned in the `⌵wcall` result. The parentheses must always be included, even for functions that don't have any parameters. In this case, there will be nothing inside the parentheses. For example: `GetFocus=H()`.

options

Additional information about the way a call is processed.

Base Datatype Notation

The datatypes used by the functions include:

Built-in Datatypes for Win32 Functions

Datatype	C Datatype	Suitable APL data
C1	unsigned char	character data
I1	signed char	integer value -128 to 127
I2	signed short	integer value -32768 to 32767
I4	signed long	integer value $(-2 \star 31)$ to $(2 \star 31) - 1$
U1	unsigned char	integer value 0 to 255
U2	unsigned short	integer value 0 to 65535
U4	unsigned long	integer value 0 to $(2 \star 32) - 1$
HW	unsigned long	window handle 0 to $(2 \star 32) - 1$ (see Note)
H4	unsigned long	other handle 0 to $(2 \star 32) - 1$
P4	void	valid pointer 0 to $(2 \star 32) - 1$
F4	float	any numeric
F8	double	any numeric
X0	external	any APL array

Note: HW can be either a scalar integer or a two-element vector that contains a window handle followed by a control ID number. The two-element vector specifies a parent window handle followed by a child control identifier. The corresponding child window handle (obtained via an implicit call to `GetDlgItem(hwndParent, idChild)`) is passed as the argument in place of the two-element vector.

The .ADF file often uses one-character synonyms for these datatypes. The datatype synonyms are:

Built-in Datatype Synonyms for Win32 Functions

Synonym	Datatype	Name
C	C1	Char
T	I1	Tiny
S	I2	Short
L	I4	Long
I	I4	Int
Y	U1	Byte
W	U2	Word
D	U4	Dword
B	U4	Boolean (0 vs any non-zero)
U	U4	Unsigned
H	U4	Handle
F	F4	Float
E	F8	Extended
P	P4	Pointer
X	X0	External

Note: Four values changed in migrating from Win16 to Win32; the synonym I changed from I2 to I4 and B, U, and H changed from U2 to U4.

Parameter and Result Notation

The notation for describing a parameter or result for a Windows function takes the following form:

[~] [ptr] dtype [*repeat*] [*name*] [=value]

~

An optional prefix that indicates that the call made to Windows requires a parameter, but your APL code does not provide it in the argument to `□wcall`. `□wcall` generates a parameter to include in the DLL call, which is the *value* specified, if any, or zero otherwise. Parameters with this prefix are often used as placeholders for real arguments for which a zero (or other constant) value is appropriate. Placeholder arguments *must* be included in the call definition so that `□wi` can build an appropriately sized and initialized parameter list for the actual function call.

ptr

An optional prefix, either an asterisk (*) or greater-than sign (>), that indicates that the APL argument should be passed by reference rather than by value. In both cases, the argument array is copied into Windows memory, and a 32-bit pointer is passed. The > prefix is like the ← in `□na` arguments. It denotes an APL argument value that the called function can modify; for example, a buffer in which to write the result of a query. This array then becomes an item of the result returned from `□wcall`. The * prefix has the same meaning as with `□na`, a pointer to the data part of an array.

dtype

The base datatype for the parameter. This can be a type from the lists in the datatype tables above or the name of an entry defined in the [Type] section of the application initialization file.

[repeat]

An optional count (enclosed in brackets) can follow the datatype. The count indicates the number of times the base datatype should be used. Using a repeat count is functionally equivalent to using a named type defined in the [Type] section that has the specified number of *dtype* items enclosed in braces. Thus, `C[4]` is the same as `{C,C,C,C}`. However, note that a parameter list such as `(L,L,C,C)` is not the same as `(L,L,C[2])`. The first list specifies four parameters, while the second specifies three parameters, the last of which is a structure of two elements. The list `(L,L,C[2])` is the same as `(L,L,{C,C})`. See the "Defining Types" section below for a more detailed discussion of repeat counts.

name

An optional parameter name, separated from *dtype* by a least one space. This is for documentation purposes only; typically it is the name of the formal parameter in the C function header, to remind you of how items of the argument correspond to the routine's parameters.

=value

The optional default value suffix that is used as the parameter in the windows call. It can be a numeric value or the name of one or more constants defined in the [Constant] section of the file. If you specify the equal sign without any value, `□wcall` looks for a constant with the same name as the function. See the example of `EM_UNDO` in the next section.

Function Call Specification Options

You can use several options in the description of a function call:

LIB dllname

Specifies the name of the DLL that contains the function. If this option is not specified, the system searches for the function in the set of libraries specified by the `LIBS` parameter in the [Config] section of your `APLW.INI` file. If your `APLW.INI` file does not specify a `LIBS` parameter, the system searches the following

libraries: `USER32.DLL`, `GDI32.DLL`, `KERNEL32.DLL`, `COMDLG32.DLL`, and `LZ32.dll`, in the order listed. It is more efficient to explicitly specify the library since this avoids the overhead of searching through the set of `LIBS` DLLs and avoids ambiguity if the function appears in more than one of the `LIBS` DLLs.

The *dllname* can be a long file name but it cannot contain any spaces. The end of the *dllname* is determined by the first blank space following the name or the end of line. If the name of the DLL you need to specify contains spaces, you must use its short path name (which does not contain spaces).

PACK

Causes the arguments to be packed together rather than aligned on four-byte boundaries when building the parameter list. This allows packing of parameters narrower than four bytes into the same Windows parameter. Without this option, each argument is aligned on a four-byte boundary, regardless of its size.

ALIAS name

Indicates that the function specification is not an actual entry point but an alias for an entry point in the DLL. For example, the following specification makes it possible to use the name `EM_UNDO` to send an undo message to an edit control.

```
EM_UNDO=B(HW hwnd,~U=,~U,~D) ALIAS SendMessageA
```

This specification has the following meaning:

- `EM_UNDO` is an alias for the Windows API call `SendMessageA()`. Note the “A” suffix on the `SendMessage` function name. This causes the ASCII variant, rather than the UNICODE variant of the `SendMessage` function to be called. The `SendMessage` function itself is actually aliased to `SendMessageA` as well.
- APL passes only one argument to `SendMessageA()` in the argument to `□wcall`, the handle corresponding to the `hwnd` parameter of `SendMessageA()`. The other arguments do not need to be passed to `□wcall` because they are prefixed by the tilde character, “~”.
- When APL calls `SendMessageA()`, it evaluates the constant `EM_UNDO`, which is the name of a specific Windows message defined in the `[Constant]` section, and specifies that as the second parameter of `SendMessageA()`. This substitution is specified by the “=” without a value following it for the second parameter (`~U=`).
- Default values of 0 are provided for the third and fourth parameters. This works because the `EM_UNDO` message does not define any meaning for the *wParam* and *lParam* parameters of standard Windows message format.

You use this technique so you can use the following statement

```
□wcall 'EM_UNDO' hwnd
```

instead of having to code the following, which also works:

```
□wcall 'SendMessage' hwnd 'EM_UNDO' 0 0
```

VIRTUAL number

Indicates that the function specification is not an actual entry point but an entry in a Vtable and specifies which entry in the Vtable to use. When using `□wcall` to call virtual functions, you also pass an additional parameter (as the first parameter following the function name) of the pointer to the interface whose virtual function is being called. This pointer is used to locate the Vtable whose function is to be called and is also passed as an implicit “this” pointer to the called virtual function.

The `IMalloc` interface is used to allocate and manage memory. Its Vtable contains nine functions: `QueryInterface`, `AddRef`, `Release`, `Alloc`, `Realloc`, `Free`, `GetSize`, `DidAlloc`, `HeapMinimize`.

To use the `Free` function in the `IMalloc` Vtable, define it as:

```
IMalloc::Free=(P pv) VIRTUAL 5
```

This specification has the following meaning:

- `IMalloc::Free`, although the sixth entry, is index 5 (origin 0) in its Vtable. It is important to use the correct Vtable index. Using the wrong index will cause a different function to be called.
- Its argument is the pointer to be freed. However, when you use `␣wcall` to call the function, you must include an interface pointer argument, as the first argument, before the pointer to be freed.
- You could use any name for the function but the convention is to use the name of the interface, double colons, and then the function name.

You use this technique so you can use the following statements

```
A get an IMalloc interface pointer
IMalloc←↑1↓←␣wcall 'SHGetMalloc' (,0)
A note that IMalloc::Free takes an additional argument which is not
A specified in its definition - the pointer to the IMalloc interface
␣wcall 'IMalloc::Free' IMalloc p  A free the pointer "p"
A release the IMalloc pointer after you are done using it
A This is done using the built-in W_Release function
␣wcall 'W_Release' IMalloc
```

You can also use the `VIRTUAL` option without a Vtable index. When no number is specified, the first argument passed to `␣wcall` is treated as a function address. This enables you to call a function whose address is obtained dynamically. You pass a function pointer rather than an interface pointer as the first argument to the function. When this function is called, the function argument is NOT passed as an implicit argument. It is used only to determine the address of the function to be called. Only the actual arguments specified in the `[Call]` specification are passed to the function.

The following example shows how you can cause an ActiveX object to register itself. If the following is specified in the ADF file:

```
[Call]
DLLRegisterServer=L() VIRTUAL
```

Then you could call the `DLLRegisterServer` function in the ActiveX module with the following code:

```
hlib ← ␣wcall 'LoadLibrary' 'path\SomeActiveX.dll'
pf ← ␣wcall 'GetProcAddress' hlib 'DLLRegisterServer'
hr ← ␣wcall 'DLLRegisterServer' pf
␣wcall 'FreeLibrary' hlib
```

Examples of Function Call Specifications

`CreateMenu=H()`

Calls `CreateMenu()`, which takes no parameters and returns a handle.

`SendMessage=L(HW,U,U,*C) ALIAS SendMessageA`

Calls `SendMessage()`, in its general form, which takes four parameters, all of which your code must provide in the argument to `␣wcall`; they are a window handle, two unsigned integers, and a pointer to character data. The result is a long, returned to APL as a 32-bit integer. Invoking `␣wcall 'SendMessage'` actually results in a call to `SendMessageA()`, the Win32 variant of this function that expects one-byte ASCII characters instead of two-byte UNICODE characters.

`RedrawWindow=U(HW hwnd, *RECT lprcUpdate, H hrgnUpdate,U flags)`

Calls `RedrawWindow()`, passing parameters that correspond to the formal parameters *hwnd*, *lprcUpdate*, *hrgnUpdate*, and *flags*. The second parameter is a pointer to a `RECT` structure, using the definition of `RECT` found in the `[Type]` section of the file: `RECT={L,L,L,L}`.

`CB_SetEditSel=I(HW hwnd,~U=,~U,S ichStart, S ichEnd) PACK ALIAS SendMessageA`
 Sends a `CB_SetEditSel` message to the window with `□wcall` supplying three arguments, *hwnd*, *ichStart*, and *ichEnd*. The latter two arguments are packed into the *lParam* parameter.

In general, it is easier to understand the notation if you compare the entries in `APLWADF.INI` with the descriptions of Windows library functions in the *Win32 Programmers' Reference*.

Defining Filter Patterns [Pattern]

The `[Pattern]` section of `APLW.ADF` is used to define a mapping between the callbacks performed by Windows and the callbacks from the APL interpreter into your application. Each line in the `[Pattern]` section defines a template for which callbacks are passed through to APL, and how their arguments are converted to `□warg`. `APLW.ADF` provides for these general types of callbacks:

WP — Window Procedure Patterns

Define how calls to a window's window procedure are filtered and converted. Callbacks for Windows messages that match the filter pattern are in turn transferred to your APL handler, while others are passed to the default window procedure `DefWindowProc()`. Filtering is particularly important for this type of callback because Windows can call a window procedure more often than it is practical for APL to manage.

SP — Subclass Procedure Patterns

Provide a means to intercept calls to existing windows without requiring you to create a new window class. The callback handling is similar to `WP` except that the APL interpreter remembers the address of the original window procedure for the window and transfers any calls that your code does not handle to it.

Other Callbacks

Some windows actions produce specialized callbacks, such as the function `EnumFonts()`, which enumerates the fonts available for a given device. One of the parameters for this call is the address of a callback routine that Windows calls once for each available font. The `EnumFonts` pattern in `APLW.ADF` defines how the parameters to these callbacks are converted.

Other Windows libraries that you might want to use from APL can also make callbacks. If you use them, you need to define new `[Pattern]` entries to control how these callbacks are passed to APL.

Filters can be hierarchical, as is done in `APLWADF.INI` to define a hierarchy of window procedure callbacks. A primary pattern `WP` appears in the `[Pattern]` section. Below that is a section of derivative patterns `[Pattern WP]`, and below that is a section of still more specific patterns `[Pattern WP WM_COMMAND]`. These subfilter specifications make it possible for your application to constrain callbacks to specific Windows messages, so that an APL handler is called for different messages. Alternatively, you can pass every `WM_COMMAND` family message back to APL and let the handler decode the specific message.

Pattern Specification

The general form of a pattern specification is similar to a function call specification, but there are subtle differences.

pname = *rtype* (*pspec*, *pspec* ...) *options*

pname

The name of the pattern, used to define a pattern for a filter specification in the argument to `W_CreateFilter`; (see the section "Creating a Filter," below).

rtype

The result type, which defines the kind of result that must be returned to the caller. Since Windows functions handle only scalar return values, callback handler functions can return only simple scalar values. When a callback occurs, `▯wres` is set to 0; this becomes the return value unless your handler function sets it to a different value.

You can specify a different default return value by following *rtype* with = and the value. If the callback does not expect a return value, the *rtype* should be omitted. In that case, any value you set in `▯wres` is ignored.

pspec

The specification that defines how the parameter is passed by Windows in the callback. The notation is similar to the *pspec* notation used in the `[Call]` section, but subtly different; they describe how Windows parameters are converted into APL arrays instead of vice versa.

options

Various options that control parameter passing and call behavior.

Filter Parameter Specifications: *pspec*

The number of bytes implied by the parameter specification in the primary pattern (that is, in a `[Pattern]` section such as `WP`) must match the number of bytes required by the arguments that Windows supplies when it makes the callback. (Typically, that would mean one parameter specification for each argument; however, four parameters such as `(H,U,U,D)` can also be phrased as `(H,U,U,S,S)` if the `PACK` option is used.) Subordinate patterns (such as in the `[Pattern WP]` section) do not need to have this one-to-one correspondence of parameter specifications to arguments. They can omit trailing arguments. The specification consists of the following parts:

[~] [*ptr*] *dtype* [*repeat*] [*name*] [= [*value*]] [*@*]

~

[Optional] The tilde indicates that the parameter is not passed to APL in `▯warg`.

ptr

[Optional] This indicates that the parameter is a pointer; it must be either '`★`' or '`>`'. The asterisk (`★`) passes a nested array representation of the data structure to APL if *dtype* is specified as a structure in the `[Type]` section of `APLW.ADF`. The greater-than sign (`>`) passes the pointer itself. The specification '`★C`' is treated in a special way: the sequence of characters pointed to is scanned up to a NUL (zero-terminator) byte, and this string is passed to APL as a character vector. If *ptr* is omitted, Windows treats the data as a scalar value and converts it to the appropriate APL datatype.

dtype

The base datatype for the parameter. It can be a simple datatype from the table preceding, or the name of an entry defined in the `[Type]` section. If you define *dtype* with more than one item (see, for example, `LOGFONT` and `RECT` in the `[Type]` section of `APLWADF.INI`), the structure passed is automatically converted into a nested array. Thus, a `RECT` parameter is passed as an item of `▯warg` that is itself a four-element nested array.

[*repeat*]

An optional count (enclosed in brackets) can follow the datatype. The count indicates the number of times the base datatype should be used. Using a repeat count is functionally equivalent to using a named type defined in the `[Type]` section that has the specified number of *dtype* items enclosed in braces. Thus, `C[4]` is the same as `{C,C,C,C}`. However, note that a parameter list such as `(L,L,C,C)` is not the same as `(L,L,C[2])`. The first list specifies four parameters, while the second specifies three parameters, the last

of which is a structure of two elements. The list $(L, L, C[2])$ is the same as $(L, L, \{C, C\})$. See the discussion of "Defining Types" below for a more detailed discussion of repeat counts.

name

[Optional] The field name. This has no effect but can serve to document the formal name of the parameter in the C language description of the callback's syntax.

=value

[Optional]. The default value of the test value used to choose a subfilter. If the *pspec* is prefixed by a tilde '~', this value will not be passed back to APL in $\square$ warg. The *value* can be a list of one or more decimal or hexadecimal constants or constant names that are defined in the [Constant] section. These values are ORed together to form a four-byte integer value that is treated as if APL specified a scalar constant with this value. If you specify the equal sign (=) suffix without any value following it, APL looks for a constant equal to the pattern name in the [Constant] section and uses that value. For example, the WM_DESTROY pattern in the [Pattern WP] section has the following definition:

```
WM_DESTROY=(HW hwnd, ~U=)
```

This means that the pattern is considered to match the incoming arguments if the second parameter has a value equal to the constant WM_DESTROY. This indicates that the message is a WM_DESTROY message. The tilde (~) tells APL not to pass the message code in $\square$ warg.

@

[Optional]. Indicates that this value is used as a range selector to select a subordinate filter. You cannot use both *=value* and *@* in the same filter specification.

Filter Option Specifications: *options*

The options part of a filter specification controls how arguments are unpacked from the C parameter list. The possible options are:

PACK

Allows multiple narrow (8- or 16-bit) pattern specifications to be packed together into a single wider argument such as the *lParam* or *wParam* parameters of a Windows message. The effect of PACK is to interpret the pattern specifications as an unaligned structure. Normally each parameter is aligned to a four-byte boundary when decoding the parameters to a callback.

DEFAULT *proc*

Specifies the name of the default procedure that is called for messages that do not meet the selection patterns for the filter. The default window procedure for WP patterns is DefWindowProc(), the default window procedure for standard Windows classes. The default for SP filters is W_SubProc, an internal function in APL that forwards the message to the window procedure that the window had before it was subclassed.

PATH *pattern*

Specifies the name of another top-level pattern whose result and parameter specification are to be followed. It allows you to specify more than one pattern that uses a common set of result and parameter definitions but different options. For example, the SP pattern specifies PATH WP. This tells APL to use WP's result and argument description, but keeps the SP pattern's DEFAULT option. Thus, the SP pattern does not need to redundantly provide the same filter hierarchy as WP.

AFTER

Indicates that the DEFAULT procedure is always executed, and the APL callback, if any, takes place after

that. For example, this could be used to paint an image on top of a background that Windows paints by default.

BEFORE

Indicates that the APL callback is executed, and then the `DEFAULT` procedure is also executed. This option can be placed only in the argument to `W_CreateFilter`, not in the `.ADF` file.

ASIS

Indicates that `□warg` should contain the parameters as described by the topmost pattern in the hierarchy.

Defining Types [Type]

The `[Type]` section of the `.ADF` file describes abstract datatypes that correspond to defined types in C. A defined type can be a name for a scalar item or a specification for a more complex structure. A type specification has the following general format:

tname = { *tspec*, *tspec* ... } [PACK *n*]

tname

The name used to reference the type in an *rspec* or *pspec* parameter that does not match one of the built-in types or synonyms listed above. For example, the `GetClientRect()` function is specified in the `[Call]` section as:

`GetClientRect=(HW hwnd,>RECT)`

It references the `RECT` type which is defined in the `[Type]` section as:

`RECT={L,L,L,L}`

tspec

A type specification for one item of a data structure. In a call, it is used to convert a nested vector into the appropriate structure or to compute the amount of space to allocate to hold a character-vector argument that you have already converted into a suitable sequence of bytes. On a callback, it is used to unpack the structure into a nested array and return that to APL. A *tspec* consists of the following parts:

[~] [*ptr*] *dtype* [*repeat*] [*name*] [= [*value*]]

- ~
[Optional] In the `[Call]` section the tilde indicates that this part of the structure is not passed by APL, and a default must be generated for the DLL call. In the `[Pattern]` section, it indicates that the parameter passed from Windows is not included in `□warg`.
- ***ptr***
[Optional] Either an asterisk (*), indicating that the parameter is a pointer and passed unmodified between APL and Windows, or greater than (>), indicating that a block of data is created to hold the argument pointed to, and that block's address is passed. The meaning of > varies between the `[Call]` and `[Pattern]` sections as described above.
- ***dtype***
Datatype, as described in the preceding descriptions of the `[Call]` and `[Pattern]` sections.

- **[repeat]**

An optional count (enclosed in brackets) can follow the datatype. The count indicates the number of times the base datatype should be used. Using a repeat count is functionally equivalent to using a named type defined in the [Type] section that has the specified number of *dtype* items enclosed in braces. Thus, `C[4]` is the same as `{C,C,C,C}`. However, note that `{L,L,C,C}` is not the same as `{L,L,C[2]}`. The first is a structure of four elements, while the second is a structure of three elements, the last of which is itself a structure of two elements. The structure `{L,L,C[2]}` is the same as `{L,L,{C,C}}`.

Repeat counts are not limited to built-in datatypes. They can also be specified following the name of a structure. However, when applied to structures, the APL argument representation is slightly different than you might expect. `POINT[3]` is similar to, but not identical to `{POINT,POINT,POINT}`. For example, if a function is defined in the [Call] section as:

```
IP3=(I,*POINT[3])
```

It could be called with two arguments (the second of which could be a six-element vector or a shape 3 2 matrix as shown in the following two calls):

```
□wcall 'IP3' 7 (6 5 4 3 2 1)
□wcall 'IP3' 7 (3 2 ρ 6 5 4 3 2 1)
```

However, if the function were defined as:

```
IPPP=(I,*{POINT,POINT,POINT})
```

Then the second argument would be a nested vector of pairs as shown below:

```
□wcall 'IPPP' 7 ((6 5) (4 3) (2 1))
```

The six-element vector or shape 3 2 matrix used for the IP3 call are not permitted in this case.

You can also use named constants as repeat counts; for example, the LOGFONT structure can be defined as: `{L,L,L,L,L,Y,Y,Y,Y,Y,Y,Y,Y,C[LF_FACESIZE]}`. In this structure, the repeat count, `LF_FACESIZE`, is looked up in the [Constant] section.

- **name**

[Optional] The field name. This has no effect but can serve to document the formal name of the parameter in the C language description of the callback's syntax.

- **value**

[Optional] The default value to use if this element is elided.

PACK *n*

[Optional] The `PACK` option specifies the maximum alignment boundary that will be used to align any element in a structure. The valid values are 1, 2, 4, and 8. If you omit the number, the system defaults to 8. The actual alignment boundary is the smaller of the Pack size and the alignment required for a specific element of the structure.

Packing was modified in version 3.5 of the system. The default value had always been 8, except that the last element of a structure was not padded and therefore the overall structure may not have ended on an alignment boundary. That anomalous behavior has been eliminated. If your application depended on the previous, incorrect, behavior, you can reset a compatibility switch to force the previous packing behavior. In the [Compatibility] section of the .INI file, set:
EnhancedWCALL=0

EnhancedWCALL Compatibility Switch

The following behaviors are affected by this compatibility switch, which is set to 1, by default.

- 1 = new structure Packing behavior, including use of the explicit `PACK` option; 0 = old style (incomplete) packing behavior, and ignore `PACK` options set in the `INI/ADF` files `[Type]` section.
- 1 = The `W_SizeOf` built-in function returns the size of a data type or structure. 0 = returns the size of the structure through the last element but ignores padding necessary to align structure.
- 1 = structured `Wcall` arguments supported; 0 = you can pass structures only as character vectors or by an integer address pointer to a globally allocated data block.
- 1 = Consecutive pointer indicators are counted individually, so that `(**C)` is seen as double indirection, a pointer to a pointer, and `(***STRUCT)` is seen as triple indirection. 0 = two consecutive pointer indicators are treated as one, so that three or four consecutive pointer indicators are needed to specify a pointer to a pointer.
- 1 = passing a character scalar as an argument results in an error message. If you actually want to pass a character scalar, such as to the `CharUpper()` function, use the origin zero `Wav` index of the character as a numeric scalar. 0 = attempt to treat a character scalar in the same manner as a numeric scalar that would be a pointer, possibly crashing the system.
- 1 = one-element vectors are treated as scalars, resulting in the behavior described above. 0 = one-element vectors passed as an argument are treated as a nested execution specification (which was once useful for the Win-16 Agent mechanism, but is now obsolete); one-element vectors passed as arguments are treated as constants to be found in the `[Constant]` section of the `INI/ADF` files.

Promotion Compatibility Switch

By default, `U4` types (unsigned integer) are promoted to `F8` (floating point) if they would be negative in an `APL I4` array. If not promoted, negative values could be returned for `U4` types. To disable this promotion, in the `[Compatibility]` section of the `.INI` file, set:

```
PromoteU4=0
```

Defining Constants [Constant]

The `[Constant]` section of the `.ADF` or `.INI` file defines names for numeric constants, corresponding to constants defined in a C header file. You can specify a constant in either hexadecimal or integer notation.

For example:

```
BLACK_PEN=7  
BAUD_9600=0x00000800
```

You can also specify other named constants to the right of the equal sign. For example:

```
WS_POPUP=0x80000000L  
WS_BORDER=0x00800000L  
WS_SYSMENU=0x00080000L  
WS_POPUPWINDOW=WS_POPUP WS_BORDER WS_SYSMENU
```

The value of `WS_POPUPWINDOW` is the logical OR of the three constants `WS_POPUP`, `WS_BORDER`, and `WS_SYSMENU`.

Note: An `L` following a numeric constant is obsolete and no longer meaningful. It is a carryover from Win16 when it meant a long integer (32-bit rather than 16-bit) constant.

Argument Conversion for the Windows Call Function

This section describes how APL+Win converts the parameters and results of the Windows function calls specified in the `.INI` and `.ADF` files. For more information on defining a Windows function call, see the “Startup Files for `□wcall` (`.ADF` and `.INI`)” earlier in this chapter.

When you specify arguments for `□wcall`, APL+Win converts these values into the format expected by the Windows function that you are calling. APL supports two basic datatypes: numeric and character. Languages such as C support several basic datatypes and contain a facility for defining complex structures composed of these types. Windows uses many of these datatypes. However, APL cannot represent all of these types distinctly. Windows also uses some arguments in several different ways. For example, the `LoadCursor` function can be called with a pointer to a null-terminated character string as its last argument; it can also be called with a small integer constant that does not represent an address. Finally, Windows uses thousands of different program constants. APL must therefore be flexible in how it specifies arguments.

Windows parameters fall into five basic classes according to their declared type in the `.INI` file. APL+Win interprets the APL argument value that you specify in completely different ways depending upon the declared datatype of the corresponding parameter, and the rank and type of the argument value. The `.INI` file defines these Windows classes as scalars, structures, pointers, window handles, and the `X0` datatype.

Scalars

These are any simple (nonstructure) datatypes, except `HW` and `X0`, that are not declared with a pointer prefix (`★` or `>`) and do not have a repeat count suffix. Although the `P4` datatype is called a pointer, it is considered to be a scalar in this context. Scalar arguments can be simple scalars or character vectors.

If the argument is a simple scalar, the value is coerced to the correct datatype. Numeric scalars are accepted as is. Character scalars are converted to their origin-0 indices in `□av`, then the value is coerced to the width specified by the datatype parameter (*dtype*) in the `.INI` function call. No warnings are given for loss of significance or truncation.

If the argument is a character vector, it is scanned for the names of program constants (such as `WM_COMMAND` and `WS_CHILD`) that are defined in the `[Constant]` section of the application `.INI` file, decimal constants (such as 197), and hexadecimal constants (such as `0x00FF0011`). The binary values of all the constants specified in the vector are extended to four-byte integers and bitwise-ORed together into a single integer value. This value is coerced to the width specified by the datatype parameter (*dtype*) in the `.INI` function call. No warnings are given for loss of significance or truncation. (This paragraph assumes the default `EnhancedWCALL` compatibility switch setting.)

Other types of values, such as numeric vectors, are not allowed and signal an error.

Structures

These are any datatypes made up of two or more other datatypes that are not declared with a pointer prefix (`★` or `>`). Structure arguments may be simple numeric vectors or nested vectors. Each element of the vector is coerced to the correct datatype based on the rules pertaining to scalars, pointers, windows handles, or other structures. Elided elements are replaced with the *value* specified in the `[Type]` section. If no value parameter is set, each is replaced with 0 if numeric or `□tcnul` if character.

Non-pointer types with a repeat count (even a repeat count of 1) are also treated like structures. However, the `□wcall` argument in this case can also be a matrix or higher rank array. A matrix can contain as many rows as specified by the repeat count. The number of columns in a matrix can be up to but not exceeding the number of items in the repeated structure. For repeated simple types, this means one column. Note however, that if a vector argument is used, the number of items can be up to the repeat count (for simple

types) or equal to the product of the repeat count and number of items in a repeated structure. Elided rows or columns in the argument are replaced with the *value* as described above.

Pointers

These are any datatype that is declared with a pointer prefix (`*` or `>`). These parameters accept three kinds of APL arguments: simple scalars, character arrays, and non-character (possibly nested) arrays.

- **Simple scalars**

The value is coerced to the correct datatype and passed by value to the Windows function. Simple scalars are treated like C pointers. Numeric scalars are accepted as is. Character scalars generate an error, assuming the default `EnhancedWCALL` compatibility switch setting. With the non-default switch setting, character scalars are converted to integers with possibly dangerous effect.

- **Character arrays**

These are copied into a temporary block of Windows memory and a `NUL` (zero-terminator) character is appended following the data. The address of this block is passed by reference as the argument value. This may represent true character data or data that have been mapped into a character vector to be passed as a function argument. The `S2C`, `C2S`, `I2C`, `C2I`, `W2C`, and `C2W` functions in the `WINAPI` workspace are convenient utilities that convert data between character and integer format. Upon return from the function, the temporary block is freed if you used a `*` pointer prefix, or returned as a character array in the result if you used a `>` pointer prefix.

The last dimension of the array must be an even multiple of the data item length. For example, the data length for `*I2` is two bytes. Therefore, a character vector with an even number of elements is acceptable. The data length for type `*RECT` is 16 bytes for calls and callbacks. (A `RECT` is a structure that contains four 4-byte integers. Therefore, a character vector with 16, 32, 48, ... elements is acceptable, as is a character matrix with shape 5 16.)

APL+Win treats empty character arrays in a special way. It generates a block of zero-initialized memory that is equal in length to the size of the data structure times the product of the leading array dimensions; for example,

```
S××/~1↓parray
```

where `S` is the size of the data structure. Then APL passes the address of this block as the function argument. For example, if you pass an empty character vector as the argument for a parameter declared as `>RECT`, APL+Win generates a 16-byte, null-terminated temporary block of memory before it makes the call. After returning from the call, this block is returned to APL as a 16-element character vector. If you pass an APL matrix with shape 3 0 to a `>RECT` parameter, APL+Win generates a 48-byte, zero-initialized block of memory and passes its address as the parameter value. After returning from the call, a shape 3 16 character array is returned to APL. Note that you could also use shape 0 0, but not a shape 0 16 array.

- **Non-character arrays (pointers to structures only)**

Each element of the array is coerced to the correct datatype for the corresponding item of the structure based on the rules pertaining to scalars, pointers, windows handles, or other structures.

The APL argument values are copied into temporary Windows memory before making the call. Upon return from the function, the temporary block is freed if you used a `*` pointer prefix, or returned in the result as a nested array with the same rank as the argument if defined with a `>` pointer prefix.

Elided elements are replaced with the *value* specified in the `[Type]` section. If no value parameter was set, each is replaced with 0 if numeric or `⌈tcnul` if character.

For example, you can call a function like `GetClientRect` using an empty numeric vector or matrix argument and it is passed as zeros (or *values*). If you use the `>` prefix, the result value is returned with the number of items in the structure as its last dimension. If a repeat count is used on a `>` parameter, the returned value contains a multiple of the number of items in the structure times the repeat count. This feature makes it possible to call a function such as `GetClientRect` conveniently because you can get the system to return a numeric result rather than having to convert a character result in extra APL code surrounding the call.

For example, rather than either of these two statements:

```
rect ← 323 ⍵dr ⍵wcall 'GetClientRect' hwnd (16 ρ ⍵tcnul)
rect ← 323 ⍵dr ⍵wcall 'GetClientRect' hwnd ''
```

You can do these:

```
rect ← ⍵wcall 'GetClientRect' hwnd (0 0 0 0)
rect ← ⍵wcall 'GetClientRect' hwnd ⍬
```

Invalid addresses passed in this way may cause an UNHANDLED EXCEPTION error or an abnormal program termination of your application or Windows. Be very careful when you pass data to a pointer parameter. Since numeric scalars are passed directly, you must ensure arguments are vectors if you want them interpreted as a structure. You must ravel such values to ensure the correct interpretation; for example, (, 0). With the ravel, APL+Win appends any elided trailing elements of the structure and passes a pointer to Windows of that structure. Without the ravel, APL+Win passes the numeric value 0.

Window Handles

This is the `HW` datatype. You specify it without a pointer prefix. Window handle parameters are treated like the scalar parameters described earlier. However, they also allow you to specify a two-element integer vector. In this case, the first item is assumed to be a parent window handle or dialog box handle. The second item is assumed to be the ID number of a control window contained in the parent window or dialog box. APL+Win uses the Windows `GetDlgItem` function to convert this pair of numbers into a single number that represents the window handle of the control ID. For example, to hide a Combo control with ID 101 and the window handle 9876 that was contained in a dialog box with handle 7732, you could use either of the following statements.

```
⍵wcall 'ShowWindow' 9876 'SW_HIDE'
⍵wcall 'ShowWindow' (7732 101) 'SW_HIDE'
```

If you have created a dialog box using low-level Windows API calls, you will know the control IDs of child windows, but you may not know their window handles. Therefore, the second statement is usually more convenient than the first statement.

This capability allows you to use one common set of functions to access child control windows by ID or handle in both dialog boxes and custom windows. Rather than using the limited set of `DlgItem` functions, you can use any Windows function. This is especially useful for the alias macro function families.

x0 Datatype

The system accepts any APL data specified for an `x0` type parameter. The entire array is copied from the APL workspace into Windows memory, and the address of the copy is passed to the called function.

Filters and Callbacks

The `⍵wcall` facility is designed so that the structure of your APL code mirrors the structure that the equivalent C program would use, including how it responds to user input. Like a typical Windows program, your application might create and display one or more windows and then wait for the user to act. The act of waiting consists of calling the function `W_Wait()`, which is a C routine built into the APL+Win interpreter, and which is generally called through an APL cover function also named `W_Wait`. `W_Wait` is the low-level equivalent of the `Wait` method of `⍵wi`. When you call it, control remains in `⍵wcall`, which can then execute callbacks (APL handler expressions) in response to Windows messages.

Some types of user activity generate many messages. For example, dragging the mouse through a window generates many messages per second. Generally, you do not want your application to receive a callback for every event. The fact that the mouse is moving may not be important, although other specific mouse activities, such as button clicks, are important.

To simplify handling of Windows messages and to avoid the execution load of handling every message, the `□wcall` facility includes the concept of *filters*. A filter is a subroutine that the APL interpreter generates dynamically. It is used to define a subset of Windows messages that trigger callbacks and event handlers.

When APL creates a window with `□wcall`, it specifies its window class, which in turn designates the *window procedure* for the window. Windows calls this window procedure to deliver messages about the window. The filter function determines whether to perform a callback to APL or to pass the message on to the default window procedure for that class of window.

Callbacks can occur on any call to APL's Windows interface facility. The same mechanism is used to perform a callback from `□wcall` as from `□wi` and `□wgive`. Therefore, it is possible for a call to `□wi` to cause pending `□wcall` callbacks to run, and vice versa. If `□wgive` is set to a value greater than zero, allowing automatic asynchronous message processing, callbacks from `□wcall`-created windows can occur asynchronously, just as they do for APL+Win Windows interface (WI) objects. `□wcall` callbacks can also occur during the implicit wait that the APL session manager runs while APL is in immediate execution mode, unless you select the menu item that disables callbacks.

From within the APL environment, a callback from `□wcall` is similar to a callback from `□wi`. The system variable `□warg` contains the parameters to the callback. The format and number of elements in `□warg` depend on the type of message-processing pattern that was specified in the filter (see the section on "Defining Filter Patterns" earlier in this chapter). `□wself` is empty, and `□wevent` is '?'. Rows in `□si` show the callout and callback. For example, during the callback for the paint handler of the function `DemoHello` in the DEMOWIN workspace, `□si` has a form like the following:

```
DemoHello_WmPaint[14]  
>[;?]  
<[;?]  
DemoHello[105]
```

Summary of the Callback Mechanism

The steps below show the overall process your code will follow in setting up callbacks into APL for Windows messages. The concepts and details are explained in the sections that follow.

1. Create a filter, which defines the specific messages that cause APL handlers to run.
2. Register a new window class with Windows and specify the filter as part of the definition.
3. Create a new window of that window class.
4. Run `W_Wait` or invoke the `Wait` method using `□wi` on the main form of your application. `W_Wait` runs the handlers you provide in response to the events you specify in the filter. Your code can wait inside `W_Wait`, with callbacks occurring automatically, in a manner similar to the `Wait` method of `□wi`.

An alternative way to set up callbacks is to *subclass* an existing window. This sequence modifies the second and third parts of the above process:

2. Create a window of an existing Windows class (either a built-in class or one that has been previously registered by APL or other software).
3. Subclass that window by specifying a filter that intercepts selected messages and calls your handler for them, letting the window's previous window procedure handle other messages.

The sample function `DemoHello` in the DEMOWIN workspace is a simple example of this process. Study its code and comments carefully. The following statements allow you to run the demo. The third statement initializes several necessary global variables.

```
)LOAD DEMOWIN  
)COPY WINAPI  
DemoLX  
DemoHello
```


Window Classes and Window Procedures

Windows delivers information to an application about user input or other events by calling the window procedure for one of the application's windows, and passing a standardized set of up to four parameters that constitute a message. The message provides information about the event that transpired, and the window procedure is expected to perform whatever processing is appropriate in response to the event.

The window procedure for a window is specified indirectly as a characteristic of its *window class*. Every window created in Windows has a specific window class. This class is a data structure that describes some attributes of the window; it is represented by the Windows structure type `WNDCLASS`. One of the fields in this structure is the address of the window procedure. Thus, every window in a specific class shares the same window procedure. Several predefined classes are built into Windows, including `BUTTON`, `COMBOBOX`, `EDIT`, `LISTBOX`, `MDICLIENT`, `SCROLLBAR`, and `STATIC`. Your application code *registers* additional classes with Windows using the function `RegisterClass()`, passing the address of a `WNDCLASS` structure. Once the new class is registered, you can use it as the basis for creating windows that are instances of that class.

Creating a Filter

The utility function `W_CreateFilter`, in the `WINAPI` workspace, creates a filter for Windows messages. `W_CreateFilter` is an APL cover function that in turn calls an entry point in the APL interpreter also named `W_CreateFilter()`. The argument to `W_CreateFilter` consists of two parts: the name of a filter pattern and a filter specification that associates handler expressions with Windows messages. For example,

```
fp ← W_CreateFilter 'WP' '□warg'
```

creates a filter that can be registered as the window procedure for a specific window class using `W_RegisterClass`. The example above represents the most generic kind of window procedure. It simply calls APL to execute the statement `□warg` every time a Windows message is received. In practice, it is seldom useful to monitor every message because Windows generates many messages, most of which you can safely ignore.

A filter like the one above also has the undesirable effect of making the window totally "dead", because every message from Windows is turned into an APL callback that does nothing except display the message in the APL session. As a result, the window's default window procedure is never called, and it is never able to respond to any windows message.

Practical use of filters requires defining more complex filters that describe the sets of messages that are of interest and that specify different handlers for different messages. Other messages are ignored and executed by the default window procedure. Such filters are generated by calling `W_CreateFilter` with a more complex argument. The general form of its syntax is:

```
filterpointer ← W_Createfilter 'pattern' ('msg' 'action') ('msg' 'action')...
```

The *pattern* is the name of a syntactic pattern found in the `[Pattern]` section of the `.ADF` file. It is a template for how the parameters in the Windows message are handled. It is followed by an arbitrary number of `('msg' 'action')` pairs, which contain the name of a Windows message and the text of the APL statement that runs when the message is received. Windows messages are encoded as named constants, making the filter specification easier to read.

For example, the following statement creates a filter to process `WM_COMMAND` and `WM_PAINT` events:

```
fp ← W_CreateFilter 'WP' ('WM_COMMAND' 'Cmd') ('WM_PAINT' 'Paint')
```

This filter is structured to be a window procedure; it therefore uses the pattern `'WP'`. It handles two Windows messages: `WM_COMMAND` and `WM_PAINT`, and executes the APL expression `'Cmd'` or `'Paint'`,

respectively. APL does not handle other Windows messages; they are passed to the generic default window procedure of the Windows API, `DefWindowProc()`.

If you look in `APLWADF.INI`, you find the pattern `WP` defined in the `[Pattern]` section with the following definition:

```
WP=D(HW hwnd,U msg @,U wParam,D lParam) DEFAULT DefWindowProc
```

This information, plus the information the `W_CreateFilter` argument provides, allows APL to generate the filter automatically. For details on the syntax of pattern descriptions, see the "Defining Filter Patterns" section earlier in this chapter. The logic of the filter is equivalent to the following function in C:

```
LONG API My_WndProc(hwnd, msg, wParam, lParam)
HWND hwnd;          /* window handle */
WORD msg;           /* message type */
WORD wParam;        /* message parameter */
LONG lParam;        /* additional message parameter */
{
    WORD hi, lo; /* unpacked parameters from lParam */
    switch(msg) {
        case WM_COMMAND: /* command message */
            lo=LOWORD(lParam)
            hi=HIWORD(lParam)
            return Cmd(hwnd, wParam, lo, hi, lParam);
        case WM_PAINT: /* repaint notification */
            return Paint(hwnd);
        default: /* all other messages */
            return defWindowProc(hwnd, msg, wParam, lParam)
    }
}
```

Registering a Window Class Based on a Filter

Once you create a filter procedure, you can use its address in a structure that defines a new window class to Windows. This structure is of type `WNDCLASS`, which is described in detail in the *Windows Programmer's Reference* chapter on Structures. One part of this structure is a pointer to the class's window procedure; this is where you store the result of `W_CreateFilter`, which is the address of the filter.

The function `W_RegisterClass`, in the `WINAPI` workspace, creates this structure and registers the new Window class. Its argument is a ten-item nested array, which corresponds to the 10 fields in a `WNDCLASS` structure that is used as the parameter of the Windows function `RegisterClass()`. The argument to `W_Register_Class` is constructed as follows:

```
wndclass←c'CS_REDRAW CS_VREDRAW'  A style
wndclass←wndclass,cfp              A window proc (filter)
wndclass←wndclass,0                A cbClsExtra (none needed)
wndclass←wndclass,0                A cbWndExtra (none needed)
wndclass←wndclass,chInst           A hInstance
wndclass←wndclass,c'Load_IDI' 'IDI_APPLICATION' A icon
wndclass←wndclass,c'Load_IDC' 'IDI_ARROW'       A cursor
wndclass←wndclass,c'GetStockObject' 'WHITE_BRUSH' A background
wndclass←wndclass,0                A MenuName (unused)
wndclass←wndclass,c'MyWindowClass' A Class name
```

This nested array is then passed as the argument to `W_RegisterClass`.

```
W_RegisterClass wndclass
```

Once the class is registered, you can use it to create new windows.

Creating a Window

The Windows API uses `CreateWindow()` to create every window, whether it is a top-level window managed by the Windows desktop manager or a simple control such as a push button. Most of the objects that `⌈wi` provides are windows in this sense. The automation provided by `⌈wi` includes automatically running `CreateWindow()`. Working at the `⌈wcall` level requires your APL code to make many Windows API calls to achieve the equivalent effect.

`CreateWindow()` takes several parameters, the first of which is the name of the window class. See the description of this function and its arguments in the Functions chapter of the *Windows Programmer's Reference* for the essential information. The following statements create a parameter list and call

```
CreateWindow().
  wspec←'MyWindowClass'  A class name
  wspec←wspec,c'MyApp'    A window text (caption)
  wspec←wspec,c'WS_OVERLAPPEDWINDOW' A style
  wspec←wspec,c'CW_USEDEFAULT'      A horizontal position
  wspec←wspec,c'CW_USEDEFAULT'      A vertical position
  wspec←wspec,c'CW_USEDEFAULT'      A window width
  wspec←wspec,c'CW_USEDEFAULT'      A window height
  wspec←wspec,c0              A parent window handle (unused)
  wspec←wspec,c0              A menu/child window id (unused)
  wspec←wspec,hInst           A application instance handle
  wspec←wspec,0               A Window creation data (unused)

  hwndMyApp←⌈wcall (<'CreateWindow'),wspec
```

The result returned by `⌈wcall` is the new window's window handle. This is the *hwnd* parameter supplied as the first item of `⌈warg` when callbacks execute. Because of the way the class `MyWindowClass` was defined above, `⌈wcall` executes an APL handler each time the window receives a `WM_COMMAND` or `WM_PAINT` message from Windows.

Subclassing a Window

You can also install a filter into an existing window through the technique of *subclassing*. This term may be a little misleading, seeming to imply more than the technique actually achieves. To subclass a window, you replace its current window procedure with your own filter, which handles messages of interest and passes any unhandled messages to the original window procedure. Conceptually, it is much the same as hooking an interrupt in DOS.

You use a different style of filter to subclass a window than to define a new window. Instead of the `WP` pattern, you use `SP`. In the `[Pattern]` section of `APLW.ADF`, the `SP` filter pattern is defined as a variant of `WP`, except that the default window procedure is named as `W_SubProc()` instead of `DefWindowProc()`. `W_SubProc()` is a routine in the APL interpreter that remembers the location of the window's original window procedure and calls it for messages that do not match the filter pattern. The following example creates a filter you can use to subclass a window:

```
sfp ← W_CreateFilter 'SP' ('WM_CHAR' 'CharFilter') (WM_SETFOCUS' 'Focus')
```

The result `sfp` is a pointer to a subclass filter. You can now use the function `W_SubClass` in the `WINAPI` workspace with this filter to subclass an existing window. Suppose that `hEdit` contains the window handle of a previously created `Edit` class window:

```
rc ← W_SubClass 0 hEdit sfp
```

Now, all `WM_CHAR` messages cause the APL statement `'CharFilter'` to be executed, all `WM_SETFOCUS` messages run `'Focus'`, and all other messages pass through to the control's original window procedure.

After whatever processing your handler performs in response to the intercepted message, your code can pass the message to the original handler by calling the utility function `W_Forward` in the `WINAPI` workspace.

Thus, the handler `Focus`, called by the subclass filter above, might perform some action in response to a change in focus, and then forward the message to the original windows procedure.

```
▽ Focus
[1]   'Lost focus; hwnd = ', ⚪warg
[2]   W_Forward
▽
```

Processing Messages

Windows has a large vocabulary of distinct messages that can be sent to a window in various situations. Each window that has an APL filter installed as its window procedure passes selected messages to APL by executing a handler. During the time that the callback handler is running, the value of `⚪warg` contains the parameters passed to the window procedure. Some messages return the standard four-element integer vector, but note that some messages result in fewer parameters being passed. You need to look up the message in `APLWADF.INI` to find out how many parameters are passed in `⚪warg` and which ones they are, or you can use the utility function `W_Ini`. For example:

```
W_Ini '[Pattern WP] WM_COMMAND'
```

For messages that return the standard four parameters, the elements of `⚪warg` are as follows:

- `⚪warg[1]` is the window handle of the window receiving the message, corresponding to the `hwnd` parameter of `SendMessage()` in the *Programmer's Reference*.
- `⚪warg[2]` is the message identifier, an unsigned numeric constant, corresponding to the `uMsg` parameter.
- `⚪warg[3]` is a message parameter, either a single 32-bit integer or two 16-bit unsigned integers packed into one 32-bit integer, corresponding to the `wParam` parameter.
- `⚪warg[4]` is a message parameter, either a single 32-bit integer or two 16-bit unsigned integers packed into one 32-bit integer, corresponding to the `lParam` parameter.

Windows messages are identified in C code by their named constants as defined in `windows.h`. Each is described in detail in the Messages chapter of the *Windows Programmer's Reference*. A naming convention uses a prefix to differentiate generic window messages from class-specific messages. The prefix also distinguishes *command* messages (sent to a window to direct it to do something) and *notify* messages (sent by a window to notify another window, usually its parent, that it has done something).

Waiting for Messages

After you set up windows and their callbacks into APL, the next step is to call the APL function `W_Wait`. This function runs a message loop for you inside the APL interpreter and passes appropriate callbacks back to APL when you want your handlers to run.

When you call `W_Wait`, APL begins polling the Windows message loop. Control remains inside `⚪wcall`, and all subsequent execution of your code occurs in response to callbacks from Windows. When a callback occurs, APL sets up the arguments and the default result in `⚪warg` and `⚪wres`, and then executes the APL expression that you defined to handle the callback. `W_Wait` remains in this polling state until your program calls `PostQuitMessage()` to terminate execution. The return value from `W_Wait` is the argument to `PostQuitMessage()`. After `PostQuitMessage()` has been called, subsequent invocations of `W_Wait` return without waiting unless you call `W_Init` to reinitialize the interface.

As a simple alternative, it usually suffices to use `⚪wi` and the `Wait` method. If your application does not wait on a form anyway, you can create a hidden form for this purpose.

Callbacks Other than the Window Procedure

Most callbacks occur from Windows delivering a message by calling your window procedure. However, some parts of the Windows API use callbacks for other purposes besides passing messages. For example, to use the function `EnumWindows()` you must pass the address of a callback function as one of its parameters. Windows then calls this callback function once to pass information about each top-level window. These callbacks are also defined in the `[Pattern]` section of `APLW.ADF`, and tell APL how to format callback arguments that are passed back to your application code in `□warg`.

Related Workspaces

The `WINAPI` workspace, distributed with `APL+Win`, contains a number of useful functions for working with `□wcall`.

The `DEMOWIN` workspace contains a selection of sample functions for working with `□wcall` and the Windows API. To prepare to use this workspace, you should follow this procedure:

```

)load \examples\demowin
)copy \tools\winapi
DemoLX

```

The `DemoLX` function initializes the global variables `hInst`, `hPrev`, `lpCmd`, and `nShow`, which are used by many of the demonstration functions. Some that are worth examining are listed below. Note that the code for these functions is intended to be their documentation. You should read it carefully before you try working with any of them.

- `Beep` is a cover for the function `MessageBeep()`, which generates a beep on the speaker.
- `DemoGDI` demonstrates the use of low-level graphics facilities with direct GDI calls. The actual operations shown are all possible to achieve with the `Draw` method; however, this function can serve as a model for how you call an output-only graphics library from APL.
- `DemoHello` is the classic "Hello, Windows" program of Charles Petzold's *Programming Windows*, recast in APL using low-level Windows calls. Extensively commented, it illustrates the essential steps of working with the Windows API with `□wcall` and may make a useful template for your own code. It also illustrates the contrast in effort between programming at the `□wi` level, where this program is trivial, and doing the same thing with low-level calls.
- `EnumFonts` illustrates the use of a callback that is not a window procedure to process a message. Returns a `LOGFONT` structure for each available style of a particular typeface. Try:

```
EnumFonts 'APLFONT'
```

Chapter 12: Zip and Unzip in APL+WIN

Background

The Zip Class is an object programming model that provides facility for building archives of files and extracting files from those archives. The Zip class provides an interface to the Info-ZIP dynamic load libraries zip32.dll and unzip32static.dll. These dlls must be in the same directory as your APL interpreter. The Zip class is manipulated with `⌵WI`.

Zip Class in APL+Win

To use the Zip class to build and manipulate file archives, you first instantiate an instance of the Zip class. This is done using `⌵WI`. Properties can then be set that control the behavior of the Zip class methods. Zip event handlers are set to provide progress reports and opportunities for user interaction. To store files into an archive, use the Zip method. To extract files or information about the archive contents, use the Unzip method.

Zip Code Examples

Below are a few samples of code to zip, unzip, and list files. For these examples, the `lf to cr` function (provided at the end of this section) just changes embedded linefeeds to carriage returns.

Creating an Archive

The following code will zip all files in directory `d:/testzip/unzip` into an archive named `d:/testzip/foo.zip`.

```
⌵wself←'z' ⌵wi 'Create' 'Zip'
⌵wi 'verbose' 1
⌵wi 'onDisplay' '(1≠ρ⌵warg)/⌵wself ⌵wevent ⌵warg'
⌵wi 'Zip' 'd:/testzip/foo' 'd:/testzip/unzip' (⊂1ρ'★')
⌵wi 'Delete'
```

The verbose option causes Display events which are set to generate output for each archived member such as:

```
z Display updating: index.html
z Display (92 bytes security)
z Display (deflated 59%)
```

Listing an Archive

The following code will list all members of the archive named `d:/testzip/foo.zip`. The members will not be unzipped. The `onListing` event is the key. The verbose property value of 1 causes the totals property to be populated during the Unzip method execution.

```
⌵wself←'z' ⌵wi 'Create' 'Zip'
⌵wi 'verbose' 1
⌵wi 'onListing' '⌵warg'
⌵wi 'Unzip' 'd:/testzip/foo'
```

```

❏wi 'totals'
❏wi 'Delete'

```

The listing is generated by the `onListing` event and is the result of `❏warg`. It consists of the following seven elements: file name, deflate method, sizes(original size, deflated size, % deflation), date(year, month, day, hour, minute), cyclic redundancy check sum, flag to indicate an encrypted file, and a reserved value. Here is some typical output:

```

index.html Defl:N 1520 625 59 2004 11 1 20 16 -1776231350
next.png Stored 3600 3600 0 2002 12 3 19 52 1567415681

```

The `❏wi 'totals'` statement will produce a sum of the following values for the archive members: compressed size, uncompressed size, compression factor, number of members, comment characters. Here is typical output:

```
214485 594715 64 31 0
```

Extracting Files from an Archive

The following code will unzip all files of the archive named `d:/testzip/foo.zip` into the directory `d:/testzip/unzip`.

```

❏wself<-'z' ❏wi 'Create' 'Zip'
❏wi 'verbose' 2
❏wi 'onDisplay' '(1≠❏warg)/❏wself ❏wevent ❏warg'
❏wi 'Unzip' 'd:/testzip/foo' 'd:/testzip/unzip'
❏wi 'Delete'

```

The verbose property value of 2 is required to trigger the `onDisplay` events. These are generated for each archived member. Here is typical output from the event handler definition shown above:

```

z Display inflating: d:/testzip/unzip/HI20050502.doc
z Display inflating: d:/testzip/unzip/index.html
z Display extracting: d:/testzip/unzip/next.png
z Display extracting: d:/testzip/unzip/nextgrey.png

```

Existing Files

If the files already exist in the specified directory, this code will generate messages something like this:

```

z Display Target file exists.
Skipping d:/testzip/unzip/HI20050502.doc

z Display Target file exists.
Skipping d:/testzip/unzip/index.html

```

Overwriting Existing Files

To force overwriting of existing files, set the `overwrite` property to 1 before running the `Unzip` method:

```

❏wi 'overwrite' 1
❏wi 'verbose' 2

```

```
⊞wi 'onDisplay' '(1≠ρ⊞warg)/⊞wself ⊞wevent ⊞warg'  
⊞wi 'Unzip' 'd:/testzip/foo' 'd:/testzip/unzip'
```


Overwriting with Newer Files

To force overwriting existing files with newer archive members, set the `onlynewer` property and the `overwrite` property both to 1 before running the `Unzip` method:

```

␣wi 'overwrite' 1
␣wi 'onlynewer' 1
␣wi 'verbose' 2
␣wi 'onDisplay' '(1≠ρ␣warg)/␣wself ␣wevent ␣warg'
␣wi 'Unzip' 'd:/testzip/foo' 'd:/testzip/unzip'

```

This will generate an output similar to the following (the archive version of `colossal_data.html` was newer than the directory version and so was replaced):

```

z Display Target file newer.
Skipping d:/testzip/unzip/colossal_data.dot
z Display inflating: d:/testzip/unzip/colossal_data.html
z Display Target file newer.
Skipping d:/testzip/unzip/colossal_data.png

```

Overwriting by Response

To enable a choice of overwriting files, set the `onReplace` event handler to return a response:

```

␣wself←'z' ␣wi 'Create' 'Zip'
␣wi 'verbose' 2
␣wi 'onReplace' 'lftocr (1≠ρ␣warg)/␣wself ␣wevent ␣warg ⋄ ␣wres←"Y"'
␣wi 'onDisplay' 'lftocr (1≠ρ␣warg)/␣wself ␣wevent ␣warg'
␣wi 'Unzip' 'd:/testzip/foo' 'd:/testzip/unzip'
␣wi 'Delete'

```

Note that this `onReplace` event handler always returns an affirmative “Y”, so all files are replaced. Return “N” to specify not replacing a file. See the reference section for more options. If the event handler returned a result depending on user input, some files might be replaced and others skipped. Here is typical output from this code:

```

z Replace d:/testzip/unzip/cfreplace.html
z Display inflating: d:/testzip/unzip/cfreplace.html
z Replace d:/testzip/unzip/colossal.log
z Display inflating: d:/testzip/unzip/colossal.log

```

Controlling Zip with onProgress

To enable the user to terminate the zip process at will, use the `onProgress` event:

```

␣wself←'z' ␣wi 'Create' 'Zip'
␣wi 'verbose' 2
␣wi 'overwrite' 1
␣wi 'onProgress' '(1≠ρ␣warg)/␣wself ␣wevent ␣warg ⋄ ␣wres←0'
␣wi 'Unzip' 'd:/testzip/foo' 'd:/testzip/unzip'
␣wi 'Close'␣wi 'Delete'

```

Note that this onProgress event handler always returns a zero. This continues execution. To terminate execution, return a one. A terminated Zip or Unzip causes an error interrupt with one of the following error messages:

␣WI Zip Failure: user interrupt or termination

␣WI Unzip Failure: user interrupt or termination

Encrypting an Archive with Passwords

To encrypt and password protect a file, use the onPassword event with the Zip method:

```
␣wself←'z' ␣wi 'Create' 'Zip'
␣wi 'onPassword' '(1≠ρ␣warg)/␣wself ␣wevent ␣warg ⧫ ␣wres←"abcdefgh"'
␣wi 'Zip' 'd:/testzip/pwtest.zip' 'd:/testzip/unzip' (←'★.png')
␣wi 'Delete'
```

Note that this onPassword event handler specifies the password directly. Normally, the password would be supplied via user input on a form. Two Password events will be generated, the first for the initial password, and the second for a confirmation. The data displayed by the event handler with this code would look like this:

```
z Password 81 Enter password:
z Password 81 Verify password:
```

To decrypt a password protected file, use the onPassword event with the Unzip method:

```
␣wself←'z' ␣wi 'Create' 'Zip'
␣wi 'verbose' 2
␣wi 'overwrite' 1
␣wi 'onDisplay' '(1≠ρ␣warg)/␣wself ␣wevent ␣warg'
␣wi 'onPassword' '(1≠ρ␣warg)/␣wself ␣wevent ␣warg ⧫ ␣wres←"abcdefgh"'
␣wi 'Unzip' 'd:/testzip/pwtest.zip' 'd:/testzip/unzip'
␣wi 'Delete'
```

Note that this onPassword event handler specifies the password directly. Normally, the password would be supplied via user input on a form. For Unzip, only one Password event is fired. The data displayed by the event handler with this code would look like this:

```
z Password 81 Enter password for: prev.png
z Display extracting: d:/testzip/unzip/prev.png
z Display inflating: d:/testzip/unzip/colossal_data.png
z Display extracting: d:/testzip/unzip/next.png
z Display extracting: d:/testzip/unzip/nextgrey.png
z Display extracting: d:/testzip/unzip/prevgrey.png
z Display extracting: d:/testzip/unzip/up.png
```

Including and Excluding Files

The Zip and Unzip methods provide arguments to specify files to include in the archive or the target folder. Additionally, the Unzip method provides an argument to specify files to be excluded. The following example shows how to create an archive specifically including all files with names ending in ‘.png’ and ‘.html’:

```

wself←'z' wwi 'Create' 'Zip'
wwi 'verbose' 1
wwi 'onDisplay' '(1≠ρwarg)/wself wevent warg'
wwi 'Zip' 'd:/testzip/foo.zip' 'd:/testzip/unzip' ('*.png' '*.html')
wwi 'Delete'

```

Below is some typical abbreviated output from the above code:

```

z Display updating: cfreplace.html
z Display (92 bytes security)
z Display (deflated 54%)

z Display updating: colossal_data.html
z Display (92 bytes security)
z Display (deflated 52%)

z Display updating: colossal_data.png
z Display (92 bytes security)
z Display (deflated 20%)

z Display updating: colossal_data_1.html
z Display (92 bytes security)
z Display (deflated 48%)

```

The following section of code demonstrates how to extract files from an archive with names that end in 'png', except for those that start with 'next' or 'prev':

```

wself←'z' wwi 'Create' 'Zip'
wwi 'verbose' 2
wwi 'onDisplay' 'lftocr (1≠ρwarg)/wself wevent warg'
wwi 'overwrite' 1
wwi 'Unzip' 'd:/testzip/foo.zip' 'd:/testzip/unzip' (<'*.png') ('next*.png'
'prev*.png')

```

Below is some typical output from the above code:

```

z Display inflating: d:/testzip/unzip/colossal_data.png
z Display inflating: d:/testzip/unzip/colossal_hole_table_a.png
z Display inflating: d:/testzip/unzip/colossal_hole_table_b.png

z Display inflating: d:/testzip/unzip/colossal_struct.png
z Display extracting: d:/testzip/unzip/up.png
z Display extracting: d:/testzip/unzip/up.png

```

The lftocr Function

The lftocr function changes Linefeed characters to Carriage Return characters in all character vectors and nested character vectors. This is used as a utility function in some of the code samples above:

```
▽ r ← lftocr arg;⍵io
[1]   ⍵io←0
[2]
[3]   :if 82=⍵dr arg
[4]       r←arg
[5]       (( r=⍵tclf)/r)←⍵tcnl
[6]
[7]   :elseif 807=⍵dr arg
[8]       r← lftocr ``arg
[9]
[10]  :elseif 326=⍵dr arg
[11]      r← lftocr ``arg
[12]
[13]  :else
[14]      r←arg
[15]  :endif
▽
```

Reference Section

This section discusses the primary methods, properties and events of the Zip class. The common methods, properties, and events possessed by most other `⍵wi` class objects have common behavior and are not discussed here.

`⍵WI` Zip Class

Syntax:

```
result←'z' ⍵WI 'Create' 'Zip'

result←'z' ⍵WI 'action_name' arguments
```

Result:

The result of Zip Class related `⍵WI` calls are similar to other uses of `⍵WI`.

Most calls return an empty array with a shape of `0 0`. Property reads return the values of the specified properties.

Zip Methods

Principal Methods: Create, Delete, Unzip, Zip

Create

Purpose:

Creates an instance of a Zip class. This Zip instance can then be used to create, read, and write file archives.

Syntax:

```
result←'z' ⍲wi 'Create' 'Zip'
```

Arguments:

The left argument is a character vector or scalar representing the name of the Zip class instance to be created. 'Zip' is the appropriate right argument to create an instance of a Zip class.

Results:

The name of the Zip class created.

Delete

Purpose:

Delete an instance of the Zip class.

Syntax:

```
'z' ⍲wi 'Delete'
```

Arguments:

The left argument is a character vector or scalar representing the name of the Zip class to be deleted.

Results:

An empty array with shape 0 0.

Zip

Purpose:

The Zip method compresses multiple files and stores them into an archive.

Syntax:

```
'z' ⌵wi 'Zip' archive folder (included) omitdir
```

Arguments:

The left argument is a character vector or scalar representing the name of the Zip instance. The ‘archive’ argument is a character vector representing the name of the archive to be created or added to. If no file extension is provided for this archive name, a default extension of ‘.zip’ is used. The ‘folder’ argument is a character vector name of the base folder in which the files to be archived are found. The (included) argument is a nested vector of character vectors representing the names of files to be included in the archive. The (included) list is an optional argument. The character vectors in (included) may include asterisks (*) as wildcards, question marks (?) as a single non-blank character place holder or a regular expression substitution with any string of characters inside the square brackets such as “aplfile.[sc]f” would find both aplfile.sf and aplfile.cf. The ‘omitdir’ argument is a Boolean flag that tells when to exclude the directory name for files in the zip file. The default value is 0.

Results:

An empty array with shape 0 0.

Unzip

Purpose:

The Unzip method reads an archive and extracts files from it, or returns information about the contents of the archive.

Syntax:

```
'z' ⌵wi 'Unzip' archive folder (included) (excluded)
```

Arguments:

The left argument is a character vector or scalar representing the name of the Zip instance. The ‘archive’ argument is a character vector representing the name of the archive to be read. The ‘folder’ argument is a character vector name of the base folder in which the archived files are to be written. The (included) argument is a nested vector of character vectors representing the names of files to be extracted. The (excluded) argument is a nested vector of character vectors representing the names of files to be excluded from the extraction. The (included) list and the (excluded) list are optional arguments. An empty character vector may be used as a place holder for the (included) list. This allows only the (excluded) list to be specified. The character vectors in (included) and (excluded) may include asterisks (*) as wildcards, question marks (?) as a single non-blank character place holder or a regular expression substitution with any string of characters inside the square brackets such as “aplfile.[sc]f” would find both aplfile.sf and aplfile.cf.

Results:

An empty array with shape 0 0.

Zip Event Handler Properties

Principle Event Handlers: `onComment`, `onDisplay`, `onListing`, `onPassword`, `onProgress`, `onReplace`, `onSound`.

For each event, you can specify an event handler, which is an APL expression. When the event occurs, the system executes the specified handler.

Three system variables, `⌵wevent`, `⌵wself`, and `⌵warg`, provide detailed information about the event and the instance it occurred on.

Syntax:

```
result←'z' ⌵wi event handler
```

```
result←'z' ⌵wi 'Set' (event1 handler1)...(eventN handlerN)
```

Arguments:

A two element vector of character vectors. The first element is the event name. The second element is the APL expression that is the corresponding event handler.

Using the `Set` method, you can set any number of Zip event-handlers in one call.

Result:

An empty array with shape `0 0`.

`onComment`

Trigger:

During the `Zip` method, when the method is ready to store a comment in the archive.

Remarks:

`⌵wevent` contains the event name 'Comment'. `⌵warg` contains any current comment from the existing archive. A new comment is set by assigning a character vector to `⌵wres`. The vector is restricted to no more than 32,767 characters.

`onDisplay`

Trigger:

During the `Zip` and `Unzip` methods, when each member of an archive is processed.

Remarks:

`⌵wevent` contains the event name 'Display', `⌵warg` contains processing information about the current member of the archive. Triggering `onDisplay` is also effected by the setting of the 'verbose' property. A value of 0 will disable `onDisplay` events, a value of 1 will produce some `onDisplay` events, and a value of 2 may

produce even more onDisplay events. Setting the value of the 'comment' property to 1 will also cause an onDisplay event for comments found in the archive.

onListing

Trigger:

During the Unzip method, when each member of an archive is processed.

Remarks:

Note that setting an onListing event handler changes the behavior of the Unzip method. Instead of extracting files from the archive, the Unzip method will only read the archive and generate a listing if onListing is defined.

When this event is fired, `□wevent` contains the event name 'Listing', `□warg` contains data about the current member of the archive. The listing data in `□warg` is a nested vector containing seven elements: File name, deflate method, file size(an integer vector of: original file size, deflated file size, percent deflation), timestamp(an integer vector of: year, month, day, hour, minute), CRC, encryption flag, (and a reserved element).

Caveat on file timestamps: This utility, like many others running in this environment, presents local timestamps. Tagging an object that can be moved across time zones and independently viewed across time zones with a time stamp that is anything other than UTC produces ambiguous results.

onPassword

Trigger:

During the Zip method, when at the beginning of the archiving processing, onPassword is triggered with a message of '81 Enter password:' and then again with a message of '81 Verify password:' in `□warg`. During the Unzip method, when beginning to read the archive, onPassword is triggered with a message of '81 Enter password for: (file name)' in `□warg`.

Remarks:

`□wevent` contains the event name 'Password', `□warg` contains the password message. Triggering onPassword during the Zip will happen at the beginning of the Zip process if there is an onPassword event handler defined. For the Unzip method, the onPassword event will be triggered if the archive is password protected and there is an onPassword event handler defined. The password is specified, by the handler, as a character vector in `□WRES`.

onProgress

Trigger:

During the Unzip method, after unzipping each file, onProgress is triggered. Also during the Zip method, as each member is processed, but before the archive is actually changed.

Remarks:

`onProgress` provides an opportunity to terminate the Unzip or Zip operation at each event. `EventArgs` contains the event name 'Progress', `EventArgs` contains a two element vector. The first element is the full path and file name of the newly unzipped file. The second element is the size of the unzipped file as an integer. The response is specified as an integer value in `WRES`. A value 0 continues processing. A value of 1 stops the processing and interrupts the Unzip or Zip method.

onReplace**Trigger:**

During the Unzip method, prior to overwriting an existing file, `onReplace` is triggered.

Remarks:

`onReplace` provides an opportunity to specify replacement or not of each file in the target folder. `EventArgs` contains the event name 'Replace', `EventArgs` contains character vector representation of the full path and file name of the target file. The response is specified as a character value in `WRES`.

A value of 'N' or 'n' means: do not replace the file.

A value of 'Y' or 'y' means: replace the file.

A value of 'A' or 'a' means : replace all files.

A value of 'Z' or 'z' means : replace no files.

onSound**Trigger:**

During the Unzip method, at each file, `onSound` is triggered. `EventArgs` contains the event name 'Sound'.

Remarks:

This provides an opportunity for the application to present an audible or a visual queue to the user.

Zip Class Properties

Principal Properties: `comment`, `level`, `omitdir`, `onlynewer`, `overwrite`, `privilege`, `recreatedirs`, `recursedir`, `totals`, `uzoptions`, `verbose`.

`comment`

Description:

An integer singleton that specifies comment display in Unzip.

Remarks:

A value of 0 specifies no comment processing. A value of 1 specifies return any comment on the Unzip method via the `onDisplay` event. The default value is 0.

`level`

Description:

On the Zip method, the value of `level` specifies the compression level of the archived files.

Remarks:

The value is an integer singleton with a range of 0 to 9. A value of 0 indicates no compression. A value of 9 indicates maximum compression. Values in-between indicate various trade-offs between compression level and efficiency. The default value is 6.

`omitdir`

Description:

On the Zip method, the value of `omitdir` specifies whether to include the `directory` name for the file in the archived file.

Remarks:

A value of 1 will include the directory name for the file in the archived file. A value of 0 will not include the directory name in the archived file. The default value is 0.

`onlynewer`

Description:

The `onlynewer` property specifies how duplicate file names should be handled during the Unzip process. The `onlynewer` property provides a prior condition that must be satisfied before any other properties and events can allow overwriting a file.

Remarks:

A value of 0 is nilpotent and does not affect other over-write behaviors.

A value of 1, allows over-writing of files only when the archive member is newer than the target. If the archive member is older than the target, the file will not be overwritten no matter what the settings of overwrite or onReplace are.

If the archive member is newer than the target, and overwrite is 1, then the file will be over-written. If overwrite is 0, then the behavior depends on the onReplace event handler. If onReplace has an event handler defined, then the file will be overwritten contingent on the result of the event handler. If not, the file will not be overwritten.

The default value is 0.

overwrite

Description:

The overwrite property specifies how duplicate file names should be handled during the Unzip process. The overwrite property is secondary to the onlynewer property if onlynewer is set to 1.

Remarks:

A value of 0 is nilpotent and allows the overwrite behavior to be determined by the onReplace event handler settings. A value of 1 causes an archive member with the same name as the target to overwrite the target. This will also override the onReplace event. The default value is 0.

privilege

Description:

The privilege property controls the use of privileges in Zip and Unzip.

Remarks:

A value of 0 on Zip and Unzip causes no special processing.

On Unzip, a value of 1 causes restoration of Access Control Lists (ACLs) in user mode. A value of 2 causes restoration of ACLs using privileges. On Zip a value of 1 or 2 causes privileges to be used during the operation. The default value is 0.

recreatedirs

Description:

The recreatedirs property controls the creation of directories during the Unzip process.

Remarks:

A value of 0 causes all files from the archive to be written into the directory specified as the folder argument to Unzip without expanding directories written to the archive. A value of 1 recreates sub-directories from the archive. These subdirectories are created within the specified target folder and archive files are written to those subdirectories. The default value is 1.

recursedir

Description:

The recursedir property controls when the specified target folder includes the subdirectories in the archive file during the Zip process.

Remarks:

This property is present in APL+Win v12.1 and newer. A value of 1 includes the subdirectories in the archive. A value of 2 means that you can specify a file pattern in the (included) argument in the Zip method. The default value is 0.

totals

Description:

The totals property is a read-only property containing summary metrics for an archive.

Remarks:

The totals property contains a five element integer vector. The values are: sum of compressed file sizes; sum of uncompressed file sizes; compression ratio; number of archive members; size of comment text.

This property is generated by setting the verbose property to 2 prior to an Unzip operation.

uzoptions

Description:

The uzoptions property is an integer sum of bits which effect the Unzip method.

Remarks:

The 1 bit specifies conversion of space characters to underscore characters in file names.

The 2 bit specifies conversion of carriage return characters to carriage return/linefeed pairs in file content.

The 4 bit specifies case sensitivity in name matching. A value of 0 indicates case sensitive matching. A value of 1 indicates case insensitive matching.

The 8 bit specifies Unzip file test mode.

The 16 bit specifies get zip file information.

The 32 bit specifies freshen only: only write existing files (do not write new files).

The default value is 0.

verbose

Description:

The verbose property controls the generation of messages via the onDisplay event.

Remarks:

A value of 0 generates no messages on Zip and Unzip methods via the onDisplay event. A value of 1 generates some messages on Zip and Unzip methods via the onDisplay event. A value of 2 generates the maximum number of messages on Zip and Unzip methods via the onDisplay event. A value of 2 also generates values for the totals property. The default value is 0.

Info-ZIP copyright and license

This is version 2005-Feb-10 of the Info-ZIP copyright and license. The definitive version of this document should be available at [Info-ZIP License \(sourceforge.net\)](http://www.info-zip.org/Info-ZIP%20License%20(sourceforge.net).txt) indefinitely.

Copyright (c) 1990-2005 Info-ZIP. All rights reserved.

For the purposes of this copyright and license, "Info-ZIP" is defined as the following set of individuals:

Mark Adler, John Bush, Karl Davis, Harald Denker, Jean-Michel Dubois, Jean-loup Gailly, Hunter Goatley, Ed Gordon, Ian Gorman, Chris Herborth, Dirk Haase, Greg Hartwig, Robert Heath, Jonathan Hudson, Paul Kienitz, David Kirschbaum, Johnny Lee, Onno van der Linden, Igor Mandrichenko, Steve P. Miller, Sergio Monesi, Keith Owens, George Petrov, Greg Roelofs, Kai Uwe Rommel, Steve Salisbury, Dave Smith, Steven M. Schweda, Christian Spieler, Cosmin Truta, Antoine Verheijen, Paul von Behren, Rich Wales, Mike White

This software is provided "as is," without warranty of any kind, express or implied. In no event shall Info-ZIP or its contributors be held liable for any direct, indirect, incidental, special or consequential damages arising out of the use of or inability to use this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. Redistributions of source code must retain the above copyright notice, definition, disclaimer, and this list of conditions.
2. Redistributions in binary form (compiled executables) must reproduce the above copyright notice, definition, disclaimer, and this list of conditions in documentation and/or other materials provided with the distribution. The sole exception to this condition is redistribution of a standard UnZipSFX binary (including SFXWiz) as part of a self-extracting archive; that is permitted without inclusion of this license, as long as the normal SFX banner has not been removed from the binary or disabled.
3. Altered versions--including, but not limited to, ports to new operating systems, existing ports with new graphical interfaces, and dynamic, shared, or static library versions--must be plainly marked as such and must not be misrepresented as being the original source. Such altered versions also must not be misrepresented as being Info-ZIP releases--including, but not limited to, labeling of the altered versions with the names "Info-ZIP" (or any variation thereof, including, but not limited to, different capitalizations), "Pocket UnZip," "WiZ" or "MacZip" without the explicit permission of Info-ZIP. Such altered versions are further prohibited from misrepresentative use of the Zip-Bugs or Info-ZIP e-mail addresses or of the Info-ZIP URL(s).
4. Info-ZIP retains the right to use the names "Info-ZIP," "Zip," "UnZip," "UnZipSFX," "WiZ," "Pocket UnZip," "Pocket Zip," and "MacZip" for its own source and binary releases.

Appendix WA: Virtual Key Codes

The tables in this section list the virtual key codes that are used in the `onKeyUp` and `onKeyDown` event-handler properties. These codes allow you to program a Windows application to respond when specific keys are pressed without needing to worry about the underlying keyboard scan codes that may differ by machine.

Virtual Key Codes

Value	Keyboard Equivalent	Value	Keyboard Equivalent
1	Left mouse button	37	Left arrow key
2	Right mouse butt on	38	Up arrow key
3	Control+Break	39	Right arrow key
4	Middle mouse button	40	Down arrow key
5	(undefined)	41	Select key
6	(undefined)	42	(OEM specific)
7	(undefined)	43	Execute key
8	Backspace key	44	Print Screen key
9	Tab key	45	Insert key
10	(undefined)	46	Delete key
11	(undefined)	47	Help key
12	Clear key	48	0 key
13	Enter key	49	1 key
14	(undefined)	50	2 key
15	(undefined)	51	3 key
16	Shift key	52	4 key
17	Ctrl key	53	5 key
18	Alt key	54	6 key
19	Pause key	55	7 key
20	Caps Lock key	56	8 key
21	(reserved for Kanji systems)	57	9 key
22	(reserved for Kanji systems)	58	(undefined)
23	(reserved for Kanji systems)	59	(undefined)
24	(reserved for Kanji systems)	60	(undefined)
25	(reserved for Kanji systems)	61	(undefined)
26	(undefined)	62	(undefined)
27	Esc key	63	(undefined)
28	(reserved for Kanji systems)	64	(undefined)
29	(reserved for Kanji systems)	65	A key
30	(reserved for Kanji systems)	66	B key
31	(reserved for Kanji systems)	67	C key
32	Spacebar	68	D key
33	Page Up key	69	E key
34	Page Down key	70	F key
35	End key	71	G key
36	Home key	72	H key

73 I key

119

F8 key

Value	Keyboard Equivalent	Value	Keyboard Equivalent
74	J key	120	F9 key
75	K key	121	F10 key
76	L key	122	F11 key
77	M key	123	F12 key
78	N key	124	F13 key
79	O key	125	F14 key
80	P key	126	F15 key
81	Q key	127	F16 key
82	R key	128	F17 key
83	S key	129	F18 key
84	T key	130	F19 key
85	U key	131	F20 key
86	V key	132	F21 key
87	W key	133	F22 key
88	X key	134	F23 key
89	Y key	135	F24 key
90	Z key	136	(unassigned)
91	Left Windows key (104-key keyboard)	137	(unassigned)
92	Right Windows key (104-key keyboard)	138	(unassigned)
93	Application key (104-key keyboard)	139	(unassigned)
94	(undefined)	140	(unassigned)
95	(undefined)	141	(unassigned)
96	Numeric keypad 0 key	142	(unassigned)
97	Numeric keypad 1 key	143	(unassigned)
98	Numeric keypad 2 key	144	Num Lock key
99	Numeric keypad 3 key	145	Scroll Lock key
100	Numeric keypad 4 key	146	(unassigned)
101	Numeric keypad 5 key	147	(unassigned)
102	Numeric keypad 6 key	148	(unassigned)
103	Numeric keypad 7 key	149	(unassigned)
104	Numeric keypad 8 key	150	(unassigned)
105	Numeric keypad 9 key	151	(unassigned)
106	Multiply key	152	(unassigned)
107	Add key	153	(unassigned)
108	Separator key	154	(unassigned)
109	Subtract key	155	(unassigned)
110	Decimal key	156	(unassigned)
111	Divide key	157	(unassigned)
112	F1 key	158	(unassigned)
113	F2 key	159	(unassigned)
114	F3 key	160	(unassigned)
115	F4 key	161	(unassigned)
116	F5 key	162	(unassigned)
117	F6 key	163	(unassigned)
118	F7 key	164	(unassigned)

165 (unassigned)

211 (unassigned)

Value	Keyboard Equivalent	Value	Keyboard Equivalent
166	(unassigned)	212	(unassigned)
167	(unassigned)	213	(unassigned)
168	(unassigned)	214	(unassigned)
169	(unassigned)	215	(unassigned)
170	(unassigned)	216	(unassigned)
171	(unassigned)	217	(unassigned)
172	(unassigned)	218	(unassigned)
173	(unassigned)	219	(OEM specific)
174	(unassigned)	220	(OEM specific)
175	(unassigned)	221	(OEM specific)
176	(unassigned)	222	(OEM specific)
177	(unassigned)	223	(OEM specific)
178	(unassigned)	224	(OEM specific)
179	(unassigned)	225	(OEM specific)
180	(unassigned)	226	(OEM specific)
181	(unassigned)	227	(OEM specific)
182	(unassigned)	228	(OEM specific)
183	(unassigned)	229	(unassigned)
184	(unassigned)	230	(OEM specific)
185	(unassigned)	231	(unassigned)
186	(OEM specific)	232	(unassigned)
187	(OEM specific)	233	(OEM specific)
188	(OEM specific)	234	(OEM specific)
189	(OEM specific)	235	(OEM specific)
190	(OEM specific)	236	(OEM specific)
191	(OEM specific)	237	(OEM specific)
192	(OEM specific)	238	(OEM specific)
193	(unassigned)	239	(OEM specific)
194	(unassigned)	240	(OEM specific)
195	(unassigned)	241	(OEM specific)
196	(unassigned)	242	(OEM specific)
197	(unassigned)	243	(OEM specific)
198	(unassigned)	244	(OEM specific)
199	(unassigned)	245	(OEM specific)
200	(unassigned)	246	(unassigned)
201	(unassigned)	247	(unassigned)
202	(unassigned)	248	(unassigned)
203	(unassigned)	249	(unassigned)
204	(unassigned)	250	(unassigned)
205	(unassigned)	251	(unassigned)
206	(unassigned)	252	(unassigned)
207	(unassigned)	253	(unassigned)
208	(unassigned)	254	(unassigned)
209	(unassigned)	255	(unassigned)
210	(unassigned)		

INDEX

#

#, 138–39

A

Abort (method), 320
access keys, 83, 148, 416
action
 defined, 392
Action, onAction (event), 273, 481, 483, 484, 488, 489, 490
Activate, onActivate (event), 274
active form (property), 142
ActiveControl, 463
 methods and events, 466
 properties of, 464
ActiveObject, 463
 methods and events, 467
 properties of, 467
ActiveX
 array properties, 463, 465
 enumerating collections, 474
 license, 464, 467
 multiple index notation, 465
 server object, 467
 typed values, 469
AddImages (method), 320
AddRows (method), 320, 438
ADF startup files for `OWCALL`, 516
alias, 198, 435
alignment, 19, 39, 66, 104, 109, 112, 114, 321
alternate (polyfill mode), 205
always on top, 52
APL characters, 429
APL fonts, 182
Arrange (method), 320
array properties, 269
associated function, running (`Ona`), 512
AutoFit (method), 321

B

background color, 152
bars (read-only property), 143
BeginLabelEdit (method), 322
BeginLabelEdit, onBeginLabelEdit (event), 274

behavior
 programming alert, 492
bin (property), 144
bins (read-only property), 144
Bitmap (operation), 333
bitmap (property), 145
bitmaps, 55, 92, 193, 333, 437, 445, 446
 device dependent, 155
bitmapsizes (read-only property), 145
border (property), 145–47
Brush (operation), 334
brushes, 442, 446
buddy (property), 147
Button, 18–21, 404

C

calendar, 37
callback fields, 44
callbacks, 411, 424, 425, 522, 530, 531, 536
callouts, 425
cancel button, 19
CancelLabelEdit (method), 322
canpaste (property), 147
canredo (property), 147
canundo (property), 104, 148
caption (property), 148, 427, 449
captiondock (property), 149
captionfloat (property), 149
cascading
 forms, 321
 menus, 83
Change, onChange (event), 17, 275, 449, 450, 452
channelwhere (read-only property), 149
character units, 413
CharToLine (method), 322
Check, 21–23, 404
check boxes, 74, 130
children (read-only property), 150, 413
Chord (operation), 335
Circle (operation), 335
class (read-only property), 150, 413
class (window)
 registering, 532, 533
classes
 user-defined. *See* user-defined class
classes (read-only property), 151

- classes, built-in
 - introduced, 390
 - modifying with `onNew`, 487
 - modifying without `onNew`, 488
 - specifying class to be modified, 487
- Clear (operation), 336
- Click (method), 323
- Click, `onClick` (event), 276, 417
- client
 - APL as, for DDE, 449
 - CommandBar as, 169
 - for ActiveX server, 463
- client area, 151, 158, 178, 193, 212, 215, 224, 238, 265, 266
- `clientwnd` (read-only property), 151
- `clientwhere` (property), 151
- Clip (operation), 337
- Close (method), 323, 411, 427
- Close button, 50, 146
- Close, `onClose` (event), 277, 424
- closed objects, 411
- `CloseUp`, `onCloseUp` (event), 278
- `clsid` (property), 152
- coercing the data (`⌘na`), 510
- `ColClick`, `onColClick` (event), 278
- collate (property), 152
- collection, 472
- color
 - background, 152
 - composite, 153
 - foreground, 152
- Color (operation), 338
- color (property), 152–54
- `colordepth` (property), 155
- `ColResize`, `onColResize` (event), 278
- `columndisplay` (property), 156
- Combo, 23–26, 405
- Command (method), 324
- CommandBar, 26–35, 404
 - properties (other), 32
 - properties affecting docking, 28
 - properties affecting floating bars, 31
 - properties affecting size, 29
 - uses, 418
- CommandButton, 35–37, 404
- compatibility flags, 527
- Component Object Model (COM), 463
- composite color, 153
- constants, defining (`⌘wcall`), 527
- content (property), 157, 466
- context sensitive help, 296
- `ContextHelpMode` (method), 325
- controller, 463
- controls, 403
 - containers for, 403

- introduced, 393
- invisible (objects), 404
- multiple alternatives, 405
- passive, 403
- positioning, 413
- range, numeric or physical, 405
- single choice, 404
- sizing, 413
- special purpose, 406
- user input, 405

- `copies` (property), 158
- `Copy` (method), 325
- `count` (read-only property), 158
- `Create` (method), 325, 411, 436
- creating objects. *See* windows interface
- cursor
 - animated files, 179, 188
 - location of, 197, 228
 - over a target, 173
- cursors, 56, 194
- `Cut` (method), 326

D

- data (property), 157, 158
- datatypes
 - ActiveX, 385, 469
 - APL and Windows, 528
 - `⌘na`, 509
 - `⌘wcall`, 518, 525
- `DateTime`, 37–47, 406
 - customizing the display of, 43
- `DbClick`, `onDbClick` (event), 279
- `ddeApplication` (property), 161, 453
- `DdeConnect`, `onDdeConnect` (event), 280, 448, 450, 453
- `DdeDisconnect`, `onDdeDisconnect` (event), 280, 448, 451, 453
- `DdeExecute` (method), 326, 452, 454
- `DdeExecute`, `onDdeExecute` (event), 280, 448, 449, 454
- `ddeItem` (property), 161, 449, 452
- `ddeMode` (property), 162, 451, 453
- `DdePoke` (method), 327, 452
- `DdeRequest` (method), 327, 452
- `ddeTimeout` (property), 162, 450, 453
- `ddeTopic` (property), 162, 448, 450, 453
- `def` (property), 164, 396, 417
- default (property), 165
- default button, 19
- `Defer` (method), 328, 481, 485, 486
- `deferexit` (property), 168
- definition, constructing a (`⌘na`), 509
- `Delete` (method), 329, 411, 436
- `Delete`, `onDelete` (event), 281
- `DeleteCols` (method), 501, 502, 503, 504, 505
- `DeleteNodes` (method), 330

DeleteRows (method), 330
 DEMOCALC workspace, 458
 DEMODRAW workspace, 458
 DEMOWIN workspace, 536
 dense set, 419
 description (read-only property), 168
 Destroy, onDestroy (event), 281
 device-dependent bitmap, 155
 DevMode (method), 330
 DevNames (method), 331
 dialog boxes, 426
 dialog units, 413
 disabled objects, 412
 DLL (Dynamic Link Library), 59, 507, 508, 509, 512, 514, 519
 dock (property), 169
 dockable (property), 169, 195
 dockbreak (property), 170
 DockChange, onDockChange (event), 282
 dockheader (property), 170
 dockideal (property), 170
 docking, 28
 docklength (property), 170
 dockmargin (read-only property), 171
 dockmax (property), 171
 dockmin (property), 171
 dockmindepth (property), 172
 dockside (property), 172
 dockvert (read-only property), 172
 DragContinue, onDragContinue (event), 282, 437
 DragEnd, onDragEnd (event), 283, 437
 DragEnter, onDragEnter (event), 283
 DragFeedback, onDragFeedback (event), 284, 437
 dragimage (property), 173
 DragLeave, onDragLeave (event), 285, 437
 DragOver, onDragOver (event), 285, 437
 dragsource (read-only property), 173
 DragStart, onDragStart (event), 286, 437
 Draw (method), 331, 428, 439–46
 Drop, onDrop (event), 286, 437, 438
 dropcolor (property), 174
 drop-down

- calendar, 38
- Combo, 25
- CommandBar, 37
- menu, 84

 DropDown, onDropDown (event), 287, 417
 dropfont (property), 174
 droptarget (read-only property), 174
 Dynamic Data Exchange (DDE), 447–51
 Dynamic Link Library. *See* DLL

E

edge (property), 174

Edit, 47–49, 405
 Eject (method), 347, 427
 Ellipse (operation), 338
 empty (read-only property), 175
 enabled (property), 175, 412, 417
 enabled objects, 412
 EndLabelEdit, onEndLabelEdit (event), 290
 EnhancedWCALL compatibility switch, 527, 528
 EnsureVisible (method), 351
 enter text, 47–49
 entry point, 507, 512, 514, 517, 520, 532
 EnumEnd (method), 351
 enumerating collections, 474
 EnumNext (method), 351
 EnumStart (method), 352
 Error, onError (event), 290
 Event (method), 352, 481, 485, 486
 event handler, 271, *See also* under name of each event

- creating, 390
- creating an onAction, 483
- creating an onNew, 481

 events

- introduced, 390

 events (read-only property), 178, 413
 Excel

- used as ActiveX server, 471

 Exec (method), 353
 exit APL, 324
 Exit, onExit (event), 291
 ExitError, onExitError (event), 291
 Expand (method), 353
 Expanded, onExpanded (event), 292
 Expanding, onExpanding (event), 292
 explicit results ($\Box a$), 512
 extent (property), 178, 266, 414
 external routine, 507, 508, 511

F

file (property), 179
 File, onFile (event), 292
 filter

- creating, 532
- option specifications ($\Box wcall$), 524
- patterns, defining ($\Box wcall$), 522
- patterns, specification of ($\Box wcall$), 523

 FindNode (method), 354
 firstday (property), 179
 fixedtabsize (property), 180
 floating command bar, 31
 floating tool palette, 51, 147
 floatlayout (property), 180
 floatsize (property), 180
 floatwhere (property), 181

focus, 393, 417
Focus (method), 354
focus (property), 181
Focus, onFocus (event), 293
Font (operation), 338
font (property), 181–83, *See Also* selfont
 LOGFONT, 181
 WChooseFont, 182
foreground color, 152
Form, 49–53
format (property), 183
FormatQuery, onFormatQuery (event), 293
FormatString, onFormatString (event), 293
forms
 position and sizing, 414
Frame, 53–54
function call (`□wcall`)
 specification options, 519
 syntax, 517

G

Get (method), 355
GetCells (method), 356
GetDayState, onGetDayState (event), 294
GetImages (method), 356
GetInfo (method), 356, 438
GetPos (method), 357
GetRows (method), 358, 438
GetVirtualData, onGetVirtualData (event), 294, 295
graphical user interface (GUI). *See* windows interface
gripper, 27
GUID (globally unique identifier), 150, 152, 463, 464, 467, 468

H

hdc (property), 184
help
 ActiveX, 469
help (property), 185
help, in-session, 492
Help, onHelp (event), 296
helpcontext (property), 185
hidden objects, 411
Hide (method), 359
Hide, onHide (event), 297
highlightbold (property), 185
highlightcut (property), 185
highlightdrop (property), 186
highlightfocus (property), 186
highlightselect (property), 186
himage (read-only property), 186
HitTest (method), 359
HotChange, onHotChange (event), 298

hwnd (read-only property), 187
hwndddropdown (read-only property), 187
hwndmain (read-only property), 187
hwndsplash (property), 187
HwndToName (method), 360

I

Icon (operation), 339
icon (property), 188
icons, 56, 188, 194, 339, 445
image (property), 188
imagealloc (property), 189
imagecount (read-only property), 189
imageindex (property), 189
Imagelist, 55–59
Imagelist (operation), 340
imagelist (property), 190
imagelistdisabled (property), 190
imagelisthot (property), 191
imagelistlarge (property), 191
imagelistuser (property), 191
imagenames (property), 192
imagesize (property), 193
imagespace (property), 194
increment (property), 194
indent (property), 195
info tip. *See* tooltip, item-specific
initialization file (INI)
 `□wcall`, 516
InsertNodes (method), 361
InsertRows (method), 362
instance (property), 195
interface (property), 196, 468
interval (property), 196

J

justification, 20, 69

K

key codes, 298, 554
keyboard, 420
 character, action based on, 421
 handling, 420
 input, 420
 physical key, action based on, 421
KeyDown, onKeyDown (event), 298, 420
KeyPress, onKeyPress (event), 299, 420, 421, 431
keys (property), 196, 420
KeyUp, onKeyUp (event), 300, 420

L

Label, 59–61
 label tips. *See* tooltip, Listview label
 labeledithwnd (read-only property), 197
 labels, editing, 274, 290
 largeimage view, 65
 limit (property), 197
 Limit, onLimit (event), 301
 limitwhere (property), 198
 Line (operation), 340
 LineToChar (method), 362
 links (property), 198, 435, 436
 List, 61–63, 405
 list (property), 199–201
 list view, 65
 Listview, 65–75, 405, 438
 LOGFONT, 181

M

Make function. *See* Jmake (user command)
 margin (property), 202
 maskcolor (property), 202
 masked imagelist, 56
 maximize button, 50, 146
 maximized form, 414
 MDIForm, 75–77
 Media, 77–83, 406
 mediahwnd (read-only property), 202
 mediatype (property), 203
 Memory, onMemory (event), 301
 Menu, 83–87
 menus
 access keys, 416
 alternatives to, 418
 behavior, 404
 changing, 417
 popup, 418
 separators, 416
 uses, 415
 MergedCells (property), 496, 497, 498, 499, 500
 message box, 426
 methods
 introduced, 390
 methods (read-only property), 203, 413
 minimize button, 50, 146
 minimized form, 414
 minsize (read-only property), 203
 missing (property), 203
 modal form, 393, 400
 Mode (operation), 340
 mode (property), 204–6, 444
 modified (property), 206

Modified, onModified (event), 301
 Modify (method), 362
 modifystop (property), 206
 monthdelta (property), 207
 mouse
 handling, 422
 input, 420
 Mouse, onMouse (event), 302, 422
 MouseDouble, onMouseDouble (event), 303
 MouseDown, onMouseDown (event), 303, 418
 MouseDrag, onMouseDrag (event), 304
 MouseEnter, onMouseEnter (event), 305
 MouseLeave, onMouseLeave (event), 305
 MouseMove, onMouseMove (event), 305
 MouseUp, onMouseUp (event), 306
 Move, onMove (event), 306
 msversion (property), 207

N

name (property), 208
 name association. *See* `Onna`
 names for constants (`Onwcall`), 515
 naming
 relative, 397
 using links, 435
 windows objects, 391
 New (method), 363, 411, 436
 New, onNew (event), 306, 481, 486
 newclasses (property), 208, 481
 noredraw (property), 208
 normsize (read-only property), 209
 Notify (method), 364
 null (property), 209

O

obj (property), 209, 468
 objects
 contents, 413
 creating, 411
 creating from precompiled definition, 396
 internal definition, 157, 164
 naming, 391
 windows user interface, 17
 octothorpe, 138–39
 Open (method), 364, 411, 423
 open objects, 411
 Open, onOpen (event), 307
 opened (property), 412
 opened (read-only property), 210
 Option, 87–89, 404
 order (property), 211, 419
 orientation (property), 212

origin (property), 212
output arrays (`Outa`), 512
overlay images, 55, 67
overlays (property), 212

P

padding (property), 212
Page, 89–91
page (read-only property), 213
Page Setup (method), 366
Page, onPage (event), 308, 428
pages (read-only property), 213
Paint (method), 366
Paint, onPaint (event), 308
palette (property), 213
PaneWhere (method), 367
paper (property), 214
paper tray, 144
papers (read-only property), 214
parameter notation (`Param`), 519
parameters, specifying (`Para`), 510
parentage, 396, 398, 404
Paste (method), 367
pattern specification (`Pattern`), 522
Pen (operation), 341
pending (property), 214
pens, 442, 446
persistence, 466
Picture, 91–94
place (property), 214, 414
Play (method), 367
Point (operation), 342
pointer (property), 215
points, 413
Poly (operation), 342
polyfill mode, 205
Pop (method), 368, 430
Popup (method), 368–71, 418
popup menu, 418
position (property), 216
poslargeimage (property), 216
possmallimage (property), 217
Print (method), 371, 427, 428
PrintDlg (method), 371
printer
 location on page, 216
 margins, 202
 name of, 217
 number of copies, 158
 orientation, 212
 page, 213
 paper size, 214
Printer, 94–99, 427

printrname (property), 217
Printers (method), 373, 429
procedure (window), 532
progid (property), 218
programmatic identifier, 463, 464, 467, 468
Progress (progress bar), 99–101
Promotion compatibility switch, 527
prompt (property), 218
Prompt, onPrompt (event), 309
properties
 introduced, 390
 windows interface, 141
properties (read-only property), 218, 413
proportional scaling, 413
Protected, onProtected (event), 309
Push (method), 374, 430
Put (method), 374

R

radio button (Option), 87–89
radio-style menu, 85
range (property), 219
rasterop mode, 204
Rebars. *See* CommandBar
Rect (operation), 342
recursions (property), 219
redirection syntax
 ActiveX, 471
redoname (property), 220
Ref (method), 375
registration
 new window class, 531, 533
relative naming, 397
ReleaseObjects (method), 376
rendering, 347
rendering operations, 332, 337, 441
Reopen, onReopen (event), 310
report view, 65
ReqResize (method), 376
ReqResize, onReqResize (event), 310
Reset (method), 376
Resize (method), 376
Resize, onResize (event), 310, 444
Resource, onResource (event), 311
restored form, 414
result notation (`Result`), 519
RichEdit, 102–8, 207, 405, 437
roworigin (read-only property), 222
rtf (property), 223

S

savecontent (property), 223, 466

- Scale (operation), 343
 - scale (property), 223, 413, 443, 444
 - scaling units, 413
 - Scroll (scroll bar), 109–10, 405
 - Scroll, onScroll (event), 311
 - scrollaccel (property), 225
 - ScrollCaret (method), 377
 - scrollmargin (property), 225
 - searchstring (read-only property), 225
 - Seek (method), 377
 - Seg (operation), 343
 - selalign (property), 226
 - selbullet (property), 226, 227, 233
 - SelChange, onSelChange (event), 312
 - selcolor (property), 228
 - Select, onSelect (event), 312
 - selection (property), 228
 - Selector, 111–14
 - self (property), 228
 - selfont (property), 229
 - selfontall (read-only property), 230
 - selindents (property), 230
 - selnumstart (property), 231
 - selnumstyle (property), 231
 - selnumtab (property), 232
 - selfpargall (read-only property), 232
 - selrtf (property), 233
 - selstyle (property), 233
 - selstyleall (read-only property), 234
 - seltabs (property), 234
 - seltext (property), 234
 - Send (method), 358, 377
 - Send, onSend (event), 312
 - separator, 22, 35, 86, 89, 416
 - separator (property), 235, 416
 - server
 - ActiveX, 463, 467, 471
 - APL as, for DDE, 448
 - Excel used as ActiveX, 471
 - server (read-only property), 235
 - Set (method), 378
 - SetCells (method), 379
 - SetImages (method), 379
 - SetInfo (method), 379
 - SetLinks (method), 380, 435, 436
 - SetPos (method), 380
 - SetRows (method), 381
 - SetStatus (method), 381
 - Setup (method), 381
 - shortcut (property), 236, 416
 - shortcut key, 84, 148, 416
 - Show (method), 381, 423, 424
 - Show, onShow (event), 313
 - ShowNode (method), 382
 - showtext (property), 237
 - siblings (read-only property), 237
 - size (property), 238, 265, 414, 439
 - sliderlen (property), 238
 - sliderwhere (read-only property), 238
 - smallimage view, 65
 - Sort, onSort (event), 313
 - SortChildren (method), 382
 - sortorder (property), 239
 - sourceformats (property), 239, 437
 - Spin, onSpin (event), 313
 - Spinner, 114–17, 405
 - splash screen, 187
 - Start, onStart (event), 314
 - startup files. *See* initialization file
 - state (read-only property), 239
 - state operations, 332, 337, 346, 441
 - status (property), 240
 - Status (status bar), 117–20
 - Stepit (method), 383
 - Stop (method), 383
 - Stop, onStop (event), 314
 - style (property), 241–50
 - subclass, 531, 534, 535
 - suppress (property), 250
 - system menu, 50, 146
 - system object, 138–39
 - SystemChange, onSystemChange (event), 314
- ## T
- tabgroup (property), 251
 - tabparent (property), 251
 - tabrows (read-only property), 251
 - tabstop (property), 252
 - targetformats (property), 252, 437
 - Text (operation), 344
 - text (property), 253, 449
 - tickinterval (property), 254
 - tickpos (read-only property), 255
 - ticks (property), 255
 - tiling, 55, 93, 321
 - time (date), 37–47
 - time selector, 39
 - Timer, 120–21
 - Timer, onTimer (event), 315
 - today (property), 255
 - toggle button, 20, 36
 - toggle menu, 85
 - Toolbox, 121–23, 405
 - tooltip
 - item-specific, 72
 - Listview label, 72
 - pane-specific, 119, 241

tooltip (property), 256
Tooltip, onTooltip (event), 289, 315
tooltipenabled (property), 256
tooltipwidth (property), 258
Trackbar, 124–26, 406
translate (property), 258
transparency mode, 204
Tree, 127–38, 405, 438
twips, 413
typed values, 469

U

UCMDSW file, 459
Undo (method), 375, 383
undolimit (property), 260
undoname (property), 260
Unfocus, onUnfocus (event), 316
Unicode, 344–46
units (read-only property), 261, 413
Update, onUpdate (event), 316
UseObject (method), 384
user command file
 UCMDSW, 459
user state images, 65, 74, 128
user-defined class
 controlling behavior of, 489
 creating, 481
 creating events for, 484
 programming style for, 487
user-defined property, 268, 486
UserKeyDown, onUserKeyDown (event), 317

V

Validate (method), 384
Validate, onValidate (event), 317
value (property), 261–63
variant type (VT), 384, 468, 470, 472
variant value (VV), 470
version, 463, 464, 468
version (read-only property), 264
viewalign (property), 264
viewmode (property), 264
virtual functions, 514, 520
virtual key codes, 298, 554
visible (property), 264, 412
visible objects, 411
VT (method), 384
VT (variant type), 384, 468, 470, 472
VV (method), 386
VV (variant value), 470

W

Wait (method), 386, 423, 424
Wait, onWait (event), 318, 423
WChooseFont, 182
where (property), 265–66, 413, 414
width (property), 266
WINAPI workspace, 459, 536
winding (polyfill mode), 205
window
 callbacks, other, 536
 classes, 532
 creating a, 534
 handle, 514, 521, 528, 529, 530, 534, 535
 procedures, 532
 processing messages, 535
 subclassing a, 534
 waiting for messages, 535
Window (operation), 346
Window menu, 84
Windows Control Panel, 153
windows interface
 action, 392
 basics, 393
 class, 390
 classes, 17
 creating objects, 395
 event handlers, 390
 event-handlers, 271
 events, 390
 methods, 319, 390
 naming objects, 391
 objects, 17
 overview, 389
 properties, 17, 141, 390
 $\square$ wi, 389, 391
WINDOWS workspace, 58, 123, 426, 460
workarea (property), 414
workarea (read-only property), 267
workspaces
 WINDOWS, 58, 123, 426
 DEMOCALC, 458
 DEMODRAW, 458
 DEMOWIN, 536
 WINAPI, 459, 536
 WINDOWS, 460
 XLCHART, 453
wrap (property), 267

X

xclasses (property), 268, 463
XInfo (method), 387, 467
XLCHART workspace, 453

Z

zoom (property), 268

]

]wmake (user command), 401, 459, 488

□

□na, 508–14

associated functions, running, 512

coercing the data, 510

constructing a definition, 509

datatypes, 509

examples using, 511

explicit results, 512

output arrays, 513

parameters, specifying, 510

syntax, 508

□warg, 392

□wcall, 424, 425, 514–36, 528, 530, 531, 534, 535

constants, defining, 527

datatypes, 518

datatypes, defining, 525

filter option specifications, 524

filter parameter specifications, 523

filter patterns, defining, 522

function call specification options, 519

function call specifications, examples of, 521

function call syntax, defining, 517

names for constants, 515

parameter and result notation, 519

pattern specification, 522

startup files, 516

using, 514

□wevent, 392

□wgive, 424, 425

□wi, 192, 193, 389, 391, 424, 425

□wres, 393

□wself, 391, 392, 398