The NeWS Toolkit Reference Manual

Sun Microsystems, Inc.
2500 Garcia Avenue
Mountain View, CA 94043

Part No: 800-5543-10
Revision A, of March, 1991

Contents

Figures

Tables

Preface

The NeWS Toolkit (TNT) is an object-oriented programming system based on the PostScript language and NeWS. TNT implements many of the OPEN LOOK interface components required to build the user interface of an application.

Intended purpose of this manual

The *NeWS Toolkit Reference Manual* is intended to provide a comprehensive compendium of the Toolkit's API. It is not intended to provide a conceptual framework for the way the Toolkit works. That task is left to a future programmer's guide.

What you need to know

The NeWS Toolkit sits atop much functionality which you should be familiar with before you attempt to use the Toolkit. You should be familiar with:

- The PostScript language
- Object-oriented programming
- NeWS

How the Reference Manual is organized

The NeWS Toolkit Reference Manual is organized alphabetically by class. With some exceptions each class has its own chapter. Within each class's chapter the methods are grouped in sections by functionality, e.g., Geometry, Painting, etc.

TNT's class tree

As noted above The NeWS Toolkit is an object-oriented system and as such TNT subclasses inherit from their superclasses, which inherit from their superclasses and on up the class tree (see the class tree illustration on page xxxiii).

In addition, at the beginning of each class's chapter there is an illustration of that class's subtree. For example, at the beginning of the ClassItemGroup chapter you will find the following illustration:

If a class is shown in parentheses it means that it is mixed-in to the class directly below it. (See *Mixin class defined*, below.)

Mixin class defined

A mixin class is an "abstract" superclass that contains functionality applicable to a broad range of classes. Mixin classes are used to implement multiple inheritance. ClassControl is an example of this type of class. Mixin classes have only ClassObject (the root of the tree) as a superclass. Mixin classes are not intended to be instantiated.

Rules followed for documenting superclass methods

In general the Reference Manual follows the following rules for documenting inherited methods:

1. Subclasses document only those methods inherited from non-mixin superclasses that are overridden in the subclass.

2. Subclasses document the methods they inherit directly from mixin classes whether the methods are overridden or not.

How the API is represented

The methods, variables and utilities in the Reference Manual are documented using the following syntax:

- The names of the methods, variables and utilities are flush with the left margin to allow for easy scanning for a particular name.
- Methods have the following syntax:

  arguments /**name** return values

  A hyphen (-) indicates either no arguments or no return values.
- Variables take no arguments so only their value is given. In addition, variables are marked as such:

  /**variable** value (Variable)
- Utilities defined in systemdict are used directly rather than sent to an object. They are shown in the Reference Manual without the beginning slash (/) that denotes methods and variables:

  arguments **Utility** return values

What's in the chapters

Chapter 1, Introduction—an explanation of some of the general concepts of the toolkit.

Chapter 2, ClassBag—the API for the Toolkit's most basic class designed to manage a collection of clients.

Chapter 3—ClassBorderBag—the API for a bag designed to manage a specific number of clients.

Chapter 4, Buttons—contains the API for ClassButtons and ClassMenuButtons.

Chapter 5, ClassCanvas—contains the API for canvases, the substructure of almost everything you see on the screen (the look) as well as the API for linking toolkit objects to user input.

Chapter 6, ClassControl—contains the generalized architecture for toolkit controls. ClassControl is a mixin class (see above) and establishes a common API for controls(e.g., scrollbars, buttons, etc.)

Chapter 7, DisplayItems—contains the API for a set of drawing procedures. Display items are not a class

Chapter 8, Gauges—contains the API for gauges and documents both horizontal gauges (ClassHGauge) and vertical gauges (ClassVGauge).

Chapter 9, ClassItemGroup—ClassItemGroup is a utility class that helps reduce the number of clients that need to be added to a bag.

Chapter 10, ClassLabel—contains the API for labels used to identify controls and other objects.

Chapter 11, ClassLayout—a mixin class that is designed to ensure a consistent approach to layout in panels (Chapter 16) and item groups (Chapter 9).

Chapter 12, ClassMenu—the API for the Toolkit's menus.

Chapter 13, ClassNotice—the API for OPEN LOOK notices.

Chapter 14, ClassNumericField—the API for numeric fields.

Chapter 15 ClassObject—the root of the NeWS class hierarchy.

Chapter 16, ClassPanel—contains the API for panels, a subclass of ClassBag, designed to provided a control surface for placing canvases and controls.

Chapter 17, ClassRegion—regions are the toolkit's lightweight canvas replacement.

Chapter 18, Scrollbars—contains the API for OPEN LOOK scrollbars and documents both vertical scrollbars (ClassVScrollbar) and horizontal scrollbars (ClassHScrollbar).

Chapter 19, ClassScrollList—contains the API for the Toolkit's OPEN LOOK scrolling lists.

Chapter 20, ClassSelection—contains the API for selections.

Chapter 21, ClassSettings and ClassCheckboxes—contains the API for OPEN LOOK settings and checkboxes.

Chapter 22, Sliders—contains the API for OPEN LOOK sliders and documents both horizontal sliders (ClassHSlider) and vertical sliders(ClassVSlider).

Chapter 23, ClassTextCanvas—a canvas subclass that provides minimal assistance for clients whose selections are character strings and want to use an overlay canvas for dragging animation.

Chapter 24, ClassTextField—contains the API for OPEN LOOK text fields.

Chapter 25, Windows—contains the API for the Toolkit's window architecture. This chapter documents the API for ClassWindow, ClassBaseWindow and ClassPopupWindow.

Chapter 26, The Wire Service—documents the c-side library that provides for server-client communications. The wire service is an extension of the NeWS CPS facility. (See the NeWS 2.1 Programmer's Guide for information on CPS.)

Chapter 27, Jot—documents the c-side library that implements the toolkit's text facility.

Index—a subject matter index.

Class Index—an index that provides a listing of the Toolkit's methods by class.

Use of fonts

Type	Font
/method, **/Method** and **/Variable**	**helvetica bold 9 pt**
argument and /DictKey	helvetica regular 9 pt
`C code`	`courier regular 9 pt`

Subclassing issues

Throughout the documentation you will find references to subclassing. The ability to use the provided classes as a solid base for any specialized classes you want to build is one of the strengths of an object-oriented system. In order to understand how NeWS implements its classing system you should read the *NeWS 2.1 Programmer's Guide*, Chapter 5, Classes.

The NeWS Toolkit has lower case methods and mixed case methods. Lower case methods are "public"; mixed case methods are "private." You send the lower case methods to your objects and subclass the mixed case methods. In rare cases you will subclass lower case methods (e.g., **/validate**). In addition, the lower case methods are used for inter-object sends—e.g., when you want pressing a button to paint a canvas. The mixed case methods are called within the classes themselves.

Strict subclass responsibility

In TNT some methods are intended to be implemented by subclassers; some of these "subclasser" methods have default implementations; some do not. In general, those methods that have no default implementation are designated as "Strict subclass responsibility." If you instantiate a class that contains strict subclasser methods, when the strict subclass responsibility method is sent you get an error. ClassSelection is an example of one such class that contains strict subclass responsibility methods.

Subclass responsibility

Some classes in the TNT hierarchy contain methods that are designated as subclass responsibility methods but have default implementations. The defaults are generally "uninteresting" but, unlike "strict subclass responsibility" methods, subclass responsibility will not cause an error if you don't override them.

Subclasser methods

In order to provide for consistency in interfaces, and methods for subclassers to override, some methods call other methods. Having methods call other methods allows the Toolkit to establish protocols that make using the Toolkit easier. In addition, these protocols allow the Toolkit to ensure that operations like painting are performed in the correct context. Painting canvases is an example of a protocol established in this manner. See Chapter 5, ClassCanvas, *Painting*.

Utility methods

The final category of mixed case methods are utility methods. Utility methods are convenience functions that provide useful functionality. You can use them in your subclasses but you never need to subclass them. In those classes where they are defined, utility methods have their own section.

Variables

In addition, TNT makes use of variables that set some property (e.g., colors in a canvas). You can subclass a TNT class to set this property to be the same for all instances. For example if you wanted all your instances of ClassCanvas to be pink you could subclass ClassCanvas and put the color pink into the relevant class variable.

Demo methods

Many of The NeWS Toolkit classes include demo methods that demonstrate the functionality of the class. You can see the demo by sending the demo method to a class. For example, to see a demo of buttons in a window you would do:

```
/demo ClassButtons send
```

When you send the demo method to a class that has a demo the demo appears on the screen and the window the demo is in and the object being demonstrated (or an array of objects) are placed on the stack. Putting the object on the stack allows you to get a handle on the object and "play" with it. For example, in order to get a handle on the window that is created when you send **/demo** to ClassBaseWindow you could do:

```
/win /demo ClassBaseWindow pop def
```

The default for TNT is to load all the demo code when the toolkit is loaded. You can change this default by setting **/IncludeDemos?** to false in your .startup.ps file. That is, you would do:

```
/IncludeDemos? false def
```

in your .startup.ps file.

Programming notes

This section contains some important notes on programming in the Toolkit. These notes have importance throughout the Toolkit.

Modifying returned arrays

Many TNT methods return arrays that, due to the nature of composite objects in PostScript (see the *PostScript Language Reference Manual*), you should never modify. You can *use* the array in a nondestructive way. For example, you could "forall" over the array. If you do need to modify a returned array you should make a copy of it first. One way to safely copy an array is:

```
dup length array copy
```

Using super in overrides

You should typically use the NeWS pseudo-variable, super, in your overrides of TNT methods. For example if you override a definition of a TNT method you should do something like:

```
/tnt'smethodname {
your overriding definition
/tnt'smethodname super send
} def
```

Doing a "super send" ensures that you are getting the benefit of any manipulations that the toolkit does in super classes. See the *NeWS 2.1 Programmer's Guide* for more information on super.

Getting the location of events

TNT uses NeWS events to distribute information (e.g., that a mouse button went down over your object. Often the most interesting piece of information that the event carries is its location. You can easily get the location of an event that is delivered to your object by doing the following:

```
begin
XLocation YLocation
end
```

which puts the x,y coordinates of the event on the operand stack.

Bibliography

The following is a partial list of books that cover subjects related to understanding and programming in The NeWS Toolkit.

NeWS

News 2.1 Programmer's Guide

UNIX Networking, Chapter 10, Networking NeWS

PostScript

PostScript Language Reference Manual from Adobe

Inside PostScript (Merritt, Braswell)

Understanding PS programming

PostScript Programmer's Reference Guide

Real World PostScript

The PostScript Journal

Object-Oriented Programming and SmallTalk:

Object-Oriented Programming on the Macintosh (Schmucker)

A Taste of SmallTalk (Kaehler & Patterson)

An Introduction to Object-Oriented Programming and SmallTalk (Wiener & Pinson)

OOP in Common Lisp (Keen)

Object Oriented Software Construction (Meyer)

C++ Primer (Lippman)

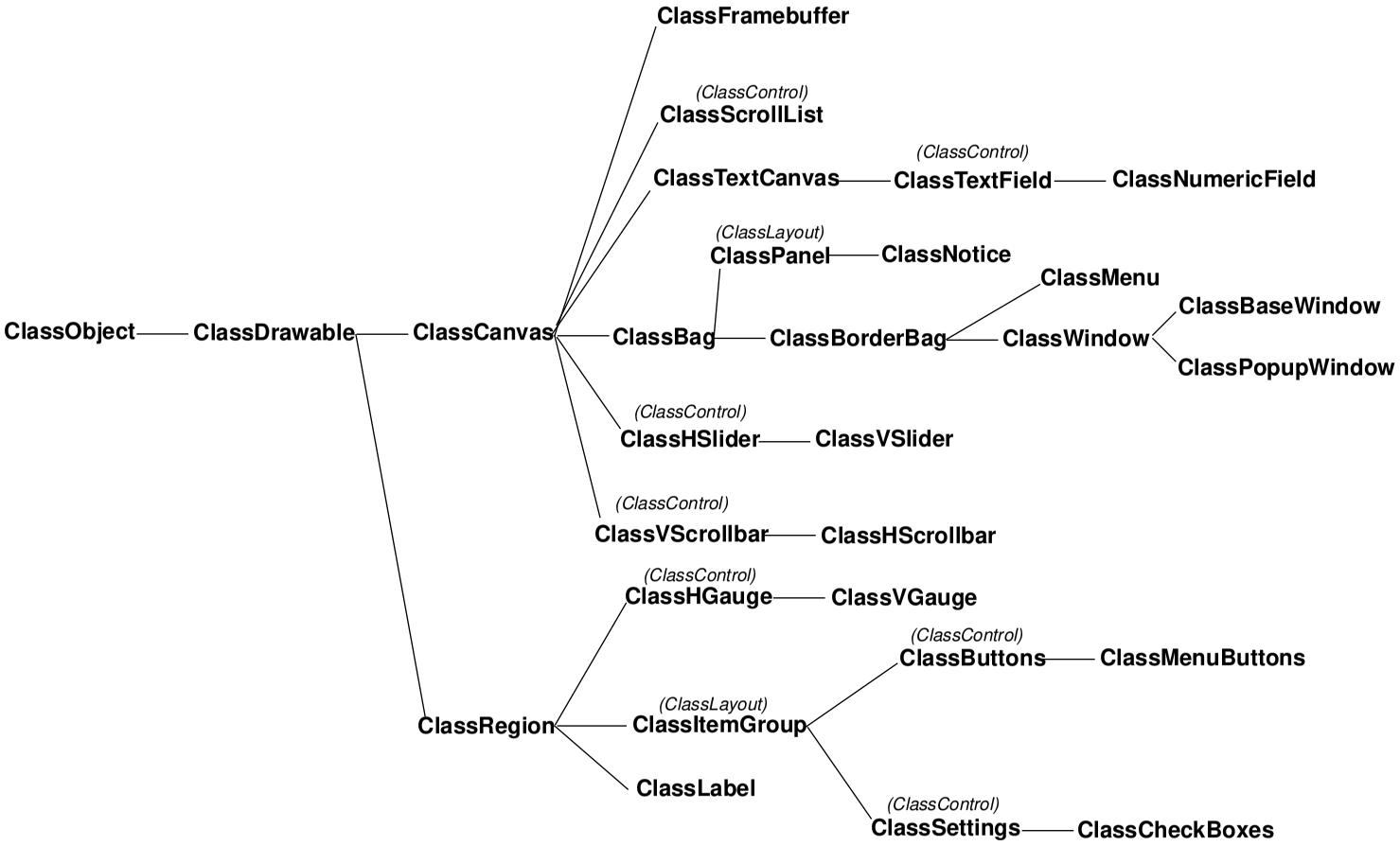
IEEE tutorials on OOP (Peterson) 1987

Journal of Object-Oriented Programing

OOPSLA proceedings

Key: *(classname)* **= mixin class**

TNT Class Hierarchy



Introduction

The NeWS Toolkit (TNT) version 2.0 is an OPEN LOOK user interface toolkit for NeWS. TNT is built on the NeWS Object Oriented PostScript (OOP) system, an interpreted system. C-based libraries are also provided that implement a lightweight notifier and connection manager for NeWS, called the Wire Service, and a text package called Jot.

Apart from the full set of OPEN LOOK components, the Toolkit provides a small set of 'core' classes that provide the substructure for the toolkit's class hierarchy and can be easily customized via subclassing.

Programming flexibility

The best thing about TNT is its flexibility. In a very real sense TNT is a toolkit without "brick walls." You can use the same core classes, the ones that the Toolkit uses to build the OPEN LOOK components, to build your own application-specific user interface components.

Furthermore, it is very easy to modify one of the OPEN LOOK components for your own needs. You simply create your own subclass of a Toolkit object, add new methods or override existing ones, and then instantiate your new class.

Finally, TNT being written in PostScript, may be read by everyone: if you want to significantly change the way the toolkit works the PostScript code is all at your disposal. TNT's classing system is so dynamic that you can even change method definitions while the system is executing. (Changing the TNT source

code should be thought of as an act of last resort however. The cost of doing this will be that you will have to duplicate your changes, or at least reexamine them, with each subsequent release of TNT.)

The cost of all this flexibility is that TNT programmers have to make more decisions than the users of simpler toolkits. In particular, you have to consider the problem of splitting your application between two very different programming environments: the object oriented PostScript world of the NeWS server and the more conventional C or C++ client-side. Use of DevGuide to replace much hand-coded PostScript improves this situation somewhat, but does not eliminate it.

Thin wire

TNT programs perform well over low bandwidth client-server connections such as telephone lines or overloaded networks because the OPEN LOOK components live in the window server and interact with the user without involving the client program at all.

Application programmers can take advantage of the programmable server in this way as well. For example, you can download user-interaction code that animates some operation.

Imaging PostScript

The full PostScript imaging model is at your disposal when writing an application under TNT. Not only can you draw using the hardware independent, scalable and transformable NeWS primitives, but you can also perform input hit detection over the shapes that you draw—no matter how complex they are.

The Classing system

The *NeWS2.1 Programmer's Guide* gives a thorough description of the NeWS Object Oriented PostScript classing system that is used by TNT and other NeWS toolkits.

Programmers already familiar with PostScript and the general principles of object-oriented programming can understand the system by seeing the transparent way that it is layered on top of PostScript in Table 1-1.

Table 1-1 Object-oriented features and their PostScript implementations

O-O feature	PostScript Implementation
Class	dictionary
Instance	dictionary
Class Method	(key, executable) entry in a class dictionary
Class Variable	(key, any) entry in a class dictionary
Instance Method	(key, executable) entry in an instance dictionary
Instance Variable	(key, any) entry in an instance dictionary
“send” operator	Place all superclass dictionaries on the dictionary stack followed by the class and instance dictionaries, then execute the name of the method.

Where the classes live

TNT2.0 classes are automatically loaded into the shared systemdict the first time they are referenced by an application program. Application-specific classes can be stored in any dictionary you choose. By convention they are usually put in the userdict associated with your connection to the server.

Userdict is the current dictionary when a connection is established, so the “def” operator will store application-specific classes or data in that dictionary. Remember however that each connection has its own userdict. If you want some class to be accessible to more than one application (or more than a single instance of an application) you should put it in systemdict.

What TNT isn’t

TNT is not a complete application programming environment. It addresses only the user interface components of the problem, and leaves such issues as application embedding, long-term persistence, and inter-application linkage to higher-level toolkits.

Nor is it a User Interface Management System. Programmers interact directly with the UI components. They are created by sending the /new method to a class, and destroy themselves when the last reference to them is removed.

The architecture of a TNT application

TNT programs come in two pieces: a user-interface component written in PostScript (usually residing in files with the suffix .ps or .cps) and a main application body written in C or C++. Very simple programs can omit the application body if everything is handled in PostScript. For an explanation of how to use the server's client-side facilities (called CPS) see the *NeWS 2.1 Programmer's Guide*, Chapter 6, C Client Interface.

To illustrate this section an example PostScript program is presented. Although all serious applications will have a client-side, many developers first write the user interface part of their application as a standalone PostScript program, and only later integrate a client process. The PostScript interpreter makes incremental development of PostScript-only programs *very* fast.

Server-side Architecture

The general structure of the PostScript part of an application is as follows:

PS-1—definition

Definition consists of the creation of application-specific subclasses of ClassCanvas and other classes. Any core or OPEN LOOK class may be subclassed but most applications that want an input/output surface will contain at least one subclass of ClassCanvas.

PS-2—instantiation

In this step you instantiate the classes you defined in PS-1, one or more windows, some TNT-provided OPEN LOOK control classes, and perhaps some classes to manage the layout of these objects. When you instantiate an OPEN LOOK control you provide a notification procedure to determine what should happen when the user manipulates this control. This procedure is written in PostScript, but can (via the NeWS tagprint and typedprint operators) send a message across the connection to the main body of your application.

PS-3—composition

Composition is the “stitching together” of the objects instantiated in PS-2. Objects can be inserted into the window directly, but more often they are inserted into a canvas that is specialized to manage layout (these are called “bags” in TNT), and this canvas is inserted into the window. This is a recursive process: applications with complex layouts can have bags within bags within bags...

PS-4—start up

Start up consists of the placement (positioning the window on the screen and choosing an appropriate starting size), activation (making the window responsive to input), and mapping (making the window appear) of the window or windows.

Changing the order of these steps is permissible, with the obvious restrictions that a class must be defined before it can be instantiated, and an object must be instantiated before it can be inserted into the window.

The example program that illustrates this structure is a simple color chooser. The program creates a single window which contains a control area holding three sliders, and a color display canvas. The sliders control the hue, saturation and brightness of the color shown in the color display canvas. As the user drags one of the sliders the color shown changes continuously. When the slider is released a message is printed to the footer giving the HSB and RGB values of the final chosen color.

An executable version of this program called manual.colors can be found in the “demo/bin” subdirectory of the TNT2.0 release. It has been annotated with numbers corresponding to the steps PS-1–PS-4 defined above.

```
#!/bin/sh
psh << 'EOF'

%---------------------------------PS-1----------------------------------------
% Define a class whose purpose will be to show something in the chosen color.
%
/ClassColorDisplay ClassCanvas
dictbegin
    /Hue        .8 def
    /Saturation .8 def
```

```
    /Brightness .8 def
dictend
classbegin
    /TextFont    /ZapfDingbats findfont 100 scalefont def

    /PreviewColor { % value control => -
        /name exch send exch 100 div def
        self setcanvas /ShowColor self send
     } def

    /ChooseColor { % value control => -
        /PreviewColor self send
        currentcolor colorhsb (HSB: % % %) [5 2 roll] sprintf
        currentcolor colorrgb (RGB: % % %) [5 2 roll] sprintf
        /setfooter Parent send
     } def

    /ShowColor { % - => -
        0 /size self send exch pop 70 sub moveto
        Hue Saturation Brightness hsbcolor setcolor
        TextFont setfont (88) show                %show DingBat character
     } def

     /Paint {BG fillcanvas /ShowColor self send} def
     /minsize {140 60} def
classend def

%---------------------------------PS-2-----------------------------------------
% Create the window, the colordisplay canvas and a panel to
% put the sliders in.
%
/Win null framebuffer /new ClassBaseWindow send def
/ColorCan framebuffer /new ClassColorDisplay send def
/Panel /Calculated framebuffer /new ClassPanel send def

% Create the three sliders
%
[/Hue /Saturation /Brightness] {
    framebuffer /new ClassHSlider send
```

```
    ColorCan /settarget 2 index send
    /ChooseColor /setnotifier 2 index send
    /PreviewColor /setpreviewer 2 index send
    80 /setvalue 2 index send
    2 copy /setname exch send
    def
} forall

% Create a label for each of the sliders
%
/HueLab (Hue:) framebuffer /new ClassLabel send def
/SatLab (Saturation:) framebuffer /new ClassLabel send def
/BriLab (Brightness:) framebuffer /new ClassLabel send def

%----------------------------------PS-3-----------------------------------------
% Add the sliders and labels to the panel, using the /Calculated
% positioning protocol, then add the panel and our canvas to the window
%
/Hue Hue [/NorthEast {/NorthEast PARENT POSITION}] /addclient Panel send
/Sat Saturation [/North {/South /Hue POSITION 10 sub}] /addclient Panel send
/Bright Brightness [/North {/South /Sat POSITION 10 sub}] /addclient Panel send

/SatLab SatLab [/East {/West /Sat POSITION 5 0 xysub}] /addclient Panel send
/HueLab HueLab [/East {/West /Hue POSITION 5 0 xysub}] /addclient Panel send
/BriLab BriLab [/East {/West /Bright POSITION 5 0 xysub}] /addclient Panel send

/Center ColorCan /addclient Win send
/West Panel /addclient Win send

%----------------------------------PS-4-----------------------------------------
% Finally, start the application up, and detach the psh connection.
%
(Reference Manual Demo #1) /setlabel Win send
10 0 15 20 /setgaps Win send
/place Win send
/new ClassEventMgr send /activate Win send
/map Win send

newprocessgroup
currentfile closefile
EOF
```

Although this program was advertised as “PostScript-only” there is indeed a client-process involved. It is the utility program “psh” which reads PostScript from standard input, sends it to the server, and prints on standard output anything that the server sends back up the connection.

When using psh you have the option of having it exit when it finishes sending your PostScript to the server, or having it stay alive until your PostScript application decides to quit.The advantage of disconnecting the psh and letting it exit after sending all your PostScript is simply that you will have one less UNIX process active, and some space in the NeWS server will be freed up when the connection is closed. The advantages of having psh maintain the connection are that your PostScript can write to standard output and that a ^C delivered to the psh will kill your entire application.

Client-side architecture

While no example code is given the general structure of the main application body, written in C or C++, is as follows:

C-1—#includes

Include the .h files for the Wire Service and any libraries (such as Jot) that you are using. You will usually also include one or more .h file containing encoded versions of the entire PostScript part of your application PS-1 to PS-4, if these were written using CPS.

C-2—callbacks

Define a number of callback functions that will receive the control notification messages described in PS-2.

C-3—allocation

Allocate a number of tags and tokens to be used in communicating with the PostScript part of your application. Tags are used to associate a PostScript control with its C callback function; tokens are handles for the C side to refer to PostScript objects.

C-4—registration

Associate your tags with your callback functions.

C-5—server initialization

Send some PostScript to the NeWS server. You do this by making CPS calls that transmit code included in the .h files.

C-6—notification

Enter the Wire Service notifier, and wait for it to call the functions you registered in C-2.

The ability to structure applications in this way is one of the unique aspects of the NeWS environment. The next section focuses on how you can take advantage of this structure.

The client-server split

In deciding how to split your application between the client process and the NeWS server the following considerations should apply.

- Low latency response—server
  If you need the system to respond in milliseconds to some particular user action then you should probably write the code to implement that response in PostScript and download it to the server. This is what was done with the slider previewing in the manual.colors example introduced in *The architecture of a TNT application*.

- Hit detection over PostScript shapes—server
  If you need to work out whether a mouse click or drag occurred inside any shape more complex than a rectangle you should probably leave that task to NeWS. This can be done either with the NeWS operator **pointinpath**, or by making a canvas whose boundary is the path in question, and requesting mouse input over that canvas.

- Broadcast-style communication with other programs—server
  The NeWS event mechanism is powerful and easy to use. If you have a group of applications that want to communicate by broadcasting information without regard to who is listening, then NeWS events are probably the easiest and cheapest way to achieve this.

- Computation intensive—client
  Your PostScript code is interpreted, and your C code is compiled. Sorting an array of strings for example, is going to execute *much* faster in the client than in the server.

- Data intensive—client
  PostScript data structures are large reference-counted objects. While the addition of a new integer field in a C structure may only cost you 4 bytes for example, the addition of an instance variable with an integer value to your PostScript code will probably grow the server by 20-30 bytes.

- Access to system calls and UNIX libraries—client
  NeWS provides access to hardly any of the UNIX system calls and very few of its standard libraries. Application code that calls UNIX utilities must be placed in the client process.

- File reading—client
  Even though NeWS offers the "file" primitive to open a UNIX file, its use should be carefully considered. The problem is caused by the flexibility of NFS which allows different machines to have different names for the same file (and the same name for different files!). In the situation where your client process is running on a different machine from the window system it is generally agreed that the file name resolution should be done on the client machine.

Finally, when in doubt you should err on the side of putting less into the server rather than more. This is because interpreted PostScript code is less space efficient than compiled C code.

Memory management in TNT

NeWS has a reference-counted garbage collection system. The creation and destruction of TNT objects follow the same rules that govern the persistence of other NeWS data structures. The X11/NeWS Server Guide describes the system in general, but the implications on TNT programming warrant some further explanation.

Creation and destruction of references

The general way that an instance of a TNT class is created is:

```
/myinstance
    <arguments> /new ClassSomething send
def
```

You now are holding one reference to the instance in your current dictionary (usually userdict), and the key associated with this reference is "/myinstance". To destroy this object all that you need do is remove this reference. There are two ways to do this in TNT:

```
currentdict /myinstance undef
```

or

```
/myinstance null def
```

Automatic cleanup of references

There is of course another way to remove this reference—you can remove your last reference to the dictionary into which you defined "myinstance". That dictionary will then be garbage collected, and everything in it will have their reference counts decremented. Those objects whose reference counts reach zero (such as the instance you created) will then be garbage collected.

This sounds complicated, but as long as the dictionary in question is userdict, this wholesale destruction will happen automatically when your application quits. So, unless you want to get rid of an instance or an application-specific class while your application is still running, you do not need to worry about explicitly removing the references to the objects you create in userdict.

If you define anything into systemdict it will *not* automatically be removed when your application quits. This is a good reason to avoid storing anything in systemdict.

References maintained by the Toolkit

Your application is not the only place where references to your objects are held, and a general understanding of how the toolkit maintains references is useful for TNT programming.

The most important references that TNT holds are references to the instances of ClassCanvas and ClassRegion that form the hierarchy of drawable objects in your application. This is done because the Toolkit needs to be able to traverse this hierarchy when a command to repaint or reshape its root canvas (usually

an instance of ClassWindow) is given. In the example program presented in The architecture of a TNT application, the canvas hierarchy held the following references (the references are the by-product of calls to the **/addclient** method in PS-3):

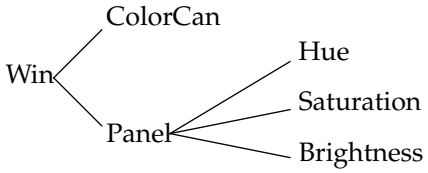


Figure 1-1 Structure of references in manual.colors

Because the objects ColorCan, Panel, Hue, Saturation, and Brightness are all being referred to by other objects in the canvas/region hierarchy there was no particular need for references to them to be left in userdict. (In section PS-2 of manual.colors this was done for programming convenience and readability.) The only object that the programmer *must* maintain a reference to is the root of the hierarchy, in this case "Win".

If the reference to Win is dropped, then the entire tree will attempt to destroy itself. As soon as all other references to objects in this tree have been removed the destruction will be complete. In the example in The architecture of a TNT application, quitting the application or interrupting the psh causes the userdict to be destroyed so the /Win reference is dropped, but so are all the references to the internal objects, and hence the cleanup proceeds immediately to completion.

Cross-Object references within your application

You can also have instance variables that hold a reference to another object in your application. You may wish to do this if you want to send a message from one object to another. Remember when you do this though, that the object being referred to will not go away until this reference is dropped. (The targeting mechanism in ClassControl, and the more general ObsoleteService allow you to defeat this behavior. Targets make use of "soft" references which are explained in the *X11/NeWS Server Guide*.)

Finally it is possible in your program to introduce reference cycles (such as A refers to B and B refers to A, or even A refers to itself!) that will defeat the garbage collection mechanism. If you need to create such cycles, then at least one of the references should be soft, and the object pointed to by the soft reference should subclass its **/destroy** method:

```
/destroy { % - -> -
    % remove your reference to the next object in the cycle
    /Something null def   /destroy super send
} def
```

The **/destroy** method will be called automatically when the only remaining references are soft ones. You can override **/destroy** to break a reference cycle and to send **/destroy** to other objects you've created.

Process management in TNT

NeWS provides the "fork" primitive by which application programmers can create lightweight processes as they see fit. Generalizing from this, TNT provides a class called ClassEventMgr. Instances of ClassEventMgr are processes that are customized specifically for receiving and processing NeWS events.

Application programs can fork as many processes as they like, and can have any number of event managers running simultaneously. Most TNT programs however will not benefit from a profusion of processes and standard programming style recommends the use of exactly three processes, only one of which is created explicitly by the application. These three processes are described below.

The connection reader

This process is created automatically by NeWS when a connection is successfully made to the server. Its only purpose is to read PostScript from a client program and execute it. It is this process that executes the PostScript code shown in The architecture of a TNT application to create a class, instantiate some objects, and start the application. This process will die automatically if the connection is closed by the client program.

The local event manager

The local event manager (LEM) is a process you create (see the "/new ClassEventMgr send" at the end of the example in *The architecture of a TNT application*), and it is responsible for receiving most of the events that are handled by the toolkit on behalf of your application.

All processing resulting from the notification or previewing of a control takes place in the local event manager. In particular, the calls to /PreviewColor and /ChooseColor in manual.colors that are caused by the user dragging and releasing a slider, will be executed by this process.

This process starts with the same execution environment as your connection process. (This is not surprising since it was forked from the connection process, and in NeWS, as in UNIX, a forked process inherits its parent's environment.) The importance of this is that it shares the same userdict and standard output file as the connection reader. Thus, any objects defined into userdict will be accessible to this process and can be referred to in the code that handles the notification or preview of a control. Similarly, if a tagprint or typedprint operator is invoked in this process the value will be written to the correct file and your client side will receive it.

Every canvas within your application keeps a reference to its LEM, holding it as the instance variable **/EventMgr**. Activating a canvas or a tree of canvases and regions (see the call to **/activate** in the manual.colors example) will cause interests to be expressed on behalf of your application within this process and within the global event manager (described below).

If you want your application to receive events that the toolkit has not anticipated (i.e., events other than mouse events, keyboard events, and damage events) you can express extra interests to your LEM using ClassInterest. For a little more information on this topic see the footnote on page 46.

The global event manager

The global event manager (GEM) is a process created when TNT is first initialized, and is used to receive certain types of events on behalf of *all TNT applications*. Its purpose is to synchronize certain user actions and avoid race conditions. The GEM also saves a great deal of space by combining similar processing across multiple canvases.

In most cases, the GEM merely determines which LEM should handle a particular event, and redirects the event to that LEM. In a few cases (inter-application communication, where more than one LEM is involved, or application destruction, where the LEM may be dead) certain client-supplied methods may be executed in the GEM or in another application's LEM. These cases are noted where they arise.

Such methods cannot assume that they have the correct userdict or connection in place when they execute. If your code requires access to the connection or userdict then you should use the **/sendmanager** or **/callmanager** utilities to temporarily create the correct environment. For an explanation of **/sendmanager** and **/callmanager** see Chapter 5, ClassCanvas, Changing event managers for execution and sends.

Starting and quitting applications

Sending **/activate** to a tree of canvases makes them ready to receive input, but system generated events (such as mouse events or keystrokes) will not be delivered to your application until it is mapped onto the screen. In general, the first event that is delivered to your application is the damage event. It is received by the window and causes the window to paint itself and its contents for the first time.

The client side of your application will not start to respond to messages sent up the connection until you call `wire_EnterNotifier`() or its single-shot variant, `wire_Notify`(). (See Chapter 26, The Wire Service for an explanation of these functions.)

If you call `wire_EnterNotifier`(), then a corresponding call to `wire_ExitNotifier`() from within one of your callbacks will indirectly cause `wire_EnterNotifier` to return. Simple applications will take this as an indication to exit, and do so.

Exiting from the client process at any time is a safe way to kill both parts of your application. The effect on the server when a client program exits is to close the connection and kill the connection reader and all processes in its process group. This includes the LEM and any other process you may have forked from the connection reader or the LEM.

Killing the connection process will in general remove the last reference to your userdict and all objects referenced directly or indirectly by it. Your application will disappear from the screen unless some part of it is referenced by

something that was not destroyed. (If your application does not disappear you may need to use the "reffinder" utility described in the X11/NeWS Server Guide to track down the rogue reference.)

When the user chooses "Quit" from a base window menu the message **/QuitFromUser** is sent to that window. The default implementation of **/QuitFromUser** provided by TNT will kill your program. If you need to inform the client side that it is about to be killed, or you want to confirm this with the user, then override **/QuitFromUser** and insert your code there.

Painting in TNT

There are two ways to create a drawing in your application under TNT. The general way is to override the **/Paint** method of one of the objects in your canvas/region hierarchy. The full power of PostScript is at your disposal.

If your drawing is as simple as a string, a bitmap, or a glyph then you can take advantage of Display Items (see the Display Items chapter). Display Items are not objects, just simple PostScript data structures that the toolkit accepts as an argument wherever you would expect to supply a string for rendering. Subclassers of ClassCanvas and ClassRegion have access to the display item handling code, and can use them in their subclasses if they choose.

The Canvas/Region hierarchy

Window systems and user interface toolkits generally provide a way to hierarchically combine objects on the screen. At the window system level this is usually in the form of nested areas on the screen that have their own coordinate spaces, receive their own input events, and clip their own painting. Viewed from the toolkit level this hierarchy allows you to build a powerful user interface to your application by gluing together simple independent components.

The objects of aggregation in NeWS are canvases. In TNT they are instances inheriting either from ClassCanvas or ClassRegion.

ClassBag

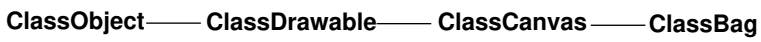


Figure 2-1 The ClassBag subtree

A bag is a canvas subclass explicitly designed to manage a collection of canvas and region clients. A borderbag is an example of a bag that is designed to manage one major client and up to four minor clients. (See *ClassBorderBag* on page 23.)

Moreover, each client of a bag must respond to the following methods:

/map /destroy /mapped?

/reparent /activate (canvas client only)

/paint /deactivate (canvas client only)

ClassBag is the most general type of container; providing some control over an arbitrary number of canvases and regions. ClassBag manages canvas activation (i.e., activating the bag also activates its canvas clients) as well as providing a convenient place to enforce layout policies on clients.

Creation

/new

parentcanvas /**new** instance

Returns an instance of ClassBag.

Activation and deactivation

A bag can receive user input if it is active and ceases to react to user input when it is deactivated. See *Activation and deactivation* on page 46 and *ClassEventMgr* on page 48 both in ClassCanvas.

/activate

event-manager /**activate** -

Turns on event management for the bag and all its canvas clients. event-manager is an instance of ClassEventMgr. Checks to see if the bag already has an event manager; if it does, the given event manager instance is just popped off the stack and /**activate** does nothing.

/deactivate

- /**deactivate** -

Turns off event management for the bag and its canvas clients. Any of the bag's canvas clients that have their own event manager stay active.

If the bag has no event manager then /**deactivate** does nothing.

Bag clients

Clients are added to bags using /**addclient** and removed from them using /**removeclient**. If you want to move a client from one bag to another you must first remove it from the bag it's in and then add it to the new bag.

/addclient

name client /**addclient** -

Adds client to the bag and associates it with name. name is any valid dictionary key, i.e., any non-null value. Each client's name *must be* unique. client is either a canvas or a region. The client is reparented to the bag. If the bag has been activated the canvas is activated also.

If you do not need handles for your bag clients you can use "dup" as the client's unique name. I.e., you could do:

```
<create a canvas client, e.g., myclientcanvas>
dup /addclient mybag send
```

Not only does this ensure that each client has a unique handle, it also lets you use the client itself as the argument to /**removeclient**:

```
myclientcanvas /removeclient mybag send
```

Finally **/addclient** invalidates the bag so its layout method is called the next time repainting or layout is necessary. For an explanation of the The NeWS Toolkit's validation scheme see *Validation* on page 44 in ClassCanvas.

/client

name **/client** client true | false

Searches the bag's client list to determine if name is known. If it is known, **/client** returns true and the client instance that was associated with name during the call to **/addclient**. If name is not known, **/client** returns false.

/clientcount

- **/clientcount** number-of-clients

Returns the number of clients managed by the bag. number-of-clients includes both canvas and region clients.

/clientlist

- **/clientlist** [client1 client2 ...]

Returns an array that contains all the clients (both canvases and regions) in the bag.

/regionclientcount

- **/regionclientcount** number-of-region-clients

Returns the number of region clients in the bag. If there are no regions in the bag **/regionclientcount** returns 0.

/regionclientlist

- **/regionclientlist** [region-client1 region-client2 . . .]

Returns an array that contains all the region clients in the bag.

/removeclient

name **/removeclient** oldclient true | false

Removes the client associated with name from the bag and invalidates the bag (if name is found). Returns the client instance and true if name exists; otherwise, false is returned.

If the removed client is a canvas and it has the *same* event manager as the bag, **/removeclient** deactivates it, which means that if a client is activated *before* it is added to a bag, it is *not* deactivated automatically when it is removed.

Destruction

/destroy

- **/destroy** -

Destroys the bag and its clients. **/destroy** executes in the global event manager.

See also *Callback context* on page 48 and **/destroy** on page 74 in ClassCanvas.

Miscellaneous

/validate

- /validate -

Updates the appearance of the bag to reflect any changes (reshaping, new clients added, etc.) that may have taken place. You should not call this yourself because painting causes validation. In those cases where you want immediate validation, call **/?validate**. (See page 45 in ClassCanvas for an explanation of validation.) **/validate** calls **/layout**.

/layout

-/layout -

Initiates layout for a bag. Establishes the layout context by making the bag the current canvas and then calls **/Layout**.

Subclasser methods

/FixChildren

- /FixChildren -

Handles the repainting of damaged bag clients by sending **/FixAll** to all the bag's clients. See *Canvas damage handling* on page 51 in ClassCanvas for an explanation of how canvases handle damage repair. See the *NeWS 2.1 Programmer's Guide* for an explanation of damage.

/Layout

- /Layout -

Lays out the clients of the bag. Subclasses override **/Layout** to implement a particular layout policy. If you are going to subclass ClassBag, you must provide a definition of **/Layout**.

/PaintChildren

- /PaintChildren -

Paints the clients of the bag by sending **/PaintAll** to all client instances. First, regions are painted in insertion order, then canvases are painted in insertion order.

Mouse tracking for regions in bags

The following mouse tracking methods are included in ClassBag to handle mouse tracking for region clients. For an explanation of mouse tracking see *Mouse tracking* on page 55 in ClassCanvas. To see how regions handle tracking see *Mouse tracking* on page 147 in ClassRegion.

/TrackMotion event /**TrackMotion** -

/TrackStart event /**TrackStart** false | [/label...] true | /name true

/TrackStop event /**TrackStop** -

/TrackCancel event /**TrackCancel** -

Menus

The following menu methods are included in ClassBag to handle menus for region clients. For an explanation of canvas menus see *Canvas menus* on page 53 in ClassCanvas. For an explanation of how regions handle menus see *Region menus* on page 147 in ClassRegion.

/MenuStart invoker posname event /**MenuStart** invoker posname event menu true
| invoker posname event false
| invoker posname event null true

/MenuStop menu /**MenuStop** -

ClassBorderBag

Figure 3-1 The ClassBorderBag subtree

ClassBorderBag, a subclass of ClassBag, is designed to manage up to five clients; one located in the center and the other four located, logically enough, on the four borders. Figure 3-2 illustrates one possible configuration of a border bag and its five clients. Also note the gaps. You use gaps to set the amount of space between border clients and the central client.

Border bag's clients

Consistent with the bag interface, ClassBorderBag's clients are referenced by name.

- A northern client that matches the width of the central client is referenced as /North. A northern client that matches the combined widths of the central, eastern, and western clients is referenced as /-North-. As the bag width changes, the north client's width also changes; however, its height remains constant.

- An eastern client that matches the height of the central client is referenced as /East. An eastern client that matches the combined heights of the central, northern, and southern clients is referenced as /-East-. As the bag height changes, the east client's height also changes; however, its width remains constant.
- A southern client that matches the width of the central client is referenced as /South. A southern client that matches the combined widths of the central, eastern, and western clients is referenced as /-South-. As the bag width changes, the south client's width also changes; however, its height remains constant.
- A western client that matches the height of the central client is referenced as /West. A western client that matches the combined heights of the central, northern, and southern clients is referenced as /-West-. As the bag height changes, the west client height also changes; however, its width remains constant.
- The central client is referenced as /Center. It inhabits the central space in the bag. Once the space requirements of the border clients are satisfied, the central client occupies the remaining space.

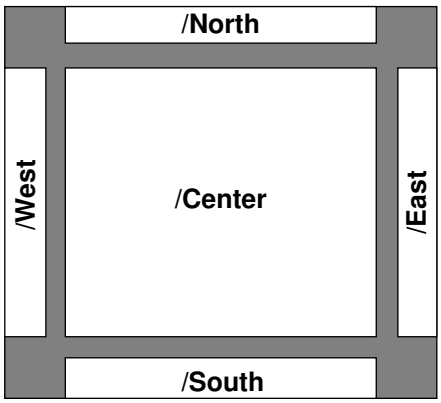


Figure 3-2 Four short border clients surrounding a central client and separated from it by symmetric gaps

A border bag can also have insets, which determine the space between the border clients and the edge of the bag (Figure 3-5).

As illustrated in Figure 3-3 you can use border bags to arrange clients the way a typical application might.

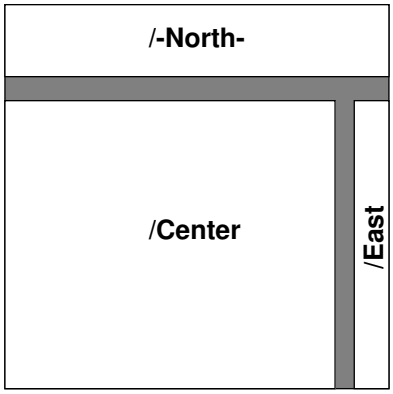
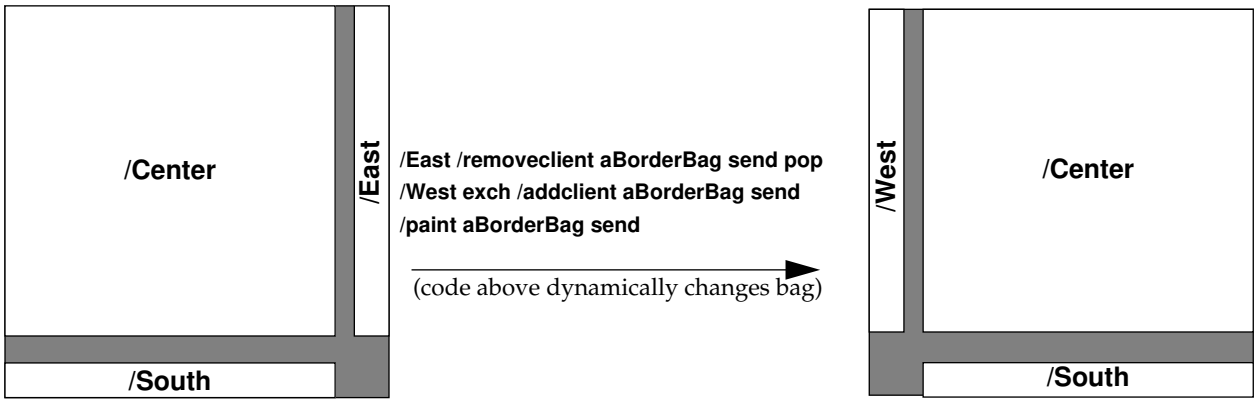


Figure 3-3 A typical OPEN LOOK arrangement: the /-North- client might be a control area, the /East client a scrollbar, and the /Center client a canvas.

You can dynamically rearrange the clients of a border bag. One way this dynamic rearrangement would be useful is to allow a user to switch a scrollbar from one side of the bag to the other. For example if you have a border bag with /Center, /East, and /South clients you could dynamically rearrange the border clients to be /Center, /West, and /South clients (Figure 3-4).



A canvas (/Center) with vertical (/East) and horizontal (/South) scrollbars.

A canvas (/Center) with vertical (/West) and horizontal (/South) scrollbars.

Figure 3-4 Dynamically changing the arrangement of border bag clients

Adding clients

/addclient

name client /**addclient** -

Adds client as a client of the border bag. In order for automatic layout of your clients to operate, name must be one of /North, /East, /South, /West, /Center, /-North-, /-East-, /-South- or /-West-. When the client has not been reshaped /**addclient** reshapes it to its preferred size. Some combinations of clients are not allowed, e.g., a /-North- and an /-East- clients. If you do add both of these clients the /-East- client takes precedence and "overlays" the extension of the /-North- client into the northeast corner.

Positioning clients

You can set gaps and insets in border bags. Gaps determine the distance between the /Center client and the bag's border clients. Insets determine the distance between the border clients and the edge of the bag.

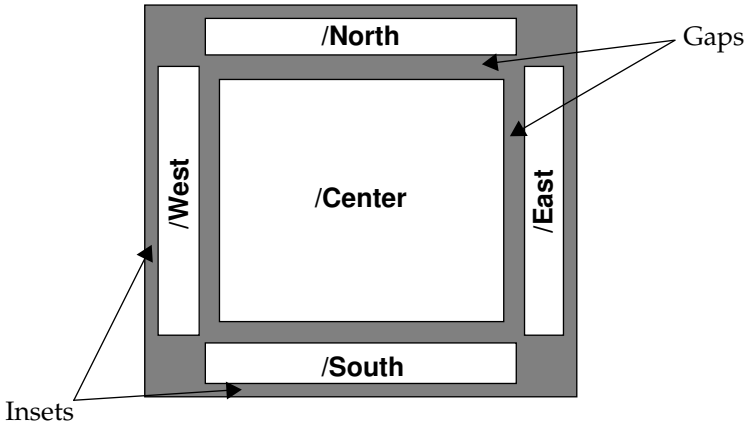


Figure 3-5 Using gaps and insets to position four border clients.

/setgaps

northgap eastgap southgap westgap /**setgaps** -

Sets the distance between border clients and the central client. The default unit is the point. Gaps specify how far each of the border clients are spaced from the center client.

/gaps

-/**gaps** northgap eastgap southgap westgap

Returns the value of the border bag's gaps.

/setinsets — northinset eastinset southinset westinset /**setinsets** -

Determines how far the /North, /East, /South and /West clients are inset from the edge of the bag.

/insets — - /**insets** northinset eastinset southinset westinset

Returns the border bag's insets.

Geometry

/minsize — - /**minsize** width height

Returns a border bag's minimum width and height. The calculation of a border bag's minsize includes the minsizes of its clients, which means there are several factors which you should consider when you set a borderbag's minsize (or preferredsize):

- Preferredsize defaults to an object's minsize if no preferredsize is set.
- Clients of borderbags are reshaped to their preferredsize when they are added to borderbags, if they haven't already been reshaped.
- North and south borderbag clients have constant heights. Thus, the height you assign to north and south clients in their preferredsizes becomes the constant width these clients maintain in the borderbag.
- East and west borderbag clients have constant widths. Thus, the width you assign to east and west clients in their preferredsizes becomes the constant width that these clients maintain in the borderbag.

Thus, if you set both a minsize and a preferred size for non-/Center clients of a borderbag **/minsize** should return the same value for the constant aspect of a client's size as **/preferredsize**. E.g., an east client has a minsize of 20x20 and a preferredsize of 20x100: both /preferredsize and /minsize for this east client must return the same width but are allowed to return different heights.

/preferredsize — - /**preferredsize** width height

Returns the "ideal" size of the border bag. Defaults to the values returned by **/minsize**.

Layout

/Layout

-/Layout -

Lays out the border bag and its clients. Overridden from ClassBag to properly lay out the border clients. East and west clients never have their width adjusted by **/Layout**, only their height is modified to fill the available space. Similarly, north and south clients maintain a constant height, but their widths vary as the bag expands and contracts. See *Border bag's clients* on page 23.

Menus and tracking

By default border bags are Menuable and Trackable. See *Canvas menus* on page 53 and *Mouse tracking* on page 55 in ClassCanvas for information on menus and tracking, respectively.

Buttons

Figure 4-1 The buttons subtree

Buttons and menu buttons come in groups; buttons are considered items in the button group. A single button is a degenerate case. The buttons in a group are items and are added, deleted and accessed using the methods found in ClassItemGroup (see ClassItemGroup on page 87). Some of the ClassItemGroup methods inherited by ClassButtons are:

/setitemlist	/itemlist	/itemcount
/insertitem	/replaceitem	/deleteitem
/appenditem	/itemsize	/itemlocation
/itembbox	/pointinitem?	/pointtoitem

In addition, ClassControl is mixed-in to ClassButtons. The control interfaces documented in ClassButtons are: /setnotifier, /notifier, /setvisualstate /visualstate and the target interface (see *Managing references between controls and other objects*).

A button group is created and items are added either one at a time or with a single call. Most of the time items are simple string display items, but theoretically any display item may be displayed in a button. (See Chapter 7, Display Items on page 79.) However, buttons are restricted to a fixed size that accommodates a maximum font size of 12 points.

ClassButtons

Creation

/new — placement parent /**new** instance

Creates a button group, i.e., an instance of ClassButtons. placement is one of /Spaced, /Absolute, /Calculated or /Grid. (See Chapter 9, ClassItemGroup; Setting the item list, for an explanation of placement).

Abbreviated buttons

/setabbreviated — item-index boolean /**setabbreviated** -

Determines whether the button at item-index is abbreviated. Invalidates the group. (See the *OPEN LOOK Graphical User Interface Functional Specification* for a definition of abbreviated buttons.)

/abbreviated? — item-index /**abbreviated?** boolean

Returns whether the button at item-index is abbreviated.

Ability to interact with buttons

/setvisualstate — item-index state /**setvisualstate** -

Sets the visual state of the button at item-index to be state. The visual state of a button determines not only its visual presentation but also whether the button can accept user interactions. state is one of /Active, /Busy, or /Inactive. Users can interact with /Active buttons; /Inactive and /Busy buttons ignore user interaction. The button repaints to reflect the new state only if it is valid.

/visualstate — item-index /**visualstate** state

Returns the state of the button at item-index.

Default

/setdefault — item-index /**setdefault** -

Makes the button at item-index be the default choice for the button group. Invalidates the group. The default button has a ring drawn around it. Button defaults are used most often (if not exclusively) in notices.

/default — - /**default** item-index | null

Returns the item-index of the default button choice if there is a default. If no default exists null is returned.

Justification

/setjustification — /Left | /Centered /**setjustification** -

Sets the justification for the button's display item. Invalidates the group. The default is /Centered.

/justification — - /**justification** /Left | /Centered

Returns the justification of the button group.

Notification

Notifiers execute in the context of the local event manager (LEM) and are called with the item-index and button group on the stack. You can associate a notifier with the entire button group (/**setnotifier**) or on a per button basis (/**setitemnotifier**).

/setnotifier — notifier /**setnotifier** -

Sets the notifier for the entire button group. If you don't use /**setitemnotifier** then all the buttons in the group have the same notifier.

/notifier — - /**notifier** notifier | null

Returns the button group's notifier. If the button group doesn't have a single notifier that is associated with all the buttons null is returned.

/setitemnotifier — item-index notifier /**setitemnotifier** -

Sets the notifier for the button at item-index. The per item notifier takes precedence over any button group-wide notifier that may exist.

/itemnotifier — item-index /**itemnotifier** notifier | null

Returns the notifier of the button at item-index. If the button doesn't have a notifier then null is returned.

Managing references between controls and other objects

This section contains the target interface that ClassButtons inherits from ClassControl.

/settarget — object /**settarget** -

Sets object as the target of the button's notifier. If a previous targets exists it is overwritten.

/cleartarget — null | object /**cleartarget** -

Clears the target. If null is given the target is cleared. If object is specified then the target is cleared only if object and the target are the same. This latter specification ensures that the target is not incorrectly cleared.

/sendtarget — arguments /method /**sendtarget** results

Sends /method and any arguments that the method requires to the target.

/target — - /**target** null | object

Returns the target.

/HandleObsoleteTarget — object /**HandleObsoleteTarget** -

Use for breaking reference chains. The default is to send /**cleartarget** to object. Subclasses of ClassControl that override /**HandleObsoleteTarget** should always do a super send.

ClassMenuButtons

ClassMenuButtons implements OPEN LOOK menu buttons.

Items in a menu button group are described using the following specification:

[item menu]

item serves as the text label or image that is presented within the borders of the menu button. menu is either a menu instance or a PostScript code fragment that returns a menu instance when executed; i.e., when MENU is pressed over the button the code fragment is executed. The menu associated with the menu button is positioned based on the menu direction state.

For example, you could set a menu button group's itemlist like:

```
[    [ (Button1) menu1 ]
     [ (Button2) menu2 ]
     [ [(RedButton) 1 0 0 rgbcolor ] redmenu ]
     . . . ] /setitemlist mymenubuttons send
```

ClassMenuButtons inherits the methods specified in ClassButtons above. In addition, menus in buttons use a mechanism that maintains a notion of which object is affected by the menu's notifier. For information on this mechanism see The invoker mechanism on page 117.

Methods related to menus and defined in ClassMenuButtons are:

/setmenudirection — item-index menu-direction /**setmenudirection** -

Specifies where the menu appears when MENU is pressed over the menu button at item-index. menu-direction is defined as either /Down or /Right. The menu mark visual corresponds to menu-direction. The default menu direction is /Down.

/menudirection — item-index /**menudirection** menu-direction

Returns the menu direction for the menu button at item-index.

/setmenu — item-index menu /**setmenu** -

Sets the menu for the menu button at item-index. menu is specified as either a menu instance or a PostScript code fragment that returns a menu instance when executed.

/menu — item-index /**menu** menu

Returns the menu for the menu button at item-index.

ClassCanvas

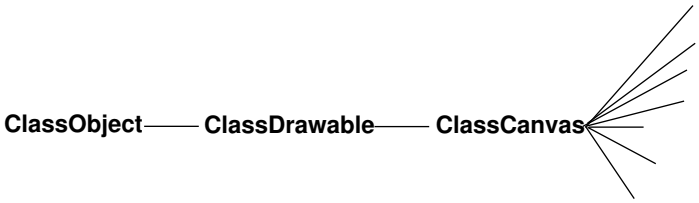


Figure 5-1 The ClassCanvas subtree

ClassCanvas provides the structure underneath most of what you see on the screen (the "look" of the Toolkit), as well as linking an application to input (the "feel" of the Toolkit). A TNT canvas is basically a NeWS canvas with additional structure to support object-oriented programming, display of the canvas, event management, and NeWS canvas tree operations.

Creation

/new

parentcanvas /**new** instance

Subclasser method: /NewInit

Returns a canvas instance, with the specified canvas as its parent. /**new** is sent to the class, which creates an object and then /**NewInit** is sent to that object.

/NewInit

args | - /**NewInit** -

This method is called by /**new** to initialize an instance. It is overridden to allow subclassers to perform any specific initialization on the instance. When it is overridden, the method should do a super send so that the superclass can do its initialization. Also, it should do any specific initialization the class requires and consume the arguments on the stack.

Canvas appearance

Colors

The NeWS Toolkit's OPEN LOOK components are drawn in a "3-D" style using five different brightness values to represent light and dark shading of beveled edges (Table 5-1). (For a complete explanation of how these shadings are used to display 3-D objects see the *OPEN LOOK Graphical User Interface Functional Specifications*.)

However, the 3-D effect is unusable on a black and white display when gray values are approximated by stipple patterns. In order to make the OPEN LOOK components usable in two colors (i.e. black and white), the Toolkit provides 2-D colors and interfaces to switch between 3-D and 2-D looks.

/setcolors

foreground-color background-color /**setcolors** -

Sets the foreground and background colors for the canvas. /**setcolors** examines the /**3D?** variable (see Switching between 3-D and 2-D looks on page 38) to determine whether to set /FG and /BG (3-D colors) or /2DFG and /2DBG (2-D colors). If /**3D?** is true, /**setcolors** also computes and sets BG0, BG2, and BG3, based on the value of background-color. See Table 5-1 for an explanation of these colors. If the instance is valid, it is repainted immediately using the new colors. If the instance is invalid the colors are reset but the instance is not painted automatically. See *Validation* on page 44 for more information on validation.

Send /**setcolors** to a class to change the class defaults or an instance to change just the instance.

/colors

- /**colors** foreground-color background-color

Returns the foreground and background colors for the canvas. /**colors** examines the value of /**3D?** to determine whether to return /FG and /BG or /2DFG and /2DBG. Values returned are the NeWS color objects.

/BackgroundColor - /**BackgroundColor** background-color

Returns BG or 2DBG, according to the dimensionality. /**BackgroundColor** checks to see what the dimensional state of the framebuffer is before returning a color. Thus you should use /**BackgroundColor** instead of /BG so your application will use the appropriate 2-D or 3-D color.

/ForegroundColor - /**ForegroundColor** foreground-color

Returns FG or 2DFG, according to the dimensionality. /**ForegroundColor** checks to see what the dimensional state of the framebuffer is before returning a color. Thus you should use /**ForegroundColor** instead of /FG so your application will use the appropriate 2-D or 3-D color.

Canvas color class variables

TNT uses slightly different names for its 3-D colors than are found in the *OPEN LOOK Graphical User Interface Functional Specifications*. Table 5-1 shows how the TNT color variables map into the OPEN LOOK names as well as the names of the 2-D class variables.

Table 5-1 Toolkit color variables and their OPEN LOOK names

TNT Class Variable	OPEN LOOK name	Explanation
/FG	Foreground	Foreground color. Used for text and the border of the canvas. Default is black.
/BG0	Background	Background color 0. Slightly darker than white, the OPEN LOOK color that it replaces.
/BG	BG1	Background color 1. Slightly darker than bg0. Replaces the OPEN LOOK color, BG1. Used for the background of the canvas.
/BG2	BG2	Background color 2. Slightly darker than bg. Used as the background of “indented” choices.
/BG3	BG3	Background color 3. Used for the shadow of 3-D objects.
/2DFG	Black	Used for the foreground of 2-D canvases. Defaults to black.
/2DBG	White	Used for the foreground of 2-D canvases. Defaults to white.

Switching between 3-D and 2-D looks

/3D? **/3D?** boolean (Variable)

Determines whether the canvas is drawn in 3-D or 2-D (i.e., the canvas's dimensionality). The default is true—use 3-D colors. 3-D canvases are drawn using the 5 colors, FG BG BG0 BG2 and BG3 . 2-D canvases are drawn using the 2 colors, 2DFG and 2DBG. If you the Toolkit to be in 2-D mode when it is first loaded you can set the dimensionality by doing a /3D? false put in UserProfile.

/set3D boolean **/set3D** -

Sets the dimensionality of the canvas. **/set3D** invalidates instances but does not cause the canvas to repaint.

Painting

TNT uses the following rendering model for painting canvases: **/paint** initiates the painting recursion by calling **/PaintAll**. **/PaintAll** sets up the context of the current object *relative to* that of the parent. The context is established using the NeWS operator, **setcanvas**. **/PaintAll** also validates the object. (See Validation on page 44.) **/PaintAll** calls **/Paint** and **/PaintChildren**. **/Paint** paints the object itself. Clients should override **/Paint** (unless the default **/Paint** method is suitable, which is rare). **/PaintChildren** sends **/PaintAll** to each Mapped child.

/paint - **/paint** -

Subclasser method: /PaintAll

Sent to a canvas to initiate painting.

/PaintAll - **/PaintAll** -

Subclasser method: /Paint
/PaintChildren

Paints the entire canvas. Checks to see if the canvas is valid and if it isn't, validates it. **/PaintAll** sets up the context for the painting by making the canvas the current canvas. Uses the **/Paint** and **/PaintChildren** methods for painting. Subclassers should override **/Paint** and/or **/PaintChildren** to change the default painting behavior.

/Paint — **- /Paint -**

Paints the interior of a canvas. You do not need to do a canvas setcanvas or a gsave/grestore in your **/Paint** procedure, they are done for you (by **/paint**) before **/Paint** is called. The default painting done is a fill using the background color. Subclassers should override **/Paint** to perform their own canvas painting.

/PaintChildren — **-/PaintChildren-**

Paints the canvas's children. **/PaintChildren** is a no-op for ClassCanvas and is overridden in ClassBag. If you create a subclass of ClassCanvas that has children and is not a subclass of ClassBag, you should override **/PaintChildren**.

Utility painting methods

The methods in this section are utility methods and are provided because of their usefulness. These are *not* subclasser methods. The methods should be called from within gsaves and grestores and the canvas must be set. If you're using these methods inside your **/Paint** definition you don't have to set the canvas yourself, TNT does this for you. See the rendering model discussion above.

/BotRightPath — x y width height **/BotRightPath -**

Constructs a path of the bottom and right edges of the box defined by x y width and height.

/FillCanvas — color **/FillCanvas -**

Fills the entire canvas with color.

/Paint3DLine — x y width **/Paint3DLine -**

Draws a 3-D line with a 1-point light line above a 1-point dark line having endcaps either all light or all dark.

/Paint3DBox — x y width height down? **/Paint3DBox -**

Paints a 3-D box defined by the arguments using **/BG*** colors. When down? is true the box appears pressed in. When down? is false the box appears "out."

/Paint2DBox — x y width height bold? **/Paint2DBox -**

Paint a 2-D box using the 2DFG and 2DBG colors, whose bounding box is defined by the arguments. When bold? is true the box appears stroked with a double thick line.

/StrokeCanvas

color inset /**StrokeCanvas** -

Draws a border inset units inside the canvas, in the current transformation matrix (CTM) and strokes it with color. For information on the CTM see Adobe Systems *PostScript Language Reference Manual*.

/TopLeftPath

x y width height /**TopLeftPath** -

Constructs a path of the top and left edges of the box defined by x y width and height.

Fonts

/TextFont

/**TextFont** font (Variable)

The default font used for the canvas. The default value is /LucidaSans 12 point. Setting this variable does not automatically make font the current font. In order to make font the canvas's current font you could do:

```
TextFont setfont
```

/settextfont

font /**settextfont** -

Subclasser method: /ModifyFont

Defines the text font for a canvas. /**settextfont** can be sent to a class or an instance. For example to change the canvas's text font you would do something like:

```
/TimesRoman findfont 12 scalefont /settextfont yourcanvas send
```

As with /**TextFont**, the code above doesn't make font the current font. In order to have the PostScript show operator use the text font sent in the example you would have to make the font the current font using the PostScript **setfont** operator. /**settextfont** invalidates canvas instances.

/textfont

- /**textfont** font

Returns the text font of the canvas.

/ModifyFont

font /**ModifyFont** font'

Returns a font that is the same as the given font except it is not printermatched and it uses ISOLatin1 encoding. If you want to use printermatched fonts and/or some encoding other than ISOLatin1 you should override /**ModifyFont**. (For an explanation of printermatched font see the *NeWS 2.1 Programmer's Guide* and the *X11/NeWS Server Guide*.)

Cursors

/setcursor — cursorobject | name | null /**setcursor** -

Sets the canvas cursor. If null is given as the argument then the canvas inherits the cursor of its parent canvas. name is one of the named cursors in the shared /**Cursors** dictionary. Currently the names are: panning, xhair, hourg, basic, beye, nouse, copy, stop, move, navigation, rtarr, busy. xcurs, and ptr.

A cursorobject created using the NeWS operator newcursor can also be given as an argument (See the *NeWS 2.1 Programmer's Guide* for information on newcursor.)

/Cursor — /**Cursor** (Variable**)**

Default cursor for the canvas. Default is null, which means that the canvas inherits the cursor of its parent.

/Cursors — /**Cursors** dictionary (Variable)

The dictionary that stores the canvas's cursors. You can create your own cursor and put it into the /**Cursors** dictionary.

The canvas tree

/children — - /**children** array

Returns an array of canvases; the children of the canvas in the canvas tree. The array is ordered from the bottom canvas first to the top canvas last.

/descendants — - /**descendants** array

Returns an array of canvases, the descendants (if any) of this canvas. The array always contains self and is ordered by proximity to this canvas, children before grandchildren. Within each generation, top canvases come before bottom canvases.

/framebufferof — - /**framebufferof** framebuffer

Returns the framebuffer on which the canvas is located.

/Mapped

/**Mapped** boolean (NeWS Variable)

Determines whether the canvas is mapped. The default is for /**Mapped** to have the same value as /**Transparent**. By default instances of ClassCanvas are transparent and therefore mapped (See *Canvas damage handling* on page 51 for an explanation of /**Transparent**.)

/map

- /**map** -

Sets the NeWS /Mapped attribute of the canvas to be true. Painting a canvas does not cause it to appear on the screen unless the canvas and all its parents have also been mapped.

/mapped?

- /**mapped?** boolean

Returns whether the canvas is currently mapped onto the screen. A true value does not mean the canvas is currently visible on the screen. A mapped canvas wouldn't be visible if:

- It hasn't been painted yet.
- It is positioned off the edge of the screen.
- It is covered by other canvases.
- It is a child of an unmapped canvas.

See the *NeWS 2.1 Programmer's Guide* for an more complete explanation of mapped canvases.

/parents

- /**parents** [ancestor1 ancestor2 . . . framebuffer]

Returns an array of the canvas' ancestors ordered from its immediate parent first to the framebuffer last. The array does not include the GlobalRoot or self.

/reparent

parentcanvas /**reparent** -

Moves the canvas in the NeWS canvas tree so that parentcanvas is its parent. You are responsible for any layout or reshaping that must be done. The child canvas is mapped after it is reparented if it was mapped before the reparenting.

Note – Do not call this method for clients of bags. Bags reparent their clients automatically.

/siblings — - /**siblings** array

Returns an array of all the sibling canvases of the canvas. Includes self. The array is ordered from bottom first to top last.

/tobottom — - /**tobottom** -

Moves the canvas to the bottom of its sibling list.

/totop — - /**totop** -

Moves the canvas to the top of its sibling list.

/unmap — - /**unmap** -

Unmaps the canvas.

Geometry

All of the methods in this section operate using the current transformation matrix (CTM).

/bbox — - /**bbox** x y width height

Returns the bounding box for the canvas.

/location — - /**location** x y

Returns the coordinates of the origin of the canvas.

/minsize — - /**minsize** width height

Returns the minimum size of the canvas. If the data your canvas manages requires you to enforce some minimum canvas size override /**minsize** to provide an appropriate value.

/move — x y /**move** -

Moves the origin of the canvas to the specified location. If you override /**reshape** to change the canvas's CTM you should override /**move** to compensate for the changed coordinate system. For example if the origin of the canvas has been translated in /**reshape** to be at the lower right instead of the lower left, you could correct for the translation by overriding /**move**:

```
/move { % x y => -
    /size self send             % x y w h
    pop 3 -1 roll add exch      % (x+w) y
```

```
        /move super send
    } def
```

/path — x y width height /**path** -

Builds a path that fits into the bounding box defined by the arguments. The default is for a rectangular path. Each canvas uses this method for defining its intrinsic shape given the desired bounding box. You should override /**path** for any canvas subclass that you want to be nonrectangular.

/preferredsize — - /**preferredsize** width height

Returns the preferred size for the canvas. The preferred size is some ideal starting size that you determine in a subclass by overriding /**preferredsize**. By default /**preferredsize** returns the canvas's minsize.

/reshape — x y width height /**reshape** -

Reshapes the canvas to fit the bounding box defined by the arguments. Invalidates the canvas. Uses the current transformation matrix. The canvas's new shape is constructed by /**path**. If you change the canvas's coordinate system in an override of /**reshape** you should override /**move** in the same way (see /**move** on page 43).

/reshaped? — - /**reshaped?** boolean

Returns whether the canvas has been reshaped yet.

/size — - /**size** width height

Returns the width and height of the canvas in the CTM.

Validation

The NeWS Toolkit uses a validation scheme to determine whether an object needs to have its visual presentation updated. One way to understand this is through the model-view portion of the model-view-controller paradigm. Remember, the model is a data object, representing application information and the view presents its model in a graphical fashion. TNT has several objects that manage other objects. For example, menus manage a list of items, and bags manage a collection of clients. The list of items in a menu and the collection of clients in a bag are the model. How they are displayed on the screen is the view.

Your application may allow the user to alter the model, by adding or deleting menu items, for example. When the user alters the list of items, TNT marks the menu as invalid by sending it the **/invalidate** method. Saying a menus is invalid is the same as saying that the model has changed but the view hasn't been updated to reflect those changes.

The **/paint** method is used to initiate the process that causes an object's view to match its model. But, in order to be efficient, TNT doesn't automatically repaint objects when they become invalid, i.e., TNT doesn't call **/paint** on invalid objects, you do. This optimization allows you to change several attributes of an object and only perform one repaint.

Certain operations are considered "basic" enough that they automatically update the view. **/setvalue** and **/setcolors** are examples of methods that do the automatic update. However, if the view is already invalid for other reasons (such as adding items) then these methods do not update the view.

/Valid?

/Valid? boolean (Variable)

Determines whether the canvas is marked as invalid. An invalid canvas is one that needs some operations executed before it can be repainted; a valid canvas is one that is ready to be painted. For example, to a bag valid means not requiring layout at the moment. ClassCanvas does not interpret what validity means itself, but leaves it to subclasses to do so by overriding the validation methods (/**invalidate**, **/valid?**, **/?validate** and **/validate**). Subclassers should call super in their override of the validation methods. E.g.,

```
/?validate {
    . . .
    /?validate super send
} def
```

/invalidate

- **/invalidate** -

Marks the canvas as invalid.

/valid?

- **/valid?** boolean

Returns whether the canvas is valid.

/?validate

- **/?validate** -

Validates the canvas if it is currently invalid.

/validate — - /**validate** -

Causes the canvas to become valid. If you override /validate you should be sure to include a /validate super send in your override.

Ability to accept user input

/setvisualstate — state /**setvisualstate** -

Sets the visual feedback state for a canvas. state is defined as one of /Active, /Inactive, or /Busy. /**setvisualstate** also determines whether users can interact with a canvas

/visualstate — - /**visualstate** state

Returns the current visual feedback state for the canvas.

Activation and deactivation

A canvas can receive user input if it is active. An active canvas is defined as a canvas that has a local event manager (LEM). See *ClassEventMgr* on page 48. When a canvas is created it has no event manager.

/activate — event-manager /**activate** -

Turns on event management for the canvas. Table 5-2 lists the types of events a canvas can receive and the section where each type of event is discussed.[1]

Table 5-2 Types of events a canvas can receive

Event	Section	Page
damage	Canvas damage handling	51
menu	Canvas menus	53

1. It is possible that you want your application to receive events that the Toolkit has not anticipated. You can express these additional interests in the LEM using ClassInterest. You create an instance of ClassInterest by sending /new to it: canvas action name /new ClassInterest send. You might was to modify some NeWS values in the resulting interest such as /Priority or /Exclusive. You would add this new interest instance to your event manager using /addclient:: interest-instance /addclient emgr send. (Note that the event manager is provided as an argument to /activate.
Interests passed to an event manager must specify executable matches; i.e., at least one of the matching fields in the interest must have a dictionary with executable values that will run on exit from the awaitevent. When all executable matches have run, the event must be consumed. Please see the *NeWS 2.1 Programmer's Guide*, Chapter 5—Events for a discussion on events, interests, and executable matches.

When you are using bags and the bag subclasses (which includes windows), you only need to activate the bag; all the bag's canvas clients are activated when the bag is activated.

Thus to activate a canvas you could do:

```
/new ClassEventMgr send /activate yourwindow send
```

/active? - /**active?** boolean

Returns whether the canvas has been activated.

/deactivate - /**deactivate** -

Turns off event management for the canvas. If the canvas does not have an event manager /**deactivate** does nothing.

/eventmgr - /**eventmgr** event-manager | null

Returns the event manager for the canvas.

/EventMgr /**EventMgr** event-manager (Variable)

The event manager for the canvas.

/EventsConsumed /**EventsConsumed** (NeWS Variable)

Determines which events that are tested for a match against the canvas post-child interests list are also matched against the post-child interests of the canvas's parent. Most programmer's won't need to change /**EventConsumed**.

/**EventsConsumed** is a NeWS canvas attribute. The default value is /MatchedEvents. Other legal values are /AllEvents and /NoEvents. When /EventsConsumed is set to /MatchedEvents events that match a post-child interest

of the canvas are consumed. Non-matching events may still be passed to the canvas' ancestors for further testing against post-child interests. See the *NeWS 2.1 Programmer's Guide* for a complete explanation of event processing.

Callback context

The context of a callback refers to the process in which the callback executes. You can find a discussion of NeWS processes in the *NeWS 2.1 Programmer's Guide*. The process maintains state in several ways, four are especially important to TNT:

- The dictionary stack, especially the process's userdict.
- The operand stack.
- The stdin/stdout pair, which is the client's wire. (See The Wire Service on page 227.)
- The graphics state.

In most cases TNT arranges that client callbacks are invoked in the context defined by the client userdict and connection (stdin/stdout pair). In general, this occurs in the application's local event manager (LEM). Only obsolescence and certain ClassSelection methods are exceptions to this.

One implication of this arrangement is that the notifier procedures for controls (e.g., buttons) shouldn't take a long time to execute in the server. If the notifier needs a long time to execute, it should spawn a separate process to do it in. If you don't spawn a process for time-consuming procedures, the LEM, being busy with the execution, can't respond to other requests, e.g., **/TrackStart** and **/MenuStart**. You *can* send a message over a wire that makes the client take a long time because this frees up the LEM.

When a callback is invoked in the global event manager's process (the GEM), and the code requires access to the client's context, there are two methods available that provide this access—**/callmanager** and **/sendmanager**.

ClassEventMgr

Global event manager

The NeWS Toolkit has a single global event manager that is an instance of ClassEventMgr. The global event manager (GEM) provides a process where synchronous event-processing occurs. Synchronized events are executed in the

GEM. Non-synchronized events are passed to a per-client event manager, the local event manager (LEM). The GEM also redistributes many events to the LEMs after ensuring proper syncronization.

See *Callback context* on page 48 for information on where callbacks execute.

Local event manager

Local Event managers (LEM) are instances of ClassEventMgr. An event manager is a NeWS process that receives events that the canvas is interested in (For information on NeWS processes see the *NeWS 2.1 Programmer's Guide*). You can share event managers between different windows by creating a pointer to a single instance of ClassEventMgr and then passing the reference to **/activate** when you activate each window:

```
/emgr /new ClassEventMgr send def
emgr /activate MyFirstWindow send
emgr /activate MySecondWindow send
```

See **/activate** on page 46 for more information on activating canvases.

Creation

/new - **/new** event-manager

Creates an event manager instance.

Error handling in event managers

/Robust? **/Robust?** boolean (Variable)

Determines whether the event manager is "robust." The robustness of an event manager determines how it reacts when it gets a error (see the *NeWS 2.1 Programmer's Guide* and **/HandleError** below). The default is false.

When a robust TNT event manager gets any error[1] it:

- clears all the send contexts which includes any classes on the dictionary stack and the history of any send contexts.

1. The NeWS operator killprocess is implemented as an error. Thus a robust event manager is not killed by killprocess.

- removes all dictionaries from the dict stack except the user- and the systemdict.
- prints out an error message from $error dict (if there is one) to the standard output file.
- continues to run.

When a non-robust event manager gets any error other then a killprocess error it:

- prints out an error message from $error dict (if there is one) to the standard output file.
- is killed using quit.

However, if the non-robust event manager gets a killprocess error it just quits without printing out any error messages.

/robust? - /**robust?** boolean

Returns the value of /**Robust?**.

/setrobust boolean /**setrobust** -

Sets the value of /**Robust?**.

/HandleError - /**HandleError** -

Event manager's error handler. When an event manager gets an error the NeWS Toolkit calls /**HandleError**. Override /**HandleError** if you want error handling behavior that's different from that described in /**Robust?**. If you do override /**HandleError** the robustness is no longer enforced and may be ignored.

Changing event managers for execution and sends

/callmanager proc /**callmanager** -

Executes proc within the event manager's process. You send /**callmanager** to an instance of ClassEventMgr, not to a canvas.

/sendmanager [<args> /method object] /**sendmanager** -

Sends /method with its accompanying arguments (if any) to object within the event manager's context. You send /**sendmanager** to an instance of ClassEventMgr, not to a canvas.

Canvas damage handling

A canvas is considered damaged if all or part of its image is incorrect and needs to be repainted. Only those canvases that are not transparent can receive damage. For an explanation of damage see the *NeWS 2.1 Programmer's Guide*.

The rendering model that TNT uses for fixing damage is similar to the one used for painting (see Painting on page 38). One of the biggest differences is that you never need to call the method that initiates the damage fixing recursion, it is called for you by the Toolkit's damage handling callback, **/HandleDamage**. For damage repair, **/HandleDamage** establishes the context by setting the clippath and then calls damage-fixing initiator, **/FixAll**. From the point where **/FixAll** is called for you, the damage repair recursion is the same as for painting (except for the method names of course). **/FixAll** calls **/Fix** and **/FixChildren**.

/Transparent

/Transparent boolean (NeWS Variable)

Determines if the canvas obscures other canvases. If the value of **/Transparent** is true the canvas won't obscure other canvases; it is transparent. Opaque canvases have a **/Transparen**t value of false. Only opaque canvases can get damaged. Instances of ClassCanvas are transparent by default.

Instances of ClassCanvas are transparent by default.

/setdamageable

boolean **/setdamageable** -

Sets the value of **/Transparent** (to be the opposite of boolean).

/damageable?

- **/damageable?** boolean

Returns whether the canvas receives damage events (see also **/Retained**, page 53). Only a nontransparent canvas can receive damage.

/damage

- **/damage** -

Adds the canvas's path to the opaque parent's damage path via the NeWS operator **extenddamage**. This is used where **/paint** would be used but you don't want to take the time to paint because you're in the global event manager.

/damageall

- /damageall -

Adds the canvas path to opaque parent and children damage paths. Used when you want to damage not only the opaque parent but also its opaque children. Similar to **/damage** but uses the NeWS operator **extenddamageall** instead of **extenddamage**.

/FixAll

- /FixAll -

Subclasser methods: **/Fix**
/FixChildren

Paints the damaged area of the canvas. **/FixAll** is called for you by the damage handler method **/HandleDamage** after the damage path has been clipped to the canvas. It is not necessary for you to call it yourself.

By default, **/FixAll** calls **/Fix** and **/FixChildren**; override these methods in your ClassCanvas subclass rather than **/FixAll**.

/Fix calls **/Paint** which simply paints the damaged section of the canvas without trying to be efficient.

It is your responsibility to override **/Fix** if you want to provide more efficient damage repair behavior.

/Fix

- /Fix -

Handles the repainting of damaged portions of a canvas. Only the damaged part of the canvas is repainted, using **/Paint**. Despite the fact that only the damaged area of the canvas is painted, all the code in your **/Paint** definition is executed. If your **/Paint** has complicated, time-consuming calculations, you may want to override **/Fix** so you can execute only the painting code that applies to the damaged area. If your painting procedures are simple and not time-consuming (e.g., doing a FillCanvas followed by a show), you can just use your painting procedures "as is" and allow the server to clip away the bits that aren't necessary.

/FixChildren

- /FixChildren -

Handles the repainting of canvas children. Sends **/FixAll** to all the children. **/FixChildren** is a no-op for ClassCanvas; definitions are provided for ClassCanvas subclasses. FixChildren should ignore opaque (damageable) children since they will get their own /Damaged events.

/HandleDamage

event /**HandleDamage** -

Callback executed when a canvas interested in receiving damage events is damaged. It clips the canvas to the damage path then calls /**FixAll**. Because /**HandleDamage** clips the canvas to the damage path only the part of the canvas that got damage is repainted. /**HandleDamage** resets the canvasclip when /**FixAll** returns.

/SaveBehind

/**SaveBehind** boolean (NeWS Variable)

Hint to the server about how to handle damage to underlying canvases when this canvas is put on screen. Default is false. See the *NeWS 2.1 Programmer's Guide* for the discussion on the SaveBehind attribute.

/Retained

/**Retained** boolean (NeWS Variable)

Determines if a copy of the canvas is saved off-screen. In general retained canvases do not receive damage, except when they are first mapped and reshaped. However, if a Retained canvas is damageable and active, sending /**damage** to it causes its /**Paint** method to be called. Covering and then exposing a Retained canvas does not cause the /**Paint** method to be called.

See the *NeWS 2.1 Programmer's Guide* for a discussion of Retained canvases.

Canvas menus

/Menu

/**Menu** menu (Variable)

The menu that is managed by the canvas. By default, canvases have no menus.

/setmenu

menu | null /**setmenu** -

Installs or removes a popup menu for the canvas. The menu is activated by the user pressing the MENU button when the pointer is over the canvas. Can be sent to a class or an instance. You shouldn't send /**setmenu** to an object that hasn't been made Menuable (see /**Menuable?**, below).

/menu

- /**menu** menu

Returns the value of /**Menu**.

/Menuable?

/**Menuable?** boolean (Variable)

Determines whether a canvas should show its menu when MENU is pressed with the pointer over the canvas.

Subclass it if all instances of this type of canvas should or should not show their menus.

/setmenuable

boolean /**setmenuable** -

Sets the value of /**Menuable?**.

/menuable?

- /**menuable?** boolean

Returns the current value of /**Menuable?**.

/MenuStart

invoker posname event /**MenuStart** invoker posname event menu true
| invoker posname event false
| invoker posname event null true

Determines which menu (if any) should be shown over this object, and where it should be positioned. You can change where the menu pops up by:

- Overriding /**MenuStart** and modifying the XLocation, YLocation pair of the event;

and/or

- Changing the posname.

posname defaults to /Default, which means that the default item (or the item at index 0 if there is no default) is aligned beside the given point. Currently, the only other legal posname is /NorthWest, which corresponds to the top-left corner of the menu. (posname is used by menu buttons for positioning their menus.)

invoker is the object that most recently caused the menu to be displayed. The invoker also determines the local event manager in which the menu's notifier executes and is the default target of the menu. (See *The invoker mechanism* on page 117 in ClassMenu.)

By default /**MenuStart** method returns the menu held in the /**Menu** class/instance variable, and does not modify the positioning arguments.

Returning invoker posname event false causes event to be redistributed allowing some other object to open a menu.

The invoker posname event null true return is provided so that you can respond to MENU down over your canvas or region by not opening any menu and preventing anyone else (e.g., the window under the canvas or region) from opening a menu in response to the MENU.

This method is executed in the local event manager (see *Callback context* on page 48).

/MenuStop

menu /**MenuStop** -

Called when the menu (the root of the chain of submenus if such a chain is up) is brought down. You can subclass it if you need to perform any action at this time.

Mouse tracking

A Trackable canvas is one which supports interactions whose form is an extended transaction which begins with button down on SELECT and ADJUST, and before being terminated, may process mouse-motion, window crossings, shift-key transitions, and/or timer events.

All the tracking methods execute in the local event manager.

/Trackable?

/**Trackable?** boolean (Variable)

Determines whether the canvas is interested in mouse tracking events. Default is false.

/settrackable

boolean /**settrackable** -

Sets the value of /**Trackable?**.

/trackable?

- /**trackable?** boolean

Returns the value of /**Trackable?**.

/starttracktimer

initial repeat /**starttracktimer** -

Determines what the initial and subsequent timing intervals (in milliseconds) are for receiving /**TrackTimer** (see below) messages. initial is the time interval that determines how long from the receipt of the /**starttracktimer** message the first /**TrackTimer** message is sent to the canvas. repeat is the time interval that determines the time between /**TrackTimer** messages.

If repeat is 0, only one /**TrackTimer** message is generated (after initial milliseconds).

/**starttracktimer** can be sent from your /**TrackStart** method or at any subsequent time, e.g., during /**TrackCrossing**.

/stoptracktimer

- /**stoptracktimer** -

Stops the generation of /**TrackTimer** messages. This happens automatically when the tracking operation is completed (i.e., the mouse button is released).

Tracking subclass responsibility methods

If you make your canvas Trackable you must provide a definition for those methods that you want to respond to. A Trackable canvas is sent **/TrackStart** when SELECT or ADJUST goes down while the pointer is over the canvas. You use **/TrackStart** to determine which intermediate mouse actions (e.g., crossings, motions) your canvas is interested in. The possible intermediate mouse actions are:

- **/TrackCrossing**
- **/TrackMotion**
- **/TrackTimer**
- **/TrackStop**

All of the tracking methods are sent to you with the event as an argument. From the event you can get which button went down (stored in /Name) and the XLocation and YLocation of the event. You can get these coordinates by doing:

```
begin % event
XLocation YLocation
end % x y
```

/TrackStart — event **/TrackStart** false | [/label...] true | /name true | nullarray true

Determines which intermediate mouse actions a Trackable canvas is interested in. Returning false indicates your canvas is not interested in tracking this particular mouse action and causes event to be redistributed. Any client that returns a non-empty array from **/TrackStart** must also implement **/TrackCancel** (see page 57). **/TrackStart** executes in the local event manager.

A true return also indicates which intermediate actions should be tracked. The [/label. . .] true return from **/TrackStart** is an array of the intermediate mouse actions you are interested in (**/TrackCrossing**, **/TrackMotion**, **/TrackTimer** and/or **/TrackStop**). The /name true return simply selects the name of a TNT supplied array. The only predefined array is:

```
/Default [ /TrackCrossing /TrackMotion /TrackStop ] def
```

In addition there is a special case that is allowed: you could define **/TrackStart** to return:

```
nullarray true
```

(an empty array and true) if you wanted to find out that the mouse button was pressed but that's all you wanted to do. This return from **/TrackStart** means don't redistribute the event and don't send the canvas any of the other callbacks.

The situation whereby a canvas always returns false should not occur. Instead **/Trackable?** should be set to false.

/TrackStop — event **/TrackStop** -

event is the terminating event for this interaction (typically, button-up on the button that initiated **/TrackStart**). **/TrackStop** executes in the local event manager.

/TrackCrossing — event **/TrackCrossing** -

event is a crossing event into or out of the subtree rooted at the client canvas.

/TrackMotion — event **/TrackMotion** -

event is a MouseDragged event anywhere on the screen.

/TrackTimer — event **/TrackTimer** -

event signifies a timer has expired. See **/starttracktimer** and **/stoptracktimer**.

/TrackCancel — event **/TrackCancel** -

Cancels the tracking sequence that began with a **/TrackStart**. Equivalent to a **/TrackStop** (see above), but without carrying through the transaction. It might be generated by the STOP key, or a second press of an initiating button without an intervening release e.g., due to lost events in a world crossing. A world crossing occurs when a user presses a mouse button down over a NeWS display and releases it over a non-NeWS display. TNT never sees the mouse up and so sends **/TrackCancel** if the button is pressed again in the NeWS display.

Getting focus and keystrokes

/Keyable? — **/Keyable?** boolean (Variable)

Determines whether the canvas is a potential client for the input focus and therefore is interested in getting keystrokes. The default is false.

/setkeyable — boolean **/setkeyable** -

Sets the value of **/Keyable?**.

/keyable? - **/keyable?** boolean

Returns the value of **/Keyable?**

/KeyStart event **/KeyStart** false | [/label...] true | /name true | nullarray true

Determines which types of keys a canvas is interested in when it has the focus. The name of the event is either /AcceptFocus or /RestoreFocus. If the canvas doesn't want the event, i.e., doesn't want subsequent keystrokes or doesn't want the focus, it should return false. The situation whereby a canvas always returns false should not occur. Instead, **/Keyable?** should be set to false.

In most cases **/Keystart** should return true and an array of labels or a single name. The single name identifies one of a set of arrays defined by The NeWS Toolkit. The only array currently defined is:

```
/Default [/StandardKey /NumPadKey] def
```

The array of labels identifies which types of keys the canvas is interested in being notified about while it has the focus. Each such label is also the name of a method in the client canvas. The possible labels are:

/StandardKey	**/NumPadKey**	**/MetaKey**
/StandardKeyUp	**/NumPadKeyUp**	**/MetaKeyUp**
/ArrowKey	**/FunctionKey**	**/FunctionString**
/ArrowKeyUp	**/FunctionKeyUp**	**/FunctionStringUp**
/ArrowString		
/ArrowStringUp		

See each method below for an explanation.

Thus, for example, if **/KeyStart** returns [/StandardKey] (or /Default, which turns into an array that includes/StandardKey), then the client's **/StandardKey** method is called whenever an appropriate key goes down. (see **/StandardKey** and **/NumPadKey**, below.)

The nullarray true return (empty array and true) allows you to have a canvas that accepts the input focus but is not notified of keystrokes. One use of this **/KeyStart** return is for canvases that want to get the PASTE key but no other keystrokes. PASTE can be seen only by Keyable canvases.

/KeyStop — event /**KeyStop** -

Sent to the canvas when the canvas loses the focus. The default behavior is for the method to simply pop the event off the stack. Override /**KeyStop** if you want to provide loss-of-focus feedback (e.g., changing the state of the caret).

Methods that can be specified in /KeyStart

For each method below, the unsuffixed form has an event with /DownTransition in its Action field; the corresponding *Up message will have /UpTransition in its Action field. The /Name of the event describes the specific key, as detailed below.

If a client selects overlapping sets of keys, such as /**ArrowKey** and /**FunctionString**, only the callback listed first will be called for those keys. E.g.,

```
[/ArrowKey /FunctionString /FunctionKey]
```

calls /**ArrowKey** when an arrow key is pressed, and calls /**FunctionString** when a function key is pressed that is NOT an arrow key; /**FunctionKey** is never called in this example. If a client selects both down- and up-transitions for overlapping keys, e.g.,

```
[/ArrowKey /FunctionKey /FunctionKeyUp /ArrowKeyUp]
```

the first callback listed for each class of key determines the relative priority of the message.

/StandardKey — event /**StandardKey** -

/StandardKeyUp — event /**StandardKeyUp** -

event's Name is one of the standard typing array characters, possibly shifted by Control, Shift, or Caps.

/NumPadKey — event /**NumPadKey** -

/NumPadKeyUp — event /**NumPadKeyUp -**

event's Name is a character from the keyboard number pad. If the keyboard has a NumLock key, this method will be called only when a NumLock is in effect.

/MetaKey — **event /MetaKey -**

/MetaKeyUp — event /**MetaKeyUp** -

event's Name is one of the standard typing-array characters, possibly shifted by Control, Shift, Caps. Unlike /**StandardKey**, these methods are called *only* if the Meta key is held down. If you want /**StandardKey** to be called only when Meta is *not* held down, request both /**MetaKey** and /**StandardKey** (in that order) and provide a nullnotify /**MetaKey** method. (See /**KeyStart** for an explanation of the significance of the request order.)

/ArrowKey — event /**ArrowKey** -

/ArrowKeyUp — event /**ArrowKeyUp -**

event's Name is one of the names /Up, /Down, /Right, /Left, or /Home.

/ArrowString — event /**ArrowString** -

/ArrowStringUp — event /**ArrowStringUp** -

event's Name is a string holding the ANSI escape sequence for a cursor function corresponding to a cursor key (^[[A, ^[[B, ^[[C, ^[[D, or ^[[H for up, down, right, left, or home, respectively).

/FunctionKey — event /**FunctionKey** -

/FunctionKeyUp — event /**FunctionKeyUp** -

event's Name is of the form /FunctionF1, /FunctionR12, etc.

/FunctionString — event /**FunctionString** -

/FunctionStringUp — event /**FunctionStringUp** -

event's Name is a string holding the ANSI escape sequence for a function key. Its form is ^[[nnnz, with nnn a 3-digit decimal number in the range 192 - 207 for a key on the left pad, 208 - 223 on the right, 224 - 239 on top, and 240 - 255 for the "bottom"—many of which appear on the right side of a type-4 keyboard.

Targets of global function keys

Table 5-3 on page 61 shows which canvas is the target of the global function keys. The target can be one of the following:

Target	Explanation
Focus	the current input focus; a canvas must be Keyable to get focus.
Selection	the current selection.
Cursor	the object beneath the cursor.

Table 5-3 Global function key targets

Key	Target	Notes
Again	Focus	sent to the focus, with a default no-op handler in ClassCanvas
Copy	Selection	Handled by the current selection.
Cut	Selection	Handled by the current selection.
Find	Focus	sent to the focus, with a default no-op handler in ClassCanvas.
Front	Cursor	windows are Frontable by default.
Help	Cursor	
Open	Cursor	windows are Openable by default,
Paste	Focus	can only be seen by Keyable canvases (see Getting focus and keystrokes on page 57).
Props	Selection	
Stop	Various	various global input reinitializations occur and then it gets redistributed.
Undo	Focus	sent to the focus, with a default no-op handler in ClassCanvas.

Again, Find, and Undo keys

Canvases that want to be informed about presses on the Find, Undo and Again keys must be Keyable. The following handler methods have no-op definitions (they just pop the event off the stack). If you want your canvas to respond to these keys you must provide definitions.

/HandleAgain

event /**HandleAgain** -

The Again key callback.

/HandleFind — event /**HandleFind** -

The Find key callback.

/HandleUndo — event /**HandleUndo** -

The Undo key callback.

Opening and closing canvases

An Openable canvas is one that is a Window, so that it may be opened itself, or one that contains objects that may be opened, e.g., a file-viewer.

/Openable? — /**Openable?** boolean (Variable)

Determines whether the canvas is interested in "open" events and wants the /**HandleOpen** method called.

/setopenable — boolean /**setopenable** -

Sets the value of /**Openable?.**

/openable? — - /**openable?** boolean

Returns the value of /**Openable?**.

/HandleOpen — event /**HandleOpen** -

The method that is executed when an Openable canvas gets an open event. The recipient Window / object at the Open event's coordinates should be opened or closed or zoomed, per the /Action field of the event. The value of /Action is one of: /Open, /Close, /Zoom, /Unzoom, /Toggle.[1]

1. Implementation note: The event might be an up-transition on the OPEN key, or it could be an up-transition on the mouse's MENU button, or other events as defined by the UI. The code handling these various events will put the desired action (i.e.,one of /Open, /Close, /Zoom, /Unzoom or /Toggle) into the /Action field and then call /HandleOpen. Clients should not make any assumptions about the event except for the coordinates and /Action.

Help facilities

The TNT help facility provides a spot help facility on a per-application basis. A Helpable canvas undertakes to provide some help for any location in its canvas. In addition, the help facility uses the Wire Service and has client-side functions. These functions can be found in Chapter 26, The Wire Service, Help facilities on page 243.

/Helpable?

/Helpable?

Determines whether the canvas is added to the help service when the canvas is activated. Default value is false (not added).

/sethelpable

boolean /**sethelpable** -

Sets the value of **/Helpable?**.

/helpable?

-/**helpable?** boolean

Returns the value of /**Helpable**?.

/HandleHelp

event /**HandleHelp** -

Callback executed when a canvas that is Helpable gets a help event while the pointer is over the canvas.

/HelpKeyword

/**HelpKeyword** keyword-string (Variable)

The PostScript string that determines where the help system looks for the help information for the canvas and which object it's looking for. The keyword-string has the following format:

(filename:objectname)

filename should be located in a path specified by the `HELPPATH` environment variable. The format for the help file should be the same as the files in `$OPENWINHOME/lib/help` and should follow the same naming conventions as these files. For example, if you have a Helpable canvas, mycanvas, and a help file, my_application_help, the keyword-string for this object would be: my_application_help:mycanvas.

In my_application_help the help message for mycanvas would be in the format:

```
:mycanvas
<the help message>
```

/sethelpkeyword — keyword-string /**sethelpkeyword** -

Sets the value of /**HelpKeyword**.

/helpkeyword — event /**helpkeyword** keyword-string

Returns the value of /**HelpKeyword**. Subclassers may override /**helpkeyword** to provide messages over any special areas of their Helpable canvases. You would get the coordinates of the help event out of the event and determine whether that point was in the area you wanted to provide help for. /**HelpKeyword** is overridden in some classes of ClassCanvas to provide specialization.

Moving canvases along the z-axis (Front key)

A Frontable canvas is one which is a Window (Frame), which can be reordered in Z, or contains objects subject to such a reordering (e.g., a structured graphics editor).

/Frontable? — /**Frontable?**

Determines whether the canvas is interested in front events and wants the /**HandleFront** method called.

/setfrontable — boolean /**setfrontable** -

Sets the value of /**Frontable**?.

/frontable? — - /**frontable?** boolean

Returns the value of /**Frontable**?

/HandleFront — event /**HandleFront** -

Callback executed when a canvas that is Frontable gets a front event when the pointer is over the canvas. The value of the /Action field is one of /Front /Back, or /Toggle. Initially defined as a no-op.[1]

1. Implementation note: The event might be an up-transition on the FRONT key, or it could be an up-transition on the mouse's MENU button, or other events as defined by the UI. The code handling these various events will put the desired action into the /Action field (i.e.,one of /Front, /Back or /Toggle) and then call /**HandleFront**. Clients should not make any assumptions about the event except for the coordinates and /Action

Selectables

Selections come in two parts, the UI part which is discussed in this section and the data part, which is discussed in Chapter 20, ClassSelection. In addition, there is a demo in the TNT directory, called circles, which you can examine to see how the methods discussed in this section and in ClassSelection are used.

A selectable canvas must define **/SelectableType** to one of /Canvas, /Graphic, /Text, or /Dynamic and it *must* implement the notification methods listed in the section, *Selectable subclass responsibility methods* on page 67. Of the methods listed only **/SelectionCancel** has a default implementation. The other methods are considered strict subclass responsibility. See *Subclass responsibility* on page xviii in the Preface for a definition of "strict subclass responsibility methods.

/Selectable?

/Selectable? boolean (Variable)

Determines whether the canvas is interested in getting events as a selectable object. When true the canvas can receive selection events. Its default value is false. Further, the ClassCanvas variable **/Holder** is promoted when you instantiate and activate your canvas (see **/Holder** on page 66). You can set **/Selectable?** when you subclass ClassCanvas or on the fly using **/setselectable** (page 65).

/setselectable

boolean **/setselectable** -

Sets the value of **/Selectable?**

/selectable?

- **/selectable?** boolean

Returns the value of **/Selectable?**.

/SelectableType

/SelectableType /Text | /Graphic | /Canvas | /Dynamic (Variable)

Determines how the system should handle selections on the canvas; defaults to /Text. Set the value of **/SelectableType** when you create your canvas subclass. The possible values of **/SelectableType** are interpreted as follows:

- /Text indicates that text within the canvas can be selected. In addition, the selection UI provides the text highlighting styles shown in Table 5-4. These styles are possible values of the /Style variable in ClassSelection. The values are set for you by the selection mechanism but it is your responsibility to actually paint each style on the canvas.

Table 5-4 The text highlighting styles.

Highlight name	Explanation
/Default	the standard inverted highlighting
/UnderScore	paints a thin line *under* selected text to indicate a Quick Copy/Paste
/StrikeThrough	paints a thin line *through* the text to indicate a Quick Cut/Paste operation.

- A value of /Graphic means that graphic objects within the canvas can be selected. For example, in a file manager-type application you may be able to select icons and move them around.
- When the value of /**SelectableType** is /Canvas, the canvas itself can be selected. For example, you may want to select windows so you can move groups of windows around the screen. For a /**SelectableType** of /Canvas, /Level has no meaning. For an explanation of /Level see Table 20-2 on page 180 in ClassSelection.
- A value of /Dynamic means that Selectable objects within the canvas can change type between /Text and /Graphic. This dynamic changing of type is useful for graphics editors that have different modes for manipulating text as a graphic object as well as normal text operations like insertions and deletions of characters. For example text selected with a modifier key held down might display the control points of its bounding box. Canvases of /**SelectableType** /Dynamic must implement /**IdentifySelectable** (page 71).

/Holder

/**Holder** canvas (Variable)

Determines which canvas is a selection client. If /**Selectable?** is true the Toolkit sets /**Holder** to be the canvas. However, if a single selection may exist in two or more canvases (e.g. a split view in a text editor, or several icons on the desktop), then the Selectables for those two canvases should have the same /**Holder**. This supports such behavior as starting a selection in one canvas of a split view and then extending it by clicking in the other canvas. This single holder can be an instance of ClassCanvas or any other object. For reasons of reference counting and garbage collecting it is recommended that the single holder for multiple views be a canvas.

/setselectionholder — any /**setselectionholder** -

Resets the value of /**Holder**. This method should not be called within the bounds of a SelectionStart / SelectionStop or DragStart / DragStop pair. For correct memory management, the argument should be an object that can go obsolete —not a name or number; null is valid only when /**Selectable?** is false.

/selectionholder — - /**selectionholder** any

Returns the current value of /**Holder**.

Selectable subclass responsibility methods

In order to implement Selectable canvases you must provide definitions for the methods in this section. One use for the event that is given as an argument to these methods is to determine where the event occurred if you need that information; event gives you the current-transformation-relative point.

/NewSelection — event rank /**NewSelection** selection

Returns an instance of your subclass of ClassSelection. For information on how to create a subclass of ClassSelection see *Making selections* on page 179 in ClassSelection. The event input queue is blocked until /NewSelection returns.

Subclasses of ClassSelection require rank and holder arguments to the /**new** method. TNT puts the rank on the stack and because /**Holder** evaluates to the canvas you can use it to put the Holder on the stack prior to your call to /**new**:

```
/NewSelection { % event rank => selection
    Holder /new YourSelectionSubclass send% event sel
    exch pop % sel
} def
```

The first line puts the Holder (the canvas) on the stack and uses it and the rank as arguments to /**new**. The second line simply pops the event off the stack and leaves the selection.

The selection mechanism keeps a reference to this selection instance. The reference is used for all subsequent interactions between the selection mechanism and the selection client from the time /**NewSelection** is called until your selection instance is destroyed.

/SelectionContext

event selection | event null /**SelectionContext** /SelectedObject
| /UnselectedObject
| /Background
| /SelectedObject boolean
| /UnselectedObject boolean
| /Background boolean

Returns one of the listed context names, (or, optionally, a context name and a boolean) depending on the relationship between the coordinates of event and the location of the current selection (if any). **/SelectionContext** is called to determine whether to start a selection or a bounding box.

Table 5-5 Selection context names

Name	Location of event
/SelectedObject	Inside the current selection. You could start a drag or toggle a multiobject selection.
/UnselectedObject	Inside an object that is not selected but could be. Don't start a drag but you could toggle the selection.
/Background	No object is selectable at that location. You could cause the bounding box of a selected object to be displayed. Not meaningful for selectables of type /Text.

The Toolkit allows selections to be initiated using ADJUST, however, conditions can arise that are ambiguous. The Tookit will occasionally need to determine whether the pointer is over a selectable object when there is no currently registered selection in that canvas (Or when the selection is not of an appropriate /Rank for the check in progress.) When this occurs, the selection parameter to **/SelectionContext** will be null.

Under certain circumstances the selection UI will send **/SelectionContext** to your canvas with an event and null as arguments. This send can occur if a user tries to extend a selection in a way that can't result in a new selection. For example, in a graphics editor with some object selected, the user presses ADJUST in the editor's background. No selection is created or extended and the canvas is sent **/SelectionContext** with event and null as arguments. In this case the returned name should not be /SelectedObject.

You can choose that your **/SelectionContext** method return a context name and a boolean. The context name boolean return is only meaningful for /Graphic /SelectableTypes. true means the event's coordinates are inside the context found by the last call to **/SelectionContext**. You keep track of where the selection context is. When the object selected is different than the last call to **/SelectionContext**, return the selection context name and false.

/SelectionStart

event selection **/SelectionStart** boolean

Resolves the coordinates of the event to an object and starts a selection on it with selection's attributes. (See Table 20-2 in ClassSelection, page 180 for a discussion of selection attributes.) The object is determined by the value of **/SelectableType**. selection is the instance that was created by the previous call to **/NewSelection**. **/SelectionStart** is called for every selection transaction in which a selection is made or adjusted and may be preceded by calls to **/IdentifySelectable** (page 71), **/SelectionContext** (page 68), and/or **/NewSelection** (page 67).

In those cases where you want to reject the selection, **/SelectionStart** should return false. Returning false allows the Toolkit to pass the event on to other possibly-interested objects.

Adjusting the selection

Adjusting a selection refers to altering its physical extent, i.e., you extend or reduce the selection. For example a text selection comprised of a single word might be adjusted to include an entire paragraph, or a selection consisting of a single graphic object might be adjusted to include other objects.

/SelectionAdjust

event selection **/SelectionAdjust** -

Adjusts the boundary of the given selection to lie on the object at the event's coordinates. Sent to the canvas when the selection is extended either by mouse movement with SELECT or ADJUST held down or when ADJUST goes down. **/SelectionAdjust** is always preceded by a **/SelectionStart** (page 69) and followed by either **/SelectionStop** or **/SelectionCancel** (page 71).

It is possible for a **/SelectionAdjust** to indicate a point outside the area in which contents can be displayed (e.g., off the bottom of a text window). This supports an auto-scroll feature, such as defined by the OPEN LOOK user interface. When an application gets such an **/SelectionAdjust**, it should (if possible) scroll some new data into the visible region from the hidden region indicated by the

location of the **/SelectionAdjust**, and select everything up to that border. It should repeat this process as long as **/SelectionAdjust** messages continue to be received.

/SelectionStop — event selection **/SelectionStop** -

Signals the end of the adjusting operation. **/SelectionStop** is always called to complete a selection transaction begun by **/SelectionStart** and not cancelled by **/SelectionCancel**. Use **/SelectionStop** to clean up anything you created (e.g., an overlay canvas) in **/SelectionStart**.

Dragging the selection

The drag methods are used for drag and drop operations. A canvas must be Receptible in order to handle drag and drops, see *Drag and Drop—receptible canvases* on page 71. The methods in this section assume a selection already exists, i.e., you've already created your subclass of ClassSelection and it's been instantiated. The method calling sequence for dragging is:

1. SELECT is pressed over the selection.
2. **/DragStart** is sent to your canvas. the drag image is created here.
3. With SELECT still down the drag image is moved. Your canvas is sent **/DragAdjust**.
4. SELECT is released. Your canvas is sent **/DragStop**. Clean up whatever user feedback you constructed in the **/DragStart**. If SELECT is released over a Receptible canvas, then that canvas is sent **/HandleReception**. See **/HandleReception** on page 72.

/DragStart — event selection **/DragStart** -

Initiates user feedback for a Drag (direct-manipulation move or copy) of the given selection; e.g., start an overlaid image of the value being dragged. If a grasping point is needed (i.e., if the cursor coordinates are needed to position the feedback) use the coordinates of event. **/DragStart** is executed in the local event manager but with the input queue blocked.

/DragAdjust — event selection **/DragAdjust** -

Moves a drag-image so its grasp-point is at the coordinates of the given event or gives other feedback of a drag in progress. For example if you are moving a selectable canvas by dragging around a wire frame with the mouse the

selection UI code calls **/DragAdjust** to tell you where the mouse is (the coordinates of event) so you could move the wire frame there. **/DragAdjust** is always preceded by a **/DragStart** and followed by a **/DragStop** or **/SelectionCancel**.

/DragStop — event selection **/DragStop** -

Signals the end of the dragging operation. Sent to the canvas when the button that initiated the drag goes up.

/SelectionCancel — selection **SelectionCancel** -

Sent to a Selectable canvas when a selection-in-process gets cancelled. In general this occurs only when the user presses the STOP key or starts a selection in a TNT window then lets up on SELECT over a SunView window where NeWS can't see the release. In the latter case when the user moves the pointer back to a TNT window and presses SELECT, TNT gets a /MouseDown on a button thought to be already down. TNT sends **/SelectionCancel** in this case to clear the selection originally begun.

/IdentifySelectable — event **/IdentifySelectable** selectable-type target true | false

Sent to a Selectable canvas whose selectable type is /Dynamic. The canvas must determine and return its current selectable type, (either /Text or /Graphic, see **/SelectableType** on 65) at the location of the event. target is the object to which other subclass responsibility selection methods should be sent. target may be the same canvas that got the **/IdentifySelectable** message, but in the case of a bag containing Selectable regions, target is the region containing event. A false return means no selection can currently be made at the given location.

Drag and Drop—receptible canvases

A Receptible canvas is one that is interested in being on the receiving end of a drag and drop operation. These operations are represented by events with the Name /TransferSelection. A reception client may choose to reject a transfer, for example, if the selection being transferred cannot render itself in a suitable data format. You must implement either **/HandleReception** or **/AsciiReception**, which are listed in the subclass responsibility section, to use The Toolkit's drag and drop facilities.

/Receptible? — -/**Receptible?** boolean

Determines whether the canvas is interested in getting drag-and-drop events.

/setreceptible boolean /**setreceptible** -

Sets the value of /**Receptible?**.

/receptible? - /**receptible?** boolean

Returns the value of /**Receptible?.**

Handling the drop—subclass responsibility methods

/HandleReception event selection /**HandleReception** boolean

Insert selection at the "focus" or at the coordinates given in the event, as described below. Some clients may choose to ignore this distinction if insertions are permitted only at a single location (e.g., current type-in point).

In receptible canvases /**HandleReception** must distinguish not only the type of transfer but also the destination of each type of transfer.

The types of transfers are:

1. A Move (the source is deleted after the transfer is completed).
2. A Copy (the source is not deleted after the transfer is completed).

The two destinations are:

1. The receptible canvas's insertion point (for a Paste or Quick-Paste)
2. The cursor location (for a Drop).

The type and destination of selections combine to create four cases. Each case is distinguished by the name in the Name field of the event argument passed to /**HandleReception**. See Table 5-6 for the value of the Name field for each type of transfer.

Table 5-6 Transferring selections and event name

Transfer type	Destination	/Name (of event)
Copy	Insertion Point	/CopyToCaret
Copy	Cursor Location	/CopyToLocation
Move	Insertion Point	/MoveToCaret
Move	Cursor Location	/MoveToLocation

/HandleReception returns true if the transfer succeeded; false if it fails, or to cause the event to be redistributed further up the canvas tree. (E.g., if an icon is dropped on a text field it might fall through to the parent canvas or even to the desktop.)

The default method extracts the selection contents as an ASCII string (returning false if the selection cannot represent itself in that form) and calls **/AsciiReception** to process the string. Thus you must either override **/HandleReception** or add **/AsciiReception**.

/AsciiReception

event string **/AsciiReception -**

Override to receive ASCII selections (i.e., what you get from /ContentsAscii in a selection; see Chapter 20, ClassSelection, Table 20-2) or override **/HandleReception** to get non-ASCII selections. **/AsciiReception** has no default implementation.

Obsolescence and destruction

Every object is automatically the object of an interest in its own obsolescence (the disappearance of the last hard reference to it). When the last hard reference to an object is broken and some soft references exist, the object is sent the obsolete message, in the context of the global event manager; TNT catches the obsolete event and sends **/destroy** to the object.

It is also useful for some objects ("aimers") to respond to the obsolescence of other objects ("targets"). The interface for this is:

```
target aimer /addclient ObsoleteService send
target aimer /removeclient ObsoleteService send
```

When a target goes obsolete, the aimer is sent a message in the GEM context:

```
target /HandleObsoleteTarget aimer send
```

When this happens, the aimer does *not* need to call **/removeclient** to unregister its interest in the target; this is done automatically. Subclasses that override **/HandleObsoleteTarget** should always do a super send.

/destroy

- /destroy -

Destroys the canvas and any children it manages (e.g., its menu).In a garbage collected system you get rid of an object by dropping any references you created to it. If the toolkit holds references to your objects, they will be dropped when you drop your references: you don't need to worry about what reference the toolkit maintains.

You should override **/destroy** in your subclasses when instances of these classes hold references that directly or indirectly point back to the instance. The parent -> child -> parent circle between a bag and its regions is an example of this kind of reference. In this case your **/destroy** should break the circle and call:

```
/destroy super send.
```

You can also override **/destroy** to send /destroy to other objects you're managing in order to minimize the generation of obsolete events.

/destroy is executed in the global event manager.

ClassControl

ClassControl is a mixin class and is not intended to be instantiated itself. The purpose of ClassControl is to establish a common interface for those classes (generally controls like buttons, scrollbars and settings) that execute notifiers, have default choices, and present a visual indication of the object's ability to accept user input.

ClassControl is directly mixed into the following classes of controls:

- ClassButtons
- ClassHSlider
- ClassScrollList
- ClassSettings
- ClassVScrollbar
- ClassTextField

In addition, because menus have many of the same characteristics as controls, i.e., they execute notifiers and have default choices, ClassControl is also mixed into:

- ClassMenu

Most controls change in discrete quanta: e.g., when you turn a settings item on or off, or when you drag and release a slider. In some cases, where it is not obvious how often the client wishes to be notified, the Toolkit broke

notification into two levels: notifier and previewer. Clients who wish intermediate notification can obtain it via the previewer (e.g., while dragging a slider).

An additional set of interfaces implementing The NeWS Toolkit's target mechanism is also included in ClassControl. Targets allow you to safely keep a reference to one object inside another object. Target references are "safe" in the sense that they look after all the associated NeWS reference counting issues. For a discussion of NeWS memory management see the *NeWS 2.1 Programmer's Guide*.

Controls and menus use targets because they have notifiers that specify the action that takes place when a button is pressed, a slider is dragged, a menu item is selected and so on. Typically, this action consists of sending a message to some other object. The target mechanism maintains a soft reference to this other object.

When an object has only soft references, NeWS generates an obsolete event. The NeWS Toolkit's target mechanism listens for obsolete events on targets and ensures that when a target goes obsolete, the Toolkit's soft reference to the target is broken. When the last soft reference is broken the target can get garbage collected.

Control values

With the exception of buttons, controls also possess an internal value which is presented in a visual manner to the user. As the user interacts with these controls, the internal value changes. This value can also be modified through programmatic means, which would update the visual representation. A simple example would be a check box which maintains an internal value to indicate whether or not it is checked.

The interpretation of the value of a control is dependent on the type of control. For controls with items, buttons, and settings, the value of the control represents the zero relative integer referencing the chosen item. Sliders define their value as an integer constrained between a minimum and maximum range. In text fields value is a string.

/setvalue

value /**setvalue** -

Sets the value of the control.

/value - /**value** value

Returns the value of the control.

Notification and previewing

Notification is performed within the context of the Local Event Manager (LEM). See ClassCanvas, Event Management.

/setnotifier notifier | null /**setnotifier** -

Sets the notifier in the control. When notifier is specified as a PostScript name type, it is used in conjunction with the target interfaces to dispatch notification to the appropriate target. During notification, the current value, the control instance, and the notifier name are pushed on the stack, prior to invoking /**sendtarget**. In other words, your notifier should be written to take a value and an instance as arguments. (See *Managing references between controls and other objects*, below.) Using null makes the notifier a no-op, i.e., it turns off notification. The default is for the notifier to be null.

Although not recommended, notifier can also be specified as a PostScript code fragment. The current value, the control instance, and the fragment are pushed on the stack, then the fragment is executed.

/notifier - /**notifier** notifier

Returns the notifier for the control.

/setpreviewer previewer | null /**setpreviewer** -

Sets the previewer procedure for the control. null removes the previewer from the control. You should see the individual controls to see if and when previewing is done.

/previewer - /**previewer** previewer

Returns the control's previewer procedure.

/ExecuteNotifier value notifier | previewer /**ExecuteNotifier** -

Executes the given notifier or previewer in the context of the local event manager. If you don't want your control notifier to execute in the LEM then you should subclass that control to override /**ExecuteNotifier** in order to provide a different context for notifier execution.

Managing references between controls and other objects

/settarget — object /**settarget** -

Sets object as the target of the control's or menu's notifier. If a previous target exists it is overwritten.

/cleartarget — null | object /**cleartarget** -

Clears the target. If null is given the target is cleared. If object is specified then the target is cleared only if object and the target are the same. This latter specification ensures that the target is not incorrectly cleared.

/sendtarget — arguments /method /**sendtarget** results

Sends /method and any required arguments to the target.

/target — - /**target** null | object

Returns the target.

/HandleObsoleteTarget — object /**HandleObsoleteTarget** -

Use for breaking reference chains. The default is to perform /**cleartarget**.

Destruction

/destroy — - /**destroy** -

Destroys the control.In a garbage collected system you get rid of an object by dropping any references you created to it. If the toolkit holds references to your objects, they will be dropped when you drop your references: you don't need to worry about what reference the toolkit maintains.

You should override /**destroy** in your subclasses when instances of these classes hold references that directly or indirectly point back to the instance. The parent -> child -> parent circle between a bag and its regions is an example of this kind of reference. In this case your /**destroy** should break the circle and call:

```
/destroy super send.
```

You can also override /**destroy** to send /destroy to other objects you're managing in order to minimize the generation of obsolete events.

/**destroy** is executed in the global event manager.

Display Items

Display items are not a class, rather they are a set of lightweight drawing specifications and associated utilities. You shouldn't rely on being able to reference userdict from display items. Make sure that they are completely self contained, or depend just on the object on which they are being drawn.

All display items have a "size" or bounding box and they are rendered starting with the current point at the lower left of that bounding box.

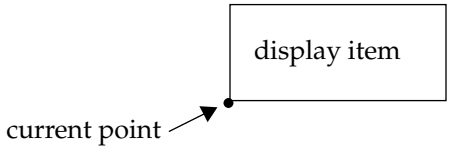


Figure 7-1 Where display items are rendered

A display item is a simple specification that allows for a visible part, called the "atom," and some number of modifiers to affect the rendering of the atom. Most places in the toolkit where you need to supply a string, such as for a label or a menu item, you can, in fact, provide any display item.

The display item specification looks like:

```
atom
```

or

```
[ atom modifier modifier ...]
```

where atom is one of the following:

string, canvas or executable array

and a modifier is one or both of the following:

font, color

The case where the atom is a string is the most common. The string is shown in the current font with its bounding box to the right of the current point. The size of the display item is determined by the bounding box of the string. Moreover, if the font extends below its baseline, the baseline of the font will be above the current point.

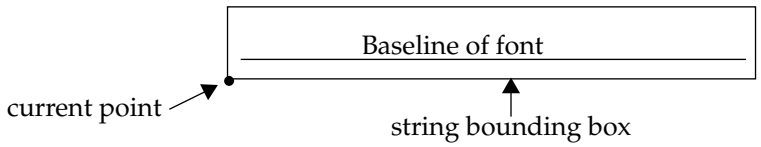


Figure 7-2 The relationship of a string's bbox, font baseline and the current point

If the canvas atom is used, the canvas is rendered as an image at the current point. When the /Color key in canvas is true, imagecanvas is used to paint the canvas; otherwise, imagemaskcanvas renders the canvas.

Most of the modifiers for a display item change the current graphics state in some way. A modifier that is a font will set the current font to new font. A color modifier changes the current color. Here are a few examples of valid display items:

```
(test)
[ (red test) 1 0 0 rgbcolor ]
[ (red times test) /Times-Roman findfont 12 scalefont 1 0 0 rgbcolor ]
```

The executable array form of an atom is used to supply a drawing procedure for the atom; it is a PostScript fragment that can respond to **/paint** and **/size**. The executable array must take one argument on the stack, a name that is either **/size**, or **/paint**. The executable array should return the width and the height that the drawing will occupy if the argument is **/size**. The image should be rendered in the current NeWS graphics state if the argument to the executable array is **/paint**. The following is an example of the executable array form of an atom:

```
{
    /paint eq { % use currentpoint to paint
        gsave
        0.5 setgray 15 15 rmoveto currentpoint 2 copy
        10 10 360 arc fill 10 0 360 arc 0.8 setgray stroke
        grestore
    } { % /size
            31 30 % width height
    } ifelse
}
```

Notice that the array should assume that the current point, current color, etc. are available in the current NeWS graphics state.

Utility procedures

The procedures in this section are defined in ClassCanvas and ClassRegion and must be called within the send context of a canvas or region.

DisplayItemPaint

display-item **DisplayItemPaint -**

Paints the display-item.

DisplayItemSize

display-item **DisplayItemSize** width height

Returns the width and height of the display item.

DisplayItemMaximumSize

[display-item . . .] **DisplayItemMaximumSize** width height

Returns the maximum width and height for an array of display items.

DisplayItemRect

/paint width height **DisplayItemRect** -

/size width height **DisplayItemRect** width height

Used to build an executable-array type of display item. For example,

```
[ {20 20 DisplayItemRect} 0 0 1 rgbcolor
```

builds a blue 20x20 display item.

Gauges

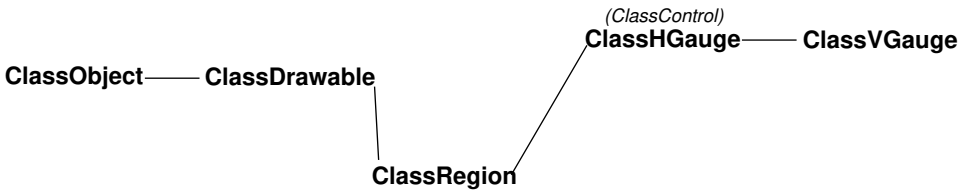


Figure 8-1 The gauge subtree

A gauge is a linear read-only control that displays a numerical value bounded by minimum and maximum values. By read-only it is meant that gauges are not manipulated directly by users and thus have no notifier or previewer.

Creation

/new

parentcanvas /**new** gauge

Creates a gauge instance. Send to ClassHGauge to create a horizontal gauge and ClassVGauge to create a vertical one.

Gauge values

/setvalue

value /**setvalue** -

Sets the value of the gauge. Any number can be given to /**setvalue** but it is normalized and forced in range. The gauge is updated to reflect the new value if the gauge is valid.

/value

- /**value** value

Returns the value of the gauge. The number returned by /**value** is always normalized and in range. See page 84, /**setnormalizer** for information on normalization.

/setrange

minimum maximum /**setrange** -

Sets the minimum and maximum (inclusive) value of the gauge. The default range is 0 to 100, inclusive. After changing the range, if you want the gauge to be large enough to display every whole number in its range, you should reshape the gauge to its preferred size. Invalidates the gauge.

/range

- /**range** minimum maximum

Returns the minimum and maximum value of the gauge

Gauge granularity.

/setnormalizer

proc /**setnormalizer** -

Sets the normalizer procedure. Invalidates the gauge. The normalizer procedure controls the gauge granularity. The normalizer procedure takes the unconstrained value on the stack and leaves a constrained value on the stack. A typical normalizer would be "{round cvi}", to constrain the value to rounded integers. By default the normalizer is defined as a null proc, which does not change the value.

/normalizer

- /**normalizer** proc

Returns the normalization procedure

Visual presentation of gauges

/settickmarks

distance /**settickmarks** -

Specifies that you want tick marks to be drawn below the gauge. Invalidates the gauge. distance is the space, in gauge value units, between consecutive tick marks. 0 specifies no tick marks (the default). You should reshape the gauge to its preferred size after adding or eliminating the tickmarks because gauges with tickmarks are a different size than gauges without them. "Gauge value units" requires some explanation: if your gauge has a value that ranges from 0 to 100 then the gauge has 101 gauge value units.

/tickmarks

- /**tickmarks** distance

Returns the distance (in slider value units) between consecutive tick marks. Returns 0 if there are no tickmarks.

/setvisualstate

state /**setvisualstate** -

Sets the visual state of the gauge. state is one of /Active or /Inactive.

/visualstate

- /**visualstate** state

Returns the visual state of the gauge.

Geometry

/minsize

- /**minsize** width height

Returns the minimum size of a gauge. The default is what is deemed the smallest reasonable size for a gauge.

/preferredsize

- /**preferredsize** width height

Returns a size that is large enough to display every whole number in the gauge's range.

/offset

name /**offset** x y

Returns the offset from the gauge's lower left corner, to the named point on the gauge. You may refer to the names during a calculated layout. Gauges know about the following four offset names: /MinEnd, /MaxEnd, /MinTick, and /MaxTick. For an explanation of how to use the offsets see Sliders, *Slider label positioning*, page 197.

ClassItemGroup

Figure 9-1 The ClassItemGroup subtree

ClassItemGroup is a utility class that is designed to reduce the number of clients that need to be added to a bag. Similar controls are managed as individual items in an item group. Controls implemented as groups include buttons, settings, and checkboxes. The group is responsible for arranging and painting its items on a in a rectangular area. State information is maintained by the group and appropriate event processing is also performed.

Creation

/new

placement parentcanvas /**new** instance

Subclasser method: /NewInit

Returns an instance of ClassItemGroup. In the NeWS Toolkit you generally instantiate subclasses of ClassItemGroup rather than the class itself. placement is one of: /Spaced, /Absolute, /Calculated, or /Grid. (See Setting the item list on page 88.)

/NewInit

placement parentcanvas /**NewInit** instance

Initializes an instance. Called by /**new**. Override it to perform any specific initialization on the instance. When it is overridden in subclasses, the method should do a super send so that the superclass can do its own initialization.

Items

When subclasses of ClassItemGroup are instantiated (sent the /**new** method) they get passed a placement parameter; this placement parameter determines how the group's items are laid out and what kind of layout data is given to /**setitemlist** (below).

The items in a group are descriptions, not objects. Each item is specified using display items. Internally, the item is stored as a PostScript dictionary which includes the following keys : /ItemX, /ItemY, /ItemWidth, and /ItemHeight. See *Item creation—strict subclass responsibility methods*, for an explanation.

Setting the item list

Items are referenced based on their insertion order. Each item can be accessed through a zero relative integer representing its position in the group. This integer is not a constant, because items can be inserted and/or deleted.

For /Absolute and /Calculated groups, layout data is required during the call to /**setitemlist**. However, for groups of /Calculated placement you can provide a default layout by using /**setlayoutparameters** (page 89). /Spaced an /Grid groups do not require layout data at setitemlist time. The different specifications of the layout data are in Table 9-1.

Table 9-1 Layout data required during calls to /setitemlist.

placement	**layout-data**
/Spaced	none required
/Absolute	x-y coordinate pair: [x y]
/Calculated	a compass point and a code fragment that the Toolkit executes whenever the item group is reshaped: [compass-point {calculated protocol }]
/Grid	none required. Define the grid using /**setlayoutparameters**.

/Spaced and /Grid placement

/setitemlist

[item1 item2 . . .] /**setitemlist** -

Sets the item list for groups using /Spaced or /Grid placement. /Spaced groups provide a minimal default layout scheme. /Grid groups have their layout data set using /**setlayoutparamters**.

Setting the layout for /Grid placement

You use /**setlayoutparameters** to define the shape of the grid in group's with /Grid placement.

/setlayoutparameters

[layout-by-rows? rows columns] /**setlayoutparameters** -

Sets the layout parameters for /Grid placement format. layout-by-rows? is a boolean; if true rows are filled with items before columns; if false, columns are filled before rows. rows is the number of rows in the item group; columns is the number of columns.

Absolute placement

/setitemlist

[item1 [x_1 y_1] item2 [x_2 y_2] . . .] /**setitemlist** -

Determines the set of items that are managed by an item group with absolute placement, and where they are to be placed.

Calculated placement

/setitemlist

[item1 [compass-point { calculated protocol }] . . .] /**setitemlist** -

Determines the set of items that are managed by an item group with calculated placement, and how they are to be laid out.

Setting a default layout for /Calculated placement

/setlayoutparameters

[compass-point {calculated-protocol}] /**setlayoutparameters** -

Sets the default placement format for /Calculated placement. If you use /**setlayoutparameters** for an itemgroup of placement-type /Calculated, you can use nullarray in place of the layout data normally associated with an item during a call to /**setitemlist**. Generally, you will want to provide layout data for the first item in the list and then have all subsequent items use the default. That is if you did:

```
[ /West { /East PREVIOUS POSITION } ] /setlayoutparameters myitemgroup send
```

Then, during a call to **/setitemlist** you could do:

```
[ item1 [ /West {/West PARENT POSITION} ]
 item2 nullarray item3 nullarray
] /setitemlist myitemgroup send
```

which would position the first item's west edge along the group's west edge and all subsequent items' west edges along the previous items' east edges.

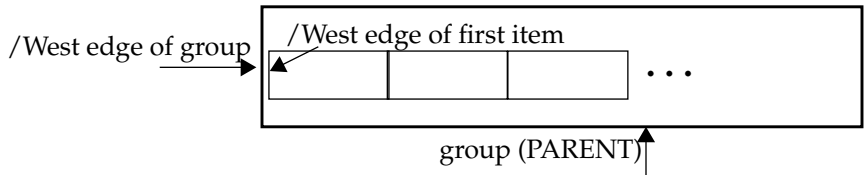


Figure 9-2 Example of items positioned using calculated layout.

/Calculated placement utilities

The calculated placement utilities are used as part of the calculated protocol for groups using calculated placement. They allow you to get references to individual items or the item group itself, and x,y position of items *while* placement is taking place. You can use these references and coordinates for positioning other items. For example the compass-point-calculated protocol pair:

```
[ /West {/West PARENT POSITION} ]
```

positions the center of item's west edge at the center of the west edge of the item's parent group.

CURRENT

- **CURRENT** current-item

Returns a reference to the current item that is being positioned. **CURRENT** can be passed as the item argument to other utilities such as **WIDTH**, **HEIGHT**, and **POSITION**.

HEIGHT

item | item-index **HEIGHT** height

Returns the height of the specified item.

PARENT

- **PARENT** item-group

Returns a reference to the parent region (self). **PARENT** can be passed as the item argument to the other utilities such as **WIDTH**, **HEIGHT**, and **POSITION**.

POSITION

compass-point item | item-index **POSITION** x y

Returns the x-y position for the specified compass-point on item-group.

PREVIOUS

- **PREVIOUS** item | item-index

Returns a reference to the previous item that was positioned. **PREVIOUS** can be passed as the item argument to other utilities such as **WIDTH**, **HEIGHT**, and **POSITION**. For the first item, **PREVIOUS**'s value is null.

WIDTH

item | item-index **WIDTH** width

Returns the width of the specified item.

Altering the item group

/appenditem

item layout-data /**appenditem** -

Adds item to the end of the item group. If the item group's placement is /Spaced or /Grid, layout-data is omitted. See Table 9-1 for the format of layout-data.

/deleteitem

item-index /**deleteitem** -

Deletes the item at item-index from the item group.

/insertitem

item-index item layout-data /**insertitem** -

Inserts item into the item group at item-index using layout-data to position the inserted item. If the item group's placement is /Spaced or /Grid, layout-data is omitted. See Table 9-1 for the format of layout-data.

/replaceitem

item-index item /**replaceitem** -

Replaces the item currently in the item group at item-index with item. The layout data of the replaced item is reused for the new item.

Querying the item group

/Item — item-index /**Item** item-dictionary

Returns the item dictionary created by /**NewItem**. /**Item** is an internal utility. See Item creation—strict subclass responsibility methods on page 97.

/item — item-index /**item** item-description

Returns the item description of the item at item-index. /**item** is a subclass responsibility method. The definition of item-description depends on the subclass of ClassItemGroup. For example a button's item description is the button's display item and notifier (if any); for menu buttons the item description is the button's display item and menu (if any).

/itemcount — - /**itemcount** number-of-items

Returns the number of items in the item group.

/itemlist — - /**itemlist** [item item. . .]

Returns an array of the items in the item group. No layout data for the items (if any) is returned by /**itemlist**.

Geometry and location of the items

/itembbox — item-index /**itembbox** item-x item-y item-width item-height

Returns the bounding box of the item at item-index.

/itemlocation — item-index /**itemlocation** item-x item-y

Returns the physical location of the item at item-index. item-x and item-y are relative to the group.

/itemsize — item-index /**itemsize** item-width item-height

Returns the size of the item at item-index.

/pointinitem? — x y item-index /**pointinitem?** boolean

Tells you whether the point specified by x,y is in the item at item-index.

/pointtoitem

x y /**pointtoitem** item-index true | false

Determines if the point specified by x,y is in any item. If the point is in an item then the item's index and true are returned. If x,y is not in any item, false is returned.

Geometry of the item group

/minsize

- /**minsize** width height

Returns the minimum size of the group. Calls /**?ValidateItemList**.

/preferredsize

- /**preferredsize** width height

Returns the group's preferred size. Calls /**?ValidateItemList**.

Painting items

/paintitem

item-index /**paintitem** -

Subclasser method: /PaintItem

Initiates painting for the item at item-index. /**paintitem** gets the item's item-dictionary and then calls /**PaintItem**.

/PaintItem

item-dictionary /**PaintItem** -

Paints the item represented by item-dictionary. This method is a strict subclass responsibility method.

Item size

/FixedItemSize?

/**FixedItemSize?** boolean (Variable)

Determines whether items in a subclass of ClassItemGroup are reshaped to the same size. If /**FixedItemSize?** is true, subclassers of ClassItemGroup are responsible for promoting /**ItemWidth** and /**ItemHeight**, which makes all items the same size.

/setfixeditemsize

boolean /**setfixeditemsize** -

Sets the value of /**FixedItemSize?**.

/fixeditemsize? - **/fixeditemsize** boolean

Returns the value of **/FixedItemSize?**.

Validation

ClassItemGroup provides these subclasser validation methods so you can implement efficient validation for groups of controls. For example, if you have a group of controls in a /Grid layout you may not have to lay out the grid when the item list changes. Another use for these methods is to avoid recalculating a group's minsize where it isn't necessary.

In general, if you do an operation that invalidates a control you are expected to call **/paint**. Further, if you're doing several operations, one or more of which invalidate a control, you should do a single **/paint** at the end of the sequence.

/ItemListValid? **/ItemListValid?** boolean (Variable)

Determines if the item list is valid. Set to false in **/insertitem**, **/replaceitem**, **/appenditem**, **/deleteitem**, and **/setitemlist**.

/ValidateItemList - **/ValidateItemList -**

A subclasser method to permit validation of the item list without executing the **/validate** method, which causes layout to occur. The default action is to simply set **/ItemListValid?** to true. **[need more doc here—maybe an example of how this is used. Pull from code?]**]

/?ValidateItemList - **/?ValidateItemList -**

Tests to see if the item list is valid. If the list is not valid **/?ValidateItemList** validates it. If the list is already valid nothing is done.

/?ValidateItemList is called during **/validate** and **/minsize**

/validate - **/validate** -

Validates the item list and calls your **/Layout** procedure. (See page 96.) Calls **/?ValidateItemList**.

Tracking and items

These methods are inherited from ClassRegion and overridden in ClassItemGroup. You do not have to provide definitions for these methods. However you do need to provide definitions for the methods in the Mouse tracking in items—subclass responsibility methods section on page 95.

/TrackStart

event **/TrackStart** -

Determines if the coordinates of event are in an item. If they are then **/ItemStart** is sent to the group.

/TrackStop

event **/TrackStop** -

Determines if the mouse was over an item when the button was released. If it was **/ItemStop** is sent to the group.

/TrackCancel

event **/TrackCancel** -

Sent when the user interrupts the action, e.g, presses the STOP key. If the mouse was in an item when the **/TrackCancel** is sent, the group is sent **ItemCancel**.

/TrackMotion

event **/TrackMotion** -

Determines where the mouse has moved in relation to the items by resolving the coordinates of event. Depending on where the mouse goes the following occurs:

- If the mouse moves within an entered item, **/ItemMotion** is sent to the group.
- If the mouse moves from one item to another, the group it left is sent **ItemCancel** and the group it entered is sent **/ItemStart**.
- If the mouse moves from an item to outside the group the group is sent **ItemCancel**.
- If the mouse moves from outside the group to inside an item, the group is sent **/ItemStart**.

Mouse tracking in items—subclass responsibility methods

/ItemStart

item-index **/ItemStart** -

The callback executed when the item is entered; i.e., the mouse is *in* an item when SELECT goes down, or *enters* an item with SELECT down.

/ItemStop — item-index /**ItemStop** -

The callback executed when an item is chosen; i.e., SELECT goes up with the mouse in an item.

/ItemMotion — item-index /**ItemMotion** -

The callback executed when the mouse moves inside an entered item.

ItemCancel — item-index **ItemCancel** -

The callback executed when the mouse leaves an item with SELECT down.

Layout

Positioning items

/setgaps — horizontal-gap vertical-gap /**setgaps** -

Sets the horizontal and vertical gaps between items being positioned in item groups using either /Spaced or /Grid placement. For other placement formats, the gaps are ignored.

/gaps — **-** /**gaps** horizontal-gap vertical-gap

Returns the horizontal and vertical gaps.

Other layout methods

/Layout — - /**Layout** -

During validation, the **/Layout** method positions the items based on the layout parameters, individual items' layout data and the item group's placement type.

/layoutparameters — - /**layoutparameters** layout-parameters

Returns the parameters set in the /**setlayoutparameters** method.

ClassLayout methods defined for ClassItemGroup

ClassItemGroup provides a suitable implementation for the methods listed in this section as required by ClassLayout. For an explanation of their functionality please see Chapter 11, ClassLayout. The names of the methods are: CellSize, List, Location, Move, ResolveReference, and Size.

Help facilites for item groups

In order for itemgroups to get help the canvas subclass on which they sit must be Helpable. Help is available for itemgroups on a group-wide and on a per item basis. See Chapter 5, ClassCanvas, *Help facilities* on page 63 for an explanation of the TNT help system and, more specifically the format for keyword-string.

/sethelpkeyword — keyword-string /**sethelpkeyword** -

Sets the help string for the group.

/helpkeyword — event /**helpkeyword** keyword-string

Returns the group's help string.

/setitemhelpkeyword — item-index keyword-string /**setitemhelpkeyword** -

Sets the help string for the item at item-index.

/itemhelpkeyword — item-index /**itemhelpkeyword** keyword-string

Returns the help string for the item at item-index.

Item creation—strict subclass responsibility methods

/NewItem — item-description /**NewItem** item-dictionary

Converts a item description into an item dictionary.

The items in a group are dictionaries that have the following keys:

Key	Value	Accessor method
/ItemX	the item's x location relative to the group	/**itemlocation** (page 92)
/ItemY	the item's y location relative to the group	/**itemlocation** (page 92)

/ItemWidth	the item's width	/**itemsize** (page 92)
/ItemHeight	the item's height	/**itemsize** (page 92)

Note – Subclassers should not access these item variables directly, rather you should use the accessor methods. Following this procedure protects you against changes in the internal structure of the toolkit.

Subclasses can promote ItemWidth and ItemHeight as instance variables when the items share a common width or height. For example, if you have items that are either all the same size or share a common width or height you don't need to put these common values in each instance. Rather you can store the values in the class.

ClassLabel

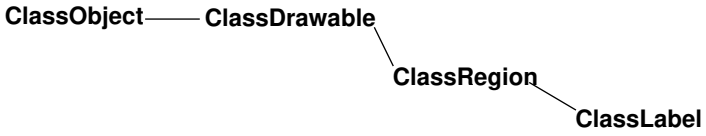


Figure 10-1 The ClassLabel subtree

A label is a region that manages a display item.

Labels override ClassRegion's text font to be the OPENLOOK default label font, which is /LucidaSans-Bold 12.

Creation

/new — displayitem parent /**new** -

A label is created by giving it a display item in addition to the parent used by ClassRegion

Geometry

/size — - /**size** width height

Returns the size of the display item, which is the size of the label. If the size is explicitly set using /**reshape**, that size is returned.

Label values

/setvalue — displayitem /**setvalue** -

Changes the label's display item

/value — - /**value** displayitem

Returns the label's display item

ClassLayout

ClassLayout is a mixin class, not intended to be directly instantiated and designed to ensure a consistent approach towards layout in panels and item groups. Panels manage groups of canvases and regions (see *ClassPanel* on page 131); item groups manage items that are defined as PostScript dictionaries (see *ClassItemGroup* on page 87).

Note – To avoid confusion in this chapter, both the canvases and regions managed by panels, and the dictionaries managed by item groups, are referred to as *elements*.

Placement

A placement parameter is passed to the ClassLayout subclass during its instantiation. The placement parameter determines how the elements of panels and itemgroups are positioned. The placements provided are: /Spaced, /Grid, /Absolute, and /Calculated.

With the exceptions of /Grid and /Spaced placement, layout information is expected by interfaces that manipulate the elements. Examples include the **/addclient** method in ClassPanel and the **/insertitem** method in ClassItemGroup.

Geometry

/minsize — - /**minsize** width height

Returns the minimum width and height required to layout the elements. When the placement format is /Calculated, the values returned might need to be adjusted, since this format is based on a heuristic.

Layout

The specification for the layout information is dependent on the placement parameter.

/Layout — - /**Layout** -

During validation, the **/Layout** method positions the elements based on the layout parameters, individual elements' layout data and the subclass's placement type. The appropriate context (e.g., setting the canvas) is established before **/Layout** is called.

Spaced placement

The /Spaced placement does not require information because it provides automatic and minimal layout. Elements are laid out from left to right and then top to bottom.

Grid placement

The /Grid placement does not require layout data because it positions elements based on three parameters set through the **/setlayoutparameters** method. These parameters determine the number of rows and columns and whether the rows or columns should be filled first. The maximum item height and width determines the cell size.

/Grid format defaults to having one row and as many columns as necessary to accomodate all the elements. To change this default use **/setlayoutparameters**.

/setlayoutparameters — [layout-by-rows? rows columns] /**setlayoutparameters** -

Sets the number of rows and columns and indicates whether rows are supposed to be filled before columns.

Absolute placement

The /Absolute placement allows you to position an element using an X-Y coordinate pair:

[X Y]

Calculated placement

/Calculated placement allows you to position an element using a code fragment that is executed whenever the layout subclass is reshaped.

/setlayoutparameters

[compass-point {calculated-protocol}] /**setlayoutparameters** -

Sets the layout parameters for objects that use calculated placement to position their elements. The compass-point notation allows elements to be positioned relative to a corner of another element or the container. compass-point must be defined as either /North, /East, /West, /South, /NorthEast, /NorthWest, /SouthEast, /SouthWest, or /Center.

{calculated-protocol} is a code fragment that is executed whenever a reshape is done. The fragment must return an x-y coordinate pair when executed.To aid in placement, six utilities have been defined: PARENT, WIDTH, HEIGHT, POSITION, CURRENT, and PREVIOUS. See *Calculated placement utilities*.

For example the following calculated parameters positions the west edge of the element being positioned, along the west edge of its parent:

```
[ /West {/West PARENT POSITION} /setlayoutparameters
```

Using /**setlayoutparameters** for calculated layouts allows you to specify a default layout. If you have a default calculated layout you can specify an empty array instead of associating layout data with each element. (In most cases you will want to supply layout data for the first element being positioned.) The empty array is detected during validation and your default is used to achieve positioning.

Calculated placement utilities

CURRENT

- **CURRENT** current-element

Returns a reference to the current element which is being positioned. **CURRENT** can be passed as the element argument to other utilities such as **WIDTH**, **HEIGHT**, and **POSITION**.

HEIGHT — element **HEIGHT** height

Returns the height of the specified element.

PARENT — - **PARENT** parent

Returns a reference to the parent container (self). **PARENT** can be passed as the element argument to other utilities such as **WIDTH**, **HEIGHT**, and **POSITION**.

POSITION — compass-point element **POSITION** X Y

Returns the X-Y position for the specified compass-point on element.

PREVIOUS — - **PREVIOUS** previous-element

Returns a reference to the previous element which was positioned. PREVIOUS can be passed as the element argument to other utilities such as **WIDTH**, **HEIGHT**, and **POSITION**.

WIDTH — element **WIDTH** width

Returns the width of the specified element.

Miscellaneous

/layoutparameters — - /**layoutparameters** layout-parameters

Returns the parameters set in the /**setlayoutparameters** method.

/setgaps — horizontal-gap vertical-gap /**setgaps** -

Sets the horizontal and vertical gaps between elements being positioned using either /Spaced or /Grid placement. For other placement formats, this method is ignored.

/gaps — **-** /**gaps** horizontal-gap vertical-gap

Returns the horizontal and vertical gaps.

Subclass responsibility procedures

During validation, ClassLayout requests a list of elements from the subclass. The elements in the list are positioned based on the placement parameter. To assist in layout, the subclasses are responsible for implementing the procedures in this section.

CellSize

- **CellSize** cell-width cell-height

Returns the size of a single element of a ClassLayout subclass that has been laid out using /Grid layout.

List

- **List** [element ...]

Returns an array of elements to be positioned during validation.

Move

x y element **Move** -

Moves the element to the specified x,y position.

Size

element **Size** width height

Returns the width and height for the specified element. In addition, **Size** is expected to return the appropriate answer for self.

Location

element **Location** X Y

Returns the X-Y origin for the specified element. In addition, **Location** is expected to return the appropriate answer for self.

ResolveOffset

layout-client compass-point | offset-name **ResolveOffset** x y

Returns the x,y coordinates (in the current CTM) of layout-client's compass-point or the location of the specified offset-name. compass-point is one of: /North, /East, /West, /South, /NorthWest, /SouthWest, /NorthEast, /SouthEast, or /Center. offset-name is a name accepted by the /**offset** method of the layout client. (Not all objects provide an /**offset** method. See Chapter 22, Sliders.) Both compass-point and offset-names are used for calculated layout.

ResolveReference

element-reference **ResolveReference** element

Returns the element associated with element-reference. For example, ClassPanel uses PostScript names to reference its clients. ClassButtons uses an integer index.

This method is used in the /Calculated placement protocol. When the element-reference is self, then self is returned.

Useful systemdict utilities

The utilities in this section are defined in systemdict, not ClassLayout. They are included here because they are useful for positioning elements of ClassLayout subclasses that use calculated layout.

xyadd — $x_1\ y_1\ x_2\ y_2$ **xyadd** $x_1+x_2\ y_1+y_2$

Does vector addition on the two sets of x,y coordinates.

xysub — $x_1\ y_1\ x_2\ y_2$ **xysub** $x_1-x_2\ y_1-y_2$

Does vector subtraction on the two sets of x,y coordinates.

xymax — $x_1\ y_1\ x_2\ y_2$ **xymax** xmax ymax

Returns the largest x and largest y of two sets of x,y coordinates.

xymin — $x_1\ y_1\ x_2\ y_2$ **xymin** xmin ymin

Returns the smallest x and the smallest y of two sets of x,y coordinates.

ClassMenu

Figure 12-1 The ClassMenu subtree

ClassMenu implements OPEN LOOK menus. In addition to explaining the methods that a menu responds to, this chapter also includes menu-related interfaces from ClassCanvas and ClassRegion.

Four types of menus are possible: /Command, and three settings-based menus, /Exclusive, /NonExclusive, and /ExclusiveVariation. Only /Command menus can have submenus. The "type" of menus refers to the kind of items a menu has and therefore the types of choices that can be made. Command menus have items that look and act like buttons (see Buttons on page 29). The settings-based menus look and act like settings (see ClassSettings and ClassCheckBoxes on page 189). You determine the type of menu using **/setchoicemode** (page 108).

Creation

/new

placement parent-canvas /**new** menu

Creates a menu that uses the given placement style. Placement determines how menu items are laid out. placement is one of /Spaced, /Absolute, /Calculated, or /Grid (see *Placement* on page 101 in ClassLayout).

Choice Modes

/setchoicemode

mode-name /**setchoicemode** -

Sets the choice mode for the menu. mode-name is one of: /Command (which allows a mixture of command and submenu items), /Exclusive, /NonExclusive, or /ExclusiveVariation. /Command is the default choice mode.

If you call /**setchoicemode** *after* having called /**setitemlist** or other methods that manipulate the item list (e.g., /**deleteitem**, /**insertitem**), the items (and layout info, if any) are copied over to the new mode. However, changing an existing menu between /Command and non-/Command modes is likely not to result in a working menu, because:

1. non-/Command menus don't have submenu items, which will confuse the non-/Command menus

and

2. The notifier callbacks for /Command and non-/Command menus expect slightly different arguments on the stack. (See /**setnotifier**, page 116.)

/choicemode

- /**choicemode** mode-name

Returns the menus mode. mode-name is one of: /Command (which allows a mixture of command and submenu items), /Exclusive, /NonExclusive, or /ExclusiveVariation. /Command is the default choice mode.

Menu items

For the methods in this section the argument item expands to:

string | [display-item <notifyname | notifyproc | submenu>^{optional}]

An item is either a simple string, or it is an array whose first entry is a display item, and whose *optional* second entry is one of the following three objects:

- the name of a notification method that is called in the target object when this item is chosen from the menu
- a procedure to execute when this item is chosen from the menu
- a submenu to bring up when the user pulls-right over this object. Use of the submenu argument is legal only for /Command menus.

Thus the following are all legal arguments to /setitemlist:

Example 1:

```
[ string string. . . ]
```

Example 2:

```
[ [displayitem1] [displayitem2] . . . ]
```

Example 3:

```
[
  [ displayitem1 /notifier1 ]
  [ displayitem2 /notifier2 ]
  [ displayitem3 submenu ]
  . . .
]
```

If no notifier is provided in the array then the menu-wide notifier (if any) is used when the user selects that item. Menu items are accessed via an item-index.

Setting the menu's item list

Menu items can be accessed through a zero relative integer (its index) representing its position in the menu. This integer is not a constant, because items can be inserted and/or deleted. The NeWS Toolkit does not check for indexes out of range.

For /Absolute and /Calculated placement menus, layout data is required during the call to /**setitemlist**. However, for menus of /Calculated placement you can provide a default layout by using /**setlayoutparameters** (page 111). /Spaced and /Grid groups do not require layout data at setitemlist time. The different specifications of the layout data are in Table 12-1.

Table 12-1 Layout data required during calls to /setitemlist.

placement	**layout-data**
/Spaced	none required
/Absolute	x-y coordinate pair: [x y]

/Calculated	a compass point and a code fragment that the Toolkit executes whenever the item group is reshaped: [compass-point {calculated protocol }]
/Grid	none required. Set the layout using /**setlayoutparameters**.

Spaced and Grid placements

Menus using /Spaced and /Grid placement do not pass layout data to /**setitemlist**. /Spaced layout provides automatic and limited layout. /Grid placement menus use /**setlayoutparameters** to describe the shape of the grid.

/setitemlist [item item] /**setitemlist** -

Sets the menu's item list. Invalidates the menu.

/setlayoutparameters [layout-by-rows? rows columns] /**setlayoutparameters** -

Sets the shape of the grid for /Grid placement format. layout-by-rows? is a boolean; if true rows are filled with clients before columns; if false, columns are filled before rows. Neither rows nor columns can be null.

Absolute placement

/setitemlist [item1 [x_1 y_1] item2 [x_2 y_2] . . .] /**setitemlist** -

Determines the set of items that are managed by a menu with absolute placement, and where they are to be placed. [x_n y_n] is the position where you want the lower-left corner of the item's bounding box placed. Invalidates the menu.

Calculated placement

/setitemlist [item [compasspoint {calculated protocol}]. . .] /**setitemlist** -

Sets the item list for a menu with calculated placement. [compasspoint {calculated protocol}] is the item's layout data and determines where the item is positioned in the menu. Invalidates the menu.

You can use /**setlayoutparameters** to provide a default calculated layout. (See *Setting a default layout for /Calculated placement*, below.) Generally, you will give the first item its own layout data and the default is used to place the other items. In this case you would pass an empty array as layout data for all items other than the first one:

```
[ item1  [compasspoint {calculated protocol} item2 nullarray item2 nullarray. . . ] /setitemlist
```

Setting a default layout for /Calculated placement

/setlayoutparameters

[compass-point {calculated-protocol}] /**setlayoutparameters** -

Sets the default layout for /Calculated placement. If you use /**setlayoutparameters** for a menu of placement-type /Calculated you can specify the layout data associated with an item as an empty array. That is if you did:

```
[ /West { /East PREVIOUS POSITION } ] /setlayoutparameters mymenu send
```

Then, during a call to /**insertitem**, for example, you could do:

```
item-index myitem nullarray /insertitem mymenu send
```

which would position myitem's west edge along the previous item's east edge. Each subsequent call to /**insertitem** (with an empty layout data array) would position the new client's west edge along the east edge of the previous item.

/Calculated placement utilities

The calculated placement utilities are used as part of the calculated protocol for groups using calculated placement. They allow you to get references to individual items, the menu itself, and the x,y positions of items *while* placement is taking place. You can use these references and coordinates for positioning other items. For example the compass-point-calculated protocol pair:

```
[ item1/West {/West PARENT POSITION} ]
```

positions item1's west edge along the west edge of item1's parent container.

PARENT

- **PARENT** parent

Returns a reference to the parent container (self). **PARENT** can be passed as client to other utilities such as **WIDTH**, **HEIGHT**, and **POSITION**.

CURRENT

- **CURRENT** current-client

Returns a reference to the current client that is being positioned. **CURRENT** can be passed as the client argument to other utilities such as **WIDTH**, **HEIGHT**, and **POSITION**.

PREVIOUS

- **PREVIOUS** previous-client

Returns a reference to the previous client that was positioned. **PREVIOUS** can be passed as the client argument to other utilities such as **WIDTH**, **HEIGHT**, and **POSITION**.

WIDTH

client **WIDTH** width

Returns the width of the specified client.

HEIGHT

client **HEIGHT** height

Returns the height of the specified client.

POSITION

compasspoint client **POSITION** X Y

Returns the X-Y position for the specified compasspoint on client.

Layout

/layoutparameters

- /**layoutparameters** layout-parameters

Returns the menu's layout parameters.

Other item methods

See Menu items on page 108 for a definition of item. If a method returns an item, the entire item is returned.

/appenditem

item layout-data /**appenditem** -

Adds item to the end of the menu's item list. Invalidates the menu. /Spaced and /Grid menus omit layout-data; calculated placement menus may specifiy layout-data as a null array if a default layout has been set with /**setlayoutparamters**.

/deleteitem

item-index /**deleteitem** -

Deletes the item at item-index. Invalidates the menu.

/insertitem

item-index item layout-data /**insertitem** -

Inserts item at item-index in the menu. Insertions go *before* any existing item of that index. /Spaced and /Grid menus omit layout-data; calculated placement menus may specifiy layout-data as a null array if a default layout has been set with /**setlayoutparamters**. Invalidates the menu.

/item

item-index /**item** item

Returns the item located at item-index.

/itemcount

- /**itemcount** item-count

Returns the number of items in the menu. (The pin is *not* considered an item.)

/itemlist

- /**itemlist** [item item ...]

Returns the items in the menu. The layout data is *not* returned.

/replaceitem

item-index item /**replaceitem** -

Replaces the item at item-index. The layout data, if any, of the old item at item-index is used to position the new item. Invalidates the menu.

Default item

A default item is visually highlighted, and determines where the pop-up menus are positioned on the screen relative to the mouse. Menu buttons use their associated menu's default item as their response to SELECT. TNT allows you to have a pop-up menu that has no default.

/setdefault

null | item-index /**setdefault** -

Sets the item at item-index to be the default item for the menu.

A default item is visually highlighted, and determines where the pop-up menus are positioned on the screen relative to the mouse. Menu buttons use their associated menu's default item as their response to SELECT. It *is* legal for a pop-up menu to have no default.

/default

- /**default** null | item-index

Returns the index of the menu's default item, or null if there is no default.

Labels

/setlabel — display-item | null /**setlabel** -

Sets the label shown at the top of the menu. null implies that there is no label, which is the default. Invalidates the menu.

/label — - /**label** display-item | null

Returns the label shown at the top of the menu or null if there is no label.

/setpinnedlabel — display-item | null /**setpinnedlabel** -

Sets the label for the pinned copy of the menu. OPEN LOOK requires that a pinned copy of a menu have a label; /**setpinnedlabel** is provided so you can give the pinned copy of your menu a label. If null, then the label of the original menu is used. If the original menu has no label, and you don't use /**setpinnedlabel** to provide on for the pinned copy, then the pinned copy has no label either. The default is null. /**setpinnedlabel** invalidates the menu.

/pinnedlabel — - /**pinnedlabel** display-item | null

Returns the label of pinned copy of the menu, or null if it has no label.

Pinned menus

TNT handles the interaction between a menu and its pinned copy so you don't have to make sure changes made to your menu are reflected in its pinned copy. When changes that cause invalidation (e.g., /**insertitem**) are made to a menu, you send /**paint** only to the menu. TNT ensures that the pinned copy also is updated to reflect the changes, including size, made to the menu. In general, you shouldn't send any messages to the pinned copy.

/Pinnable? — /**Pinnable?** boolean (Variable)

Determines whether the menu has a pin.

/setpinnable — boolean /**setpinnable** -

Sets the value of the variable /Pinnable?

/pinnable? — - /**pinnable?** boolean

Returns the value of the variable /Pinnable?

Programmatically pinning menus

The methods in this section are only defined for menus that are pinnable. They would only be called by applications that want to programmatically control the pinning of menus. (Users usually look after this.)

/pin — x y /**pin** -

Creates the pinned copy (if necessary), maps it, activates it, and positions it on the screen so that its northwest corner is at the given position. If the menu is already pinned the pinned window moves to the given location. The pinned menu is activated using the event manager of the canvas that most recently activated the original menu; if the original menu has never been popped up, or if that event manager has since gone away, an error will result. If the originating canvas is inside a window, the pinned menu is added as a subwindow of that window's root window.

/pinned? — - /**pinned?** boolean

Returns whether the menu has a pinned copy currently displayed.

/unpin — - /**unpin** -

Unmaps and deactivates the pinned menu.

Ability to interact with menu items

/setvisualstate — index state /**setvisualstate** -

Sets the visual presentation of the menu item at index. state can be one of /Active, /Inactive, or /Busy. /Active is the default. /**setvisualstate** also determines whether users can interact with the menu. Users can interact with /Active items; /Inactive and /Busy items ignore user interactions.

/visualstate — **-** /**visualstate** index statename

Returns the state of the item at index.

Notification and previewing

When menu notification occurs the *notification value* and the menu instance are placed on the stack before the notifier is called. The notification value varies with the type of menu:

- For /Command menus the notification value is the index of the item where the mouse button was released. The index and the menu instance are put on the stack before the notifier is called. If the notifier is a proc then the index, menu instance and proc are put on the stack and the proc is executed.
- For the settings-based menus (/Exclusive, /NonExclusive and /ExclusiveVariation) the notification value is an array:

  [item-index boolean]

Different types of settings notify at different user actions:

/Exclusive	notifies *only* when an item is turned on. This means you are not notified about the item being unchosen, and the boolean in the array is always true.
/ExclusiveVariation	notifies when an item is turned on as /Exclusive settings do and notifies when the item currently turned on is turned off and the setting has no chosen item.
/NonExclusive	notifies when any item is turned on or off.

The array and the menu instance are put on the stack before the notifier is called. If the notifier is a proc then the array, menu instance and proc are put on the stack and then the proc is executed.

/setnotifier — notifier /**setnotifier** -

Sets the notifier for the entire menu. The notifier can be either a PostScript name or a procedure. The notifier determines what action takes place when an item that has no individual notification specification is selected. (An item with no notifier is specified using the form string or [displayitem].) Your notifier should be written to use the notification value and the menu instance as arguments.

/notifier — - /**notifier** notifier

Returns the single notifier that is associated with the entire menu.

Menu targets

The NeWS Toolkit has the notion of a default object that is affected by a menu's notifier. If you want to retarget the notifier use the target mechanism (see *Target interface inherited from ClassControl* on page 117).

The invoker mechanism

Menus remember which object invoked them.The invoker of a menu is the object that most recently caused the menu to be displayed. So, for a popup menu the invoker is a canvas, for a menu-button menu the invoker is the button group, and for a submenu (pullright) the invoker is the parent menu.

If you choose not to call **/settarget** yourself then the following rules apply:

- The target of a popup-menu is its invoker (i.e. the canvas or region from which it came).
- The target of a submenu is the target of its invoker. (In this case the invoker is a menu. The invoker menu may have an application-specified target, or it too might default via these rules.)
- The target of a menu button's menu is the target of its invoker. Its invoker is the menu button instance, and because menu buttons are controls, they can have targets.

The invoker is available to programmers via the **/invoker** method. Generally you don't use **/invoker** yourself but if you wanted to find out the chain of menus that resulted in a callback to be executed you just keep walking backwards down the invoker chain until you come to an object that isn't a menu.

/invoker - **/invoker** object

Returns the object that invoked the menu.

Target interface inherited from ClassControl

/settarget object | null **/settarget** -

Sets object to be the target of the menu. If null is specified the target is cleared.

/cleartarget null | object **/cleartarget** -

Clears the target. If object is specified the target is cleared only if the target and object are the same.

/target - **/target** null | object

Returns the target object. If /settarget has not been called (or /cleartarget has been called), a default target is determined using the Invoker mechanism.

/sendtarget — args /method /**sendtarget** results

Sends /method and any necessary arguments to the target and returns the method's results, if any.

Menu Values

/setvalue — [index index ...] /**setvalue** -

Sets the value of the menu. /**setvalue** can be used to initialize a settings menu to a specific state: the items corresponding to the indices provided will be turned on—the rest will be off. The behavior is undefined if an /Exclusive menu is given an array of other than a single index, or an /ExclusiveVariation menu is given an array of other than 0 or 1 indices. Command menus are not stateful, and hence have no real need for this method. (It can be legally used however, and has the effect of changing the menu's notion of which command item was last selected. An array containg a single index should be used in this case.)

/value — - /**value** [index index ...]

Returns the value of the menu. For settings menus (/Exclusive, /NonExclusive, and /ExclusiveVariation), /**value** returns an array of indices (possibly empty) corresponding to the currently "on" items. For /Command menus an array containing a single index corresponding to the item last selected is returned.

ClassCanvas and ClassRegion interfaces

Canvas menu methods

Instances of ClassCanvas do not respond to MENU by default. They will if made "menuable." Regions sitting on menuable canvases simply make sure that their menu is not null before showing it when the user presses MENU over them.

/Menu — /**Menu** menu-instance (Variable)

The menu for a canvas or region. Subclass it to associate a menu with all instances of this type of canvas or region.

/setmenu

menu | null /**setmenu** -

Associates a menu with this Canvas or Region. Calling this method alone does not ensure that the menu will be shown when the user presses MENU. In order for the menu to appear the canvas must be made menuable.

/menu

- /**menu** menu | null

Returns the menu (if any) associated with this canvas or region.

/Menuable?

/**Menuable?** boolean (Variable) (only in ClassCanvas)

Determines whether a canvas should show its menu when MENU is pressed with the pointer over the canvas. When /Menuable? is true the canvas receives a /**MenuStart** when MENU is pressed over it (see /**MenuStart** on page 119).

/setmenuable

boolean /**setmenuable** - (only in ClassCanvas)

Sets the value of /**Menuable?**.

/menuable?

- /**menuable?** boolean (only in ClassCanvas)

Returns the value of the variable /Menuable?.

/MenuStart

invoker posname event /**MenuStart** invoker posname event menu true
| invoker posname event false
| invoker posname event null true

Determines which menu (if any) should be shown over this object, and where it should be positioned. You can change where the menu pops up by:

- Overriding /**MenuStart** and modifying the XLocation, YLocation pair of the event;

and/or

- Changing the posname.

posname defaults to /Default, which means that the default item (or the item at index 0 if there is no default) is aligned beside the given point. Currently, the only other legal posname is /NorthWest, which corresponds to the top-left corner of the menu. (posname is used by menu buttons for positioning their menus.)

invoker is the object that most recently caused the menu to be displayed. The invoker also determines the local event manager in which the menu's notifier executes and is the default target of the menu (see *The invoker mechanism* on page 117). /**MenuStart** can return a different invoker to change teh event manager and target.

By default /**MenuStart** returns the /Menu class/instance variable, and does not modify the positioning or invoker arguments.

Returning invoker posname event false causes event to be redistributed allowing some other object to open a menu.

The invoker posname event null true return is provided so that you can respond to MENU down over your canvas or region by not opening any menu and preventing anyone else (e.g., the window under the canvas or region) from opening a menu in response to the MENU.

This method is executed in the local event manager.

/MenuStop

menu /**MenuStop** -

Called when the menu (the root of the chain of submenus if such a chain is up) is brought down. You can subclass it if you need to perform any action at this time, such as destroying a temporary menu. /**MenuStop** executes in the local event manager.

Help facilites for menus

Help is available for menus on a menu-wide and on a per item basis. See Chapter 5, ClassCanvas, *Help facilities* on page 63 for an explanation of the the TNT help facilities and, more specifically, the format for keyword-string.

/sethelpkeyword

keyword-string /**sethelpkeyword** -

Sets the help string for the group.

/helpkeyword

event /**helpkeyword** keyword-string

Returns the group's help string.

/setitemhelpkeyword

item-index keyword-string /**setitemhelpkeyword** -

Sets the help string for the item at item-index.

/itemhelpkeyword

item-index /**itemhelpkeyword** keyword-string

Returns the help string for the item at item-index.

ClassNotice

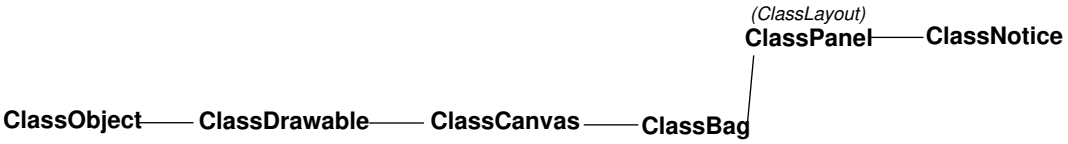


Figure 13-1 The ClassNotice subtree

Notices are panels that expect user input and usually freeze the application that generated them.

Creation

/new — base-window | null parentcanvas /**new** notice

Creates an instance of ClassNotice. If base-window is given (null is an acceptable value) it is the base window of the application that is blocked as a result of this notice. base-window should be the base window of the application so all the subwindows are also frozen.

Setting the frozen application

/setbasewindow — base-window | null /**setbasewindow** -

Sets the base window of the application that is frozen as a result of this notice.

/basewindow

- /**basewindow** base-window

Returns the base window that is frozen as a result of this notice.

Text in notices

/settext

string | [display-item display-item. . .] /**settext** -

Sets the text for the notice. If the argument is a single string it is broken up into as many strings as necessary to fit horizontally in the notice. If an array is passed, each display item in the array appears on one line, and the notice is made big enough to fit the longest one.

/text

- /**text** string | [display-item display-item . . .]

Returns the text of the notice.

Buttons in notices

/setbuttons

buttongroup | null /**setbuttons** -

Sets the buttons for the notice. The buttons should be laid out horizontally. The notice will add the button group as it client, centered. The buttons should have a default choice specified. If there isn't one, the notice just warps the cursor to some place in the notice. If null is specified the existing buttons are removed with /**removeclient**.

Note – The notifier for the buttons should send /**close** to the button group's parent to dismiss the notice. The notice does not go away automatically. For example:

```
{  % index buttongroup => -
   /close /Parent 2 index send send
}
```

/buttons

- /**buttons** buttongroup

Returns the notice's buttons.

Invoking a notice

/open

<apex> /**open** -

Invokes a notice. When invoked a notice:

- maps and paints itself
- warps the cursor to the default button
- takes the input focus
- freezes the target application until it is unmapped

<apex> determines the point from which the notice tail emanates. *The OPEN LOOK Graphical User Interface Function Specification* calls this point the apex. <apex> is used to calculate the x,y coordinates of the "emanation point" and is one of the following:

[x y] | event | canvas-instance | region-instance | [index itemgroup] | null

[x y]	You can simply furnish the point; it is CTM relative.
event	The coordinates are extracted from the event.
canvas or region	The notice apex is in the center of a region or canvas.
[item-index itemgroup]	The notice apex is at the center of the item's bounding box. (Remember, when notification occurs for an item group the item-index of the item selected and the item group are pushed on the stack.)
null	The notice does not display a tail and appears centered over the base window, or the framebuffer if the base window is null.

/close

- /**close** -

Takes the notice down, warps the cursor back to its original position, restores the input focus and unfreezes the base window.

ClassNumericField

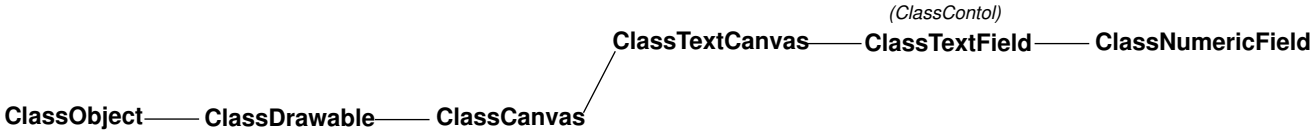


Figure 14-1 The ClassNumericField subtree

A numeric field is a control that displays numbers. Two arrows allow a client to increase and decrease the field's value by discrete stepping.

NumericField values

/setvalue — number /**setvalue** -

Sets the value of the field. Causes the field to be repainted.

/value — - /**value** number

Returns the value of the numeric field.

/setrange — min max /**setrange** -

Sets the minimum and maximum values permitted by the numeric field. When the value is changed by /**setvalue** or via the increment/decrement buttons (or the /**increment** or /**decrement** methods), or when the notifier is being called, the

value is first constrained to lie within the given range. (If min > max, the results are undefined.) The default min/max settings restrict the field to 32-bit integers, i.e., -2147483648 to +2147483647.

/range — - /**range** min max

Returns the minimum and maximum values permitted by the numeric field.

/setdelta — delta /**setdelta** -

Sets the amount by which the value is changed by a single click of the increment/decrement buttons. The field is updated if it is valid.

If the delta is set to zero, the increment/decrement buttons disappear from the field, though space is still reserved for them. (I.e., the numeric baseline is not extended to fill the space.) If the delta is set negative then the increment button decreases the value of the field and the decrement button increases the field's value.

/delta — - /**delta** delta

Returns the amount by which the value is changed by a single click of the increment/decrement buttons.

/increment — - /**increment** -

Adds the delta to the value of the field. After the delta is added the new value is checked against the field's range and adjusted if it is not in range. The new value is painted, but the increment/decrement buttons on the screen are *not* affected (i.e., they do not momentarily invert). /**increment** does not cause notification.

/decrement — - /**decrement** -

Subtracts the delta from the value of the field. The new value is painted, but the increment/decrement buttons on the screen are *not* affected (i.e., they do not momentarily invert). /**decrement** does not cause notification.

NumericField granularity

Numeric field granularity is controlled by a normalizer procedure. The normalizer procedure consumes the unconstrained value on the stack and leaves a constrained value on the stack.

/Normalizer

/Normalizer proc (Variable)

A normalizer is applied to constrain the value to some legal set. By default, the value is not constrained. A typical normalizer would be {round cvi} to constrain the value to integers.

The normalizer is applied after **/setvalue**, **/increment**, or **/decrement** (including when called due to clicking on the increment/decrement buttons), and before calling the notifier. The min/max are applied after normalizing, even if this results in a value that does not meet the normalization constraints. (E.g., consider a field where the normalizer is {round cvi} and the min/max range is 1.1 to 1.2.)

/setnormalizer

proc **/setnormalizer** -

Sets the field's normalizer.

/normalizer

- **/normalizer** proc

Returns the field's normalizer procedure.

Notification and previewing

The value of a numeric field is a number; thus the value placed on the stack before the notifier is called is a number. However, the previewer, if a client chooses to provide one, still operates on individual characters.

/setnotifier

notifier **/setnotifier** -

Sets the numeric field's notifier. notifier is either a method name that is sent to the target (as set by **/settarget**), or it is a PostScript code fragment. (The latter is not recommended but is permitted for those clients who need the generality or efficiency.) See *Notification and previewing* on page 77 in ClassControl.

/notifier

- **/notifier** notifier

Returns the numeric field's notifier.

/setpreviewer

previewer **/setpreviewer** -

Sets the previewer. Like the notifier (above), the previewer is either a method name that is sent to the target, or it is a PostScript code fragment. It can also be null to obtain default behavior. Setting the previewer allows a client to respond to each character as it is typed into the numeric field, unlike the notifier which is called only when the user is finished.

/previewer

- /**previewer** previewer

Returns the numeric field's previewer.

Inherited Methods

The following methods are inherited from ClassTextField (or ClassControl) and perform the same function as in text fields:

/new	/setcaret	/setselection
	/caret	/selection
/setvisualstate		
/visualstate	/PaintText	/scroll
		/Scroll
/setreadonly	/insertcharacter	
/readonly?	/insertstring	/fitcaret
/ReadOnly?	/InsertString	/FitCaret
		/AutoScrollPosition
	/deletecharacters	
	/deletespan	/InvisibleCaret
	/DeleteSpan	/VisibleCaret
		/CaretDelay
	/setnextfocus	
/movebaseline	/nextfocus	/ResolveToChar
	/previousfocus	
/setminimumvisible	/gotonextfocus	/SpecialActions
/minimumvisible	/gotonextfield	
/MinimumVisible	/gotopreviousfield	

A few additional methods are inherited unchanged from text fields, even though their application to numeric values is perhaps less common:

/characters (yields number of characters in the text)

/deletewords

/AlphaNumeric?

ClassObject

ClassObject is the root of the NeWS class hierarchy and therefore The NeWS Toolkits's class tree. In general ClassObject contains methods and operators that form some of the basis for NeWS. These operators are defined and discussed in the *NeWS 2.1 Programmer's Guide*. There are three exceptions: **/Properties, /setproperty, and /property**. These methods are discussed in this chapter.

/Properties

/Properties dictionary (Variable)

A dictionary used to store application-specific properties. These properties are called "client data" in other systems. During sends ClassObject is always on the dictionary stack and therefore the properties dictionary is always accessible during sends. However, care should be taken so that each instance's properties have unique keys to avoid name collisions. In addition these keys should be widely known so other applications can avoid colliding with your names.

/setproperty

key value **/setproperty** -

Adds the specified key and its associated value to the properties dictionary.

/property

key **/property** value

Returns the value associated with key in the properties dictionary.

ClassPanel

16

Figure 16-1 The ClassPanel subtree

ClassPanel is designed to be a control surface that contains canvas and region controls. A placement format is chosen when a panel is created (see **/new** below). This format determines how clients are positioned in the panel. In some cases, layout information is required when the client is added to the panel.

Creation

/new

placement parent-canvas /**new** instance

Returns a ClassPanel instance. placement indicates the placement format for the panel clients. It must be defined as either /Spaced, /Absolute, /Calculated, or /Grid:

/Spaced placement provides automatic and limited layout.

/Absolute placement allows you to specify the position of a panel client using an x-y coordinate pair, specified as an array: [x y].

/Calculated placement allows you to position a panel client using a code fragment that is executed whenever the panel is reshaped. The fragment must return an x,y coordinate pair when executed. In addition to the code fragment you may specify a compass-point that allows clients to be positioned relative to a corner of another client or the panel. Thus the /Calculated placement specification looks like:

```
[ compass-point { calculated-protocol } ]
```

compass-point is one of /North, /East, /West, /South, /NorthEast, /NorthWest, /SouthEast, /SouthWest, or /Center.

To aid in placement, six utilities have been defined, including PARENT, WIDTH, HEIGHT, POSITION, CURRENT, and PREVIOUS. These utilities are described in */Calculated placement utilities* below.

/Grid placement allows you to position clients based on the number of rows and columns you want as well as whether you want rows or columns filled first. The maximum client height and width determines the grid's cell size. The grid's three parameters, number of rows, number of columns and the order that the rows and columns are filled are set using **/setlayoutparameters** (page 133).

Panel clients

Adding panel clients

When you are adding a client to a panel (of all formats) one of the arguments is a name that is used to reference that client. This name *must* be unique. If you do not need handles for your panel clients you can use "dup" as the client's unique name. I.e., you could do:

```
<create a canvas client, e.g., mycanvasclient>
dup /addclient mypanel send
```

Not only does this ensure that each client has a unique handle, it also lets you use the client itself as the argument to **/removeclient**:

```
myclientcanvas /removeclient mypanel send
```

Spaced and Grid placements

/addclient

name client /**addclient** -

Adds client to the list of clients managed in a panel of either /Spaced or /Grid placement. client is an instance of ClassRegion, ClassCanvas, or one of their subclasses. name is a PostScript name used to reference client.

No layout data is required for /Spaced and Grid formats during /**addclient**. For the /Grid format you set the layout parameters of the panel as a whole using /**setlayoutparameters** (page 133). When the client has not been reshaped, /**addclient** reshapes it to its preferred size.

Setting the layout for /Grid placement

/setlayoutparameters

[layout-by-rows? rows columns] /**setlayoutparameters** -

Sets the layout parameters for /Grid placement format. layout-by-rows? is a boolean; if true rows are filled with clients before columns; if false, columns are filled before rows.

rows is the number of rows and columns is the number of columns.

Absolute placement

/addclient

name client [x y] /**addclient** -

Adds client to the list of clients managed in a panel of absolute placement. client is an instance of ClassRegion, ClassCanvas, or one of their subclasses. name is a PostScript name used to reference client. [x y] is the position that you want the lower left corner of client's bounding box placed. When the client has not been reshaped, /**addclient** reshapes it to its preferredsize.

Calculated placement

/addclient

name client [compass-point {calculated protocol}] /**addclient** -

Adds client to the list of clients managed in a panel with calculated placement. client is an instance of ClassRegion, ClassCanvas, or one of their subclasses. name is used to reference client and is any valid dictionary key, i.e., any non-null value.

[compass-point {calculated protocol}] determines where client is positioned in the panel.

For example, to position an object's west edge along the previous objects east edge you could do the following:

```
/clientname yourclient [ /West { /East PREVIOUS POSITION } ] /addclient yourpanel send
```

When the client has not been reshaped, /**addclient** reshapes it to its /**preferredsize**.

Setting a default for /Calculated placement

/setlayoutparameters [compass-point {calculated-protocol }] /**setlayoutparameters** -

Sets the default placement for /Calculated panels. If you use /**setlayoutparameters** for a panel of placement-type /Calculated you can specify the layout data, normally associated with a client during a call to /**addclient**, as a nullarray.

Generally, you will want to provide layout data for the first client you are adding to the panel and then have all subsequent panels use the default. That is if you did:

```
[ /West { /East PREVIOUS POSITION } ] /setlayoutparameters mypanel send
```

Then, during a call to /**addclient** for the first client you could do:

```
/myclient1 myclient1 [ /West {/West PARENT POSITION} ]
/addclient mypanel send
```

which would position the first clients's west edge along the panels's west edge and all subsequent client's west edges along the previous clients' east edges.

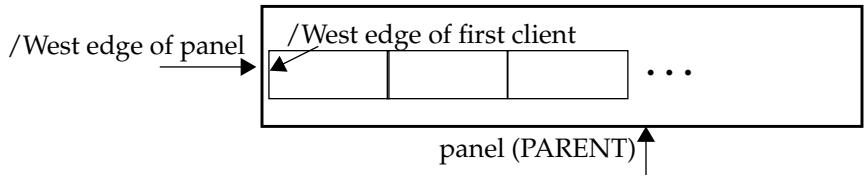


Figure 16-2 Example of clients positioned using calculated layout.

Then, during all subsequent calls to /**addclient** you could do:

```
/myclient2 myclient2 nullarray /addclient mypanel send
```

which would position myclient1's west edge along the previous client's east edge. Each subsequent call to /**addclient** (with an empty layout-data array) would position the new client's west edge along the east edge of the previous client.

/Calculated placement utilities

CURRENT

- **CURRENT** current-client

Returns a reference to the current client that is being positioned. **CURRENT** can be passed as the client argument to other utilities such as **WIDTH**, **HEIGHT**, and **POSITION**.

HEIGHT

client **HEIGHT** height

Returns the height of the specified client.

PARENT

- **PARENT** parent

Returns a reference to the parent panel (self). **PARENT** can be passed as client to other utilities such as **WIDTH**, **HEIGHT**, and **POSITION**.

POSITION

compass-point client **POSITION** X Y

Returns the X-Y position for the specified compass-point on client.

PREVIOUS

- **PREVIOUS** previous-client

Returns a reference to the previous client that was positioned. PREVIOUS can be passed as the client argument to other utilities such as **WIDTH**, **HEIGHT**, and **POSITION**. For the first client being added to a panel PREVIOUS returns null.

WIDTH

client **WIDTH** width

Returns the width of the specified client.

Removing clients

/removeclient

name /**removeclient** old-client true | false

Removes the client associated with name and invalidates the panel. The client instance and true are returned if the named client exists. Otherwise, false is returned. No layout data is returned.

If old-client is a canvas and has the *same* event manager as the panel, /**removeclient** deactivates it. If the client had its own event manager when it was added to the panel, it remains active when it is removed.

Positioning clients

/setgaps — horizontal-gap vertical-gap /**setgaps** -

Sets the horizontal and vertical gaps between clients being positioned using either /Spaced or /Grid placement. For other placement formats, this method is ignored.

/gaps — **-** /**gaps** horizontal-gap vertical-gap

Returns the horizontal and vertical gaps.

Layout

/layoutparameters — - /**layoutparameters** layout-parameters

Returns the parameters set in the /**setlayoutparameters** method.

/Layout — - /**Layout** -

During validation, the /**Layout** method positions the clients in the supplied list based on the Placement variable.

ClassLayout methods defined for ClassPanel

ClassPanel provides a suitable implementation for the methods listed in this section as required by ClassLayout. For an explanation of their functionality please see Chapter 11, ClassLayout. The names of the methods are: CellSize, List, Location, Move, ResolveOffset, ResolveReference, and Size.

Menus and tracking

By default panels are Menuable and Trackable (see ClassCanvas on page 35).

ClassRegion

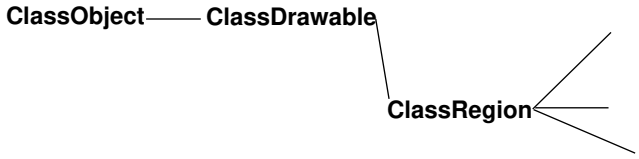


Figure 17-1 The ClassRegion subtree

Regions are the Toolkit's lightweight canvas replacement but there are some differences between canvases and regions:

- Regions can only be rectangular in shape.
- Regions cannot be made Selectable (See *Selectables* on page 65 in ClassCanvas.)
- Regions have parents, but they don't have children.
- Regions generally are required to be clients of a Bag (e.g., border bags and panels) to be useful. Bags provide the canvas "glue" to capture events and forward them to the appropriate region client.

Regions support the menu and tracking interfaces found in ClassCanvas. The Toolkit uses regions to create such objects as buttons and settings.

Creation

/new — parentcanvas /**new** instance

Creates a new region instance.

Region appearance

Colors

The NeWS Toolkit's OPEN LOOK components are drawn in a "3-D" style using five different brightness values to represent light and dark shading of beveled edges (Table 17-1). (For a complete explanation of how these shadings are used to display 3-D objects see the *OPEN LOOK Graphical User Interface Functional Specifications*.)

However, the 3-D effect is unusable on a black and white display when gray values are approximated by stipple patterns. In order to make the OPEN LOOK components usable in two colors (i.e. black and white), the Toolkit provides 2-D colors and interfaces to switch between 3-D and 2-D looks.

/setcolors — foreground-color background-color /**setcolors** -

Sets the foreground and background colors for the region. /**setcolors** examines the /**3D?** variable (see *Switching between 3-D and 2-D looks*) to determine whether to set /FG and /BG (3-D colors) or /2DFG and /2DBG (2-D colors). If /**3D?** is true, /**setcolors** also computes and sets BG0, BG2, and BG3, based on the value of background-color. See Table 17-1 for an explanation of these colors. If the instance is valid, it is repainted immediately using the new colors. If the instance is invalid the colors are reset but the instance is not painted automatically. See *Validation* for more information.

Send /**setcolors** to a class to change the class defaults or an instance to change just the instance.

/colors — - /**colors** foreground-color background-color

Returns the foreground and background colors for the region. /**colors** examines the value of /**3D?** to determine whether to return /FG and /BG or /2DFG and /2DBG. Values returned are the NeWS color objects.

/BackgroundColor - /**BackgroundColor** background-color

Returns BG or 2DBG, according to the dimensionality. /**BackgroundColor** checks to see what the dimensional state of the framebuffer is before returning a color. Thus you should use /**BackgroundColor** instead of /BG so your application will use the appropriate 2-D or 3-D color.

/ForegroundColor - /**ForegroundColor** foreground-color

Returns FG or 2DFG, according to the dimensionality. /**ForegroundColor** checks to see what the dimensional state of the framebuffer is before returning a color. Thus you should use /**ForegroundColor** instead of /FG so your application will use the appropriate 2-D or 3-D color.

Region color class variables

TNT uses slightly different names for its 3-D colors than are found in the *OPEN LOOK Graphical User Interface Functional Specifications*. Table 17-1 shows how the TNT color variables map into the OPEN LOOK names as well as the names of the 2-D class variables.

Table 17-1 Toolkit color variables and their OPEN LOOK names

TNT Class Variable	OPEN LOOK name	Explanation
/FG	Foreground	Foreground color. Used for text and the border of the region. Default is black.
/BG0	Background	Background color 0. Slightly darker than white, the OPEN LOOK color that it replaces.
/BG	BG1	Background color 1. Slightly darker than bg0. Replaces the OPEN LOOK color, BG1. Used for the background of the region.
/BG2	BG2	Background color 2. Slightly darker than bg. Used as the background of “indented” choices.
/BG3	BG3	Background color 3. Used for the shadow of 3-D objects.
/2DFG	Black	Used for the foreground of 2-D regions. Defaults to black.
/2DBG	White	Used for the foreground of 2-D regions. Defaults to white.

Switching between 3-D and 2-D looks

/3D? **/3D?** boolean (Variable)

Determines whether the region is drawn in 3-D or 2-D (i.e., the region's dimensionality). The default is true—use 3-D colors. 3-D regions are drawn using the 5 colors, FG BG BG0 BG2 and BG3 . 2-D regions are drawn using the 2 colors, 2DFG and 2DBG. If you the Toolkit to be in 2-D mode when it is first loaded you can set the dimensionality by doing a /3D? false put in UserProfile.

/set3D boolean **/set3D** -

Sets the dimensionality of the region. **/set3D** does not cause the region to repaint, but it does invalidate instances of the following region subclasses (because they have cached drawing information that needs to be updated): ClassButtons and ClassMenuButtons.

Painting

TNT uses the following rendering model for painting regions: **/paint** initiates the painting recursion by making the region's canvas parent the current canvas and then calling **/PaintAll**. **/PaintAll** sets up the context of the current object *relative to t*hat of the parent. For regions the context is established by translating the coordinate system so 0,0 is at the lower left corner of the affected region. **/PaintAll** also validates the object. (See *Validation* on page 44.) **/PaintAll** calls **/Paint** and **/PaintChildren**. **/Paint** paints the object itself. Clients should override **/Paint** (unless the default **/Paint** method is suitable, which is rare). **/PaintChildren** sends **/PaintAll** to each Mapped child.

/paint - **/paint** -

Sent to a region to initiate painting. **/paint** sets up the context for the painting by making the region's canvas parent the current canvas. Calls **/PaintAll**.

/PaintAll - **/PaintAll** -

Subclasser method: /Paint
/PaintChildren

Paints the entire region. You should neither call nor override **/PaintAll**. Checks to see if the region is valid and if it isn't, validates it. Uses the **/Paint** and **/PaintChildren** methods for painting. Subclassers should override **/Paint** and/or **/PaintChildren** to change the default painting behavior.

/Paint — **- /Paint -**

Paints the interior of a region. You can assume that **/Paint** is operating inside a gsave/grestore and with the region's canvas parent set as the current canvas. Moreover, you also can assume that a translate has been done so that the lower left corner of the region is located at (0,0). The default painting done is to fill the region with its background color; subclassers should override **/Paint** to perform their own region painting.

/PaintChildren — **- /PaintChildren -**

A no-op for ClassRegion.

Utility Painting Methods

The methods in this section are utility methods and are provided because of their usefulness. These are *not* subclasser methods. The methods should be called from within gsaves and grestores and the region's canvas parent must be set. If you're using these methods inside your **/Paint** definition you don't have to set the canvas yourself, TNT does this for you. See the rendering model discussion above.

/BotRightPath — x y width height **/BotRightPath -**

Constructs a path of the bottom and right edges of the box defined by x y width and height.

/Paint3DBox — x y width height down? **/Paint3DBox -**

Paints a 3-D box using **/BG*** colors whose bounding box is defined by the arguments. When down? is true the box appears recessed. When down? is false the box appears raised.

/Paint3DLine — x y width **/Paint3DLine -**

Draws a 3-D line with a 1-point light line above a 1-point dark line having endcaps either all light or all dark.

/Paint2DBox — x y w h bold? **/Paint2DBox -**

Paints a 2-D box using the 2DFG and 2DBG colors, whose bounding box is defined by the arguments. When bold? is true the box appears stroked with a double thick line.

/TopLeftPath

x y width height /**TopLeftPath** -

Constructs a path of the top and left edges of the box defined by x y width and height.

Fonts

/TextFont

/**TextFont** font

The default font used for the region. The default value is /LucidaSans 12 point. Setting this variable does not automatically make font the current font. In order to make font the region's current font you could do:

```
TextFont setfont
```

/settextfont

font /**settextfont** -

Subclasser method: /ModifyFont

Defines the text font for a region. /**settextfont** can be sent to a class or an instance. For example to change the region's text font you would do something like:

```
/TimesRoman findfont 12 scalefont /settextfont yourcanvas send
```

As with /**TextFont**, the code above doesn't make font the current font. In order to have the PostScript show operator use the text font sent in the example you would have to make the font the current font using the PostScript **setfont** operator.

/textfont

- /**textfont** font

Returns the text font of the region.

/ModifyFont

font /**ModifyFont** font'

Returns a font that is the same as the given font except it is not printermatched and it uses ISOLatin1 encoding. If you want to use printermatched fonts and/or some encoding other than ISOLatin1 you should override /**ModifyFont**. (For an explanation of printermatched font see the *NeWS 2.1 Programmer's Guide* and the *X11/NeWS Server Guide*.)

The region tree

/framebufferof — - /**framebufferof** framebuffer

Returns the framebuffer on which the region is located.

/map — - /**map** -

Maps the region. For the purposes of mapping, regions resemble transparent canvases in that unmapping a region does not cause it to disappear—a refresh is required.

/mapped? — - /**mapped?** boolean

Returns whether the region is currently mapped onto the screen.

/unmap — - /**unmap** -

Unmaps the region. Unmapped regions are not painted in bags.

/Parent — /**Parent** parentcanvas (Variable)

The region's parent canvas.

/reparent — parentcanvas /**reparent** -

Changes the parent of the region so that parentcanvas is it's parent.

Geometry

All coordinates are relative to the region's canvas parent, i.e., not the CTM.

/bbox — - /**bbox** x y width height

Returns the bounding box of the region.

/location — - /**location** x y

Returns the coordinates of the origin of the region.

/minsize — - /**minsize** width height

Returns the minimum size of the region. If the data your region manages requires you to enforce some minimum region size override /**minsize** to provide an appropriate value.

/move

x y /**move** -

Moves the origin of the region to x,y.

/path

x y width height /**path** -

Sets the current path to be the rectangle that is defined by the arguments. Regions always have a rectangular path.

/preferredsize

- /**preferredsize** width height

Returns the preferred size for the region. The preferred size is some ideal starting size that you determine. The Toolkit uses **/preferredsize** when clients are added to panels and border bags; unreshaped regions are reshaped to their **/preferredsize**.

One way to define **/preferredsize** in a subclass is:

```
/preferredsize { width height } def
```

By default **/preferredsize** returns the region's **/minsize**.

/reshape

x y width height /**reshape** -

Reshapes the region to fit the bounding box defined by the arguments.

/reshaped?

- /**reshaped?** boolean

Returns whether the region has been reshaped.

/size

- /**size** width height

Returns the size of the region.

Validation

The NeWS Toolkit uses a validation scheme to determine whether an object needs to have its visual presentation updated. One way to understand this is through the model-view portion of the model-view-controller paradigm. Remember, the model is a data object, representing application information and the view presents its model in a graphical fashion. TNT has several objects that manage other objects. For example, menus manage a list of items, and bags manage a collection of clients. The list of items in a menu and the collection of clients in a bag are the model. How they are displayed on the screen is the view.

Your application may allow the user to alter the model, by adding or deleting menu items, for example. When the user alters the list of items, TNT marks the menu as invalid by sending it the **/invalidate** method. Saying a menus is invalid is the same as saying that the model has changed but the view hasn't been updated to reflect those changes.

The **/paint** method is used to initiate the process that causes an object's view to match its model. But, in order to be efficient, TNT doesn't automatically repaint objects when they become invalid, i.e., TNT doesn't call **/paint** on invalid objects, you do. This optimization allows you to change several attributes of an object and only perform one repaint.

Certain operations are considered "basic" enough that they automatically update the view. **/setvalue** and **/setcolors** are examples of methods that do the automatic update. However, if the view is already invalid for other reasons (such as adding items) then these methods do not update the view.

/Valid?

/Valid? boolean (Variable)

Determines whether the region is valid. An invalid region is one that needs some operations executed before it can be repainted; a valid region is one that is ready to be painted. ClassRegion does not interpret what it means for a region to be invalid and what should be done to an invalid region is left to subclasses. Subclasses can interpret what to do by overriding the validation methods (**/invalidate**, **/valid?**, **/?validate** and **/validate**). Subclassers should call super in their override of the validation methods, e.g.,

```
/?validate {
    . . .
    /?validate super send
} def
```

/invalidate

- **/invalidate** -

Marks the region as invalid.

/valid?

- **/valid?** boolean

Returns whether the region is valid.

/?validate

- **/?validate** -

Validates the region if it is currently invalid.

/validate - **/validate** -

Causes the region to be marked as valid.

Region damage handling

Regions are generally contained in bags and don't get the Toolkit's damage handler method, **/HandleDamage**, directly. Instead, the bag holding the region gets the **/HandleDamage** message and then sends **/FixAll** to the regions in the damage path. As with canvases, **/FixAll** sends **/Fix** and **/FixChildren** to each region. (See **/HandleDamage** in ClassCanvas on page 53.)

/FixAll - **/FixAll** -

Subclasser methods: /Fix
/FixChildren

Paints the damaged area of the region. **/FixAll** is called for you. It is not necessary for you to call **/FixAll** yourself.

/FixAll calls **/Fix** and **/FixChildren**; override these methods in your ClassRegion subclass rather than **/FixAll**.

/Fix - **/Fix** -

Handles the repainting of the damaged portions of a region. By default it simply calls **/Paint**, which simply paints the damaged section of the region without trying to be efficient.

Despite the fact that only the damaged area of the region is painted, all the code in your **/Paint** definition is executed. If your **/Paint** has complicated, time-consuming calculations, you may want to override **/Fix** so you can execute only the painting code that applies to the damaged area. If your painting procedures are simple and not time-consuming, you can just use your painting procedures "as is" and allow the server to clip away the bits that aren't necessary.

/FixChildren - **/FixChildren** -

Handles the repainting of damaged region children. Sends **/FixAll** to all the children. **/FixChildren** is a no-op in ClassRegion.

Mouse tracking

Regions are not themselves Trackable. If the canvas on which they sit is Trackable then a region will receive the methods in this section when the condition(s), described under each method, are met.

/TrackStart

event **/TrackStart** -

Sent to a region when:

- SELECT is pressed in the region;

or

- when SELECT is pressed in the background and, with SELECT still down, the mouse is dragged into the region.

/TrackStart executes in the local event manager. The **/TrackStart** method in ClassRegion is more indiscriminate that its counterpart in ClassCanvas. It does not return information to indicate whether tracking should occur.

/TrackStop

event **/TrackStop** -

Sent to a region when the SELECT button is released inside the region.

/TrackMotion

event **/TrackMotion** -

Sent to a region when SELECT is pressed and the mouse is being dragged inside the region.

/TrackCancel

event **/TrackCancel** -

Sent to a region when SELECT is pressed and the mouse is dragged outside the region or when **/TrackCancel** is received in the bag instance.

Region menus

Regions are, by default, Menuable but their menus do not appear unless the canvas on which they sit is Menuable. (See *Canvas menus* on page 53.)

/Menu

/Menu menu (Variable)

The menu that is managed by the region.

/setmenu

menu | null /**setmenu** -

Installs or removes a popup menu for the region. The menu is activated by the user pressing the MENU button when the pointer is over the region. You shouldn't send /**setmenu** to a region that is on a canvas that hasn't been made Menuable. /**setmenu** can be sent to a class or an instance.

/menu

- /**menu** menu

Returns the value of /**Menu**.

/MenuStart

invoker posname event /**MenuStart** invoker posname event menu true
| invoker posname event false
| invoker posname event null true

Determines which menu (if any) should be shown over this object, and where it should be positioned. You can change where the menu pops up by:

- Overriding /**MenuStart** and modifying the XLocation, YLocation pair of the event;

and/or

- Changing posname.

posname defaults to /Default, which means that the default item (or the item at index 0 if there is no default) is aligned beside the location of event. Currently, the only other legal posname is /NorthWest, which corresponds to the top-left corner of the menu. (posname is used by menu buttons for positioning their menus.)

invoker is the object that most recently caused the menu to be displayed. The invoker also determines the local event manager in which the menu's notifier executes and is the default target of the menu (see *The invoker mechanism* on page 117 in ClassMenu).

By default /**MenuStart** returns the menu held in the /**Menu** class/instance variable, and does not modify the positioning arguments.

Returning invoker posname event false causes event to be redistributed allowing some other object to open a menu.

The invoker posname event null true return is provided so that you can respond to MENU down over your canvas or region by not opening any menu and preventing anyone else (e.g., the window under region) from opening a menu in response to the MENU.

/MenuStop

event /**MenuStop** -

Sent to a region when MENU is released. Use /**MenuStop** to clean up anything you created in /**MenuStart**.

Obsolescence and destruction

/destroy

- /**destroy** -

Destroys the region.In a garbage collected system you get rid of an object by dropping any references you created to it. If the toolkit holds references to your objects, they will be dropped when you drop your references: you don't need to worry about what reference the toolkit maintains.

You should override /**destroy** in your subclasses when instances of these classes hold references that directly or indirectly point back to the instance. The parent -> child -> parent circle between a bag and its regions is an example of this kind of reference. In this case your /**destroy** should break the circle and call:

```
/destroy super send.
```

You can also override /**destroy** to send /**destroy** to other objects you're managing in order to minimize the generation of obsolete events.

/**destroy** is executed in the global event manager.

Miscellaneous

/pointinregion?

x y /**pointinregion?** boolean

Indicates whether the point represented by x and y is located within the region's boundaries.

/eventmgr

- /**eventmgr** event-manager

Returns the event manager for the region's canvas parent.

Scrollbars

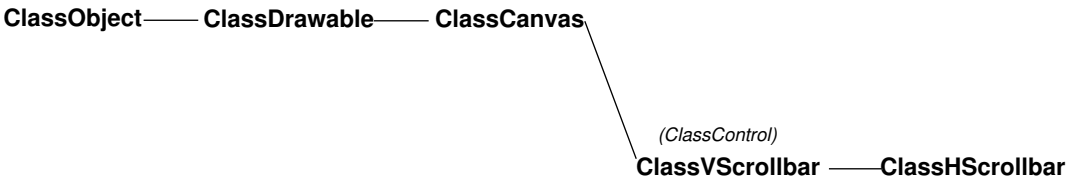


Figure 18-1 The scrollbar subtree

TNT Scrollbars are a subclass of ClassCanvas and implement OPEN LOOK scrollbars. ClassVScrollbar provides vertical scrollbars and ClassHScrollbar provides horizontal ones.

TNT scrollbars act like controls in that they have a value and a target at which they direct certain actions. As with other controls, scrollbars also have notifiers that are sent to scrollbar targets.

The scrollbar classes are very complete and by default do most of the calculations and visual updating themselves. When a user interacts with the scrollbar, it moves itself to a new value, and notifies its target of the new value.

The value of a scrollbar is defined as the first position currently displayed in its target. At any given time the visual presentation of a scrollbar is defined by three parameters:

- the total number of objects in the view

- the position of the first visible object
- the position of the last visible object

Definition of "Object." The definition of object depends on what is being scrolled. In a scrolling list an object is anything that comprises a legal item in the scrolling list; in a text pane objects might be a lines of text, a characters, or pages.

Creation

/new — parentcanvas /**new** instance

Creates a new scrollbar instance. Sent to ClassVScrollbar to create a vertical scroll bar and ClassHScrollbar to create a horizontal scrollbar.

Scrollbar auto repeat

To control scrollbar repeating yourself, you need to have two variables defined in your UserProfile. However, if you don't define these variables, default values are stored in UserProfile for you.

/**ScrollThresh**—determines how long after the /MouseDown that the scrollbar begins repeating. The default is 200 milliseconds.

/**ScrollDelay**—determines the time that elapses between each repeat. The default is 100 milliseconds.

/motion — - /**motion** motion-name

Returns the type of elevator motion. The return value is determined by where the cursor is on the scrollbar when the user presses SELECT. The OPEN LOOK GUI Specification determines, in general, what kind of motion is caused by user interaction with each part of the scrollbar. See Figure 18-2.

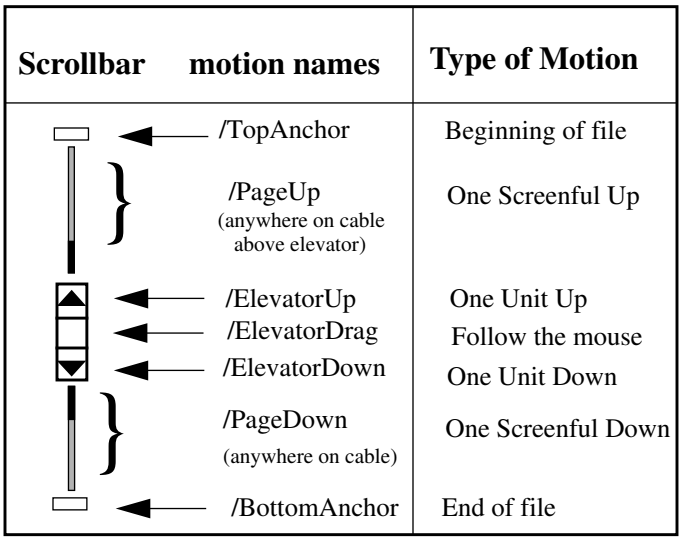


Figure 18-2 Map from scrollbar components to motion names

/warpcursor

- /warpcursor -

Warps the cursor when required. For example if you press and hold the left mouse button over the /ElevatorDown, the elevator moves downward as long as the button is depressed; the cursor is warped so the pointer stays over the /ElevatorDown.

But, the cursor can't be warped until the scrollbar is updated because the position of the elevator isn't known until the update occurs. Thus, if you have taken control of the notification and updating of the scrollbar via **/HandleMotion** you should call **/warpcursor** after you have caused the scrollbar to be updated (via a call to either **/setparameters** or **/setvalue**).

Geometry

/minsize

- /**minsize** width height

Returns the minimum acceptable size for the scrollbar. The *OPEN LOOK Graphical User Interface Specification* specifies what the smallest scrollbar looks like.

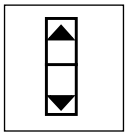


Figure 18-3 The minimum acceptable size for a scrollbar.

/preferredsize

- /**preferredsize** width height

Returns the size you think is ideal for your scrollbar. The default preferredsize describes a scrollbar with a full elevator and anchors but no cable.

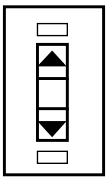


Figure 18-4 The default preferredsize scrollbar

Values and parameters

/setparameters

value viewsize max /**setparameters -**

Sets the parameters of the scrollbar and updates the visual presentation of the scrollbar to reflect the new parameters.

value is the index of the first object visible; it ranges in size from 0 to max-1.

viewsize is the number of objects currently visible.

max is the total number of objects being scrolled.

Thus in a text window that wanted to scroll by lines, value is the index of the first visible line; viewsize is the total number of lines currently visible; and max is the total number of lines in the file that the text program is managing.

/parameters

- /**parameters** value viewsize max

Returns the scrollbar's current parameters.

/setvalue

value /**setvalue** -

Sets the value of the scrollbar and updates the visual presentation of the scrollbar to reflect the new value. value ranges from 0 to max–1 (see /**setparameters** and Figure 18-5).

/value

/**value** value

Returns the current value of the scrollbar; the value is from 0 to max–1 (see /**setparameters** and Figure 18-5).

/EventToValue

event /**EventToValue** value

Resolves an event into the current value of the scrollbar. Used by subclassers, i.e., if you're going to do an /**installmethod** on /**HandleMotion**.

Notification and previewing

The scrollbar's notifier is called when the user interacts with the scrollbar in all ways except dragging the elevator. When the elevator is dragged the previewer is called. If you want no scrolling to occur when the user drags the elevator define a notifier as usual but don't define a previewer. If you want continuous scrolling when the user drags the elevator define the previewer to be the same as the notifier.

/setnotifier

notifier-name | proc /**setnotifier** -

Stores the name of a notifier or the notifier procedure itself in a scrollbar class or instance. If you use a name, the notifier is sent to the scrollbar's target with two arguments, the value of the scrollbar and the scrollbar instance itself:

```
value <scrollbar-instance> /yournotifier
```

The notifier should consume its arguments.

If you've stored a procedure in the scrollbar, the scrollbar's value and the scrollbar itself are pushed on the stack and then your procedure is executed. The new value of the scrollbar depends on where SELECT went down on the scrollbar.

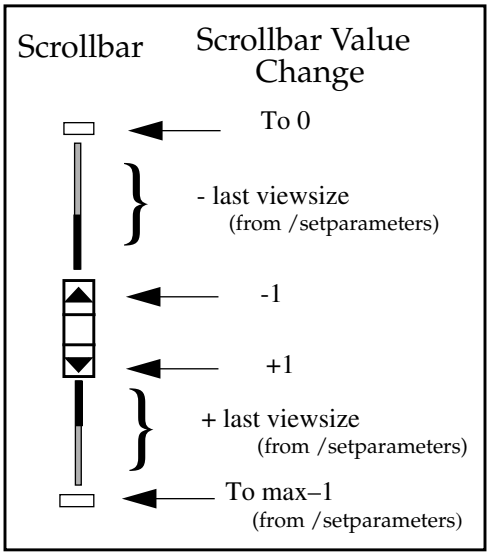


Figure 18-5 How a scrollbar's value changes

/notifier — - /**notifier** notifier-name | proc

Returns the name of the notifier or the executable procedure.

/setpreviewer — previewer-name | proc /**setpreviewer** -

Sets the scrollbar's previewing procedure. The previewer is called when the elevator is dragged. If you want no scrolling to occur until the user lets up on the mouse define a notifier but leave the previewer undefined. If you want continuous scrolling when the user drags the elevator define the previewer to be the same as the notifier.

/previewer — - /**previewer** previewer-name | proc

Returns the scrollbar's previewer.

Scrollbar motion

/HandleMotion

event /**HandleMotion** notifier-arguments

/HandleMotion is called internally whenever the user interacts with the scrollbar. In particular, it is called immediately before the scrollbar's notifier is executed. Therefore, **/HandleMotion** is responsible for calculating the arguments that the notifier expects, and pushing them onto the operand stack.

Figure 18-6 shows the default **/HandleMotion**. The API for scrollbars says that the notifier is called with the scrollbar and the new value on the stack. Therefore, the default **/HandleMotion** simply calculates a new value and leaves it on the stack.

A new **/HandleMotion** can be methodinstalled into scrollbar instances for applications that need more control over the scrollbar. Figure 18-7 shows a **/HandleMotion** that is automatically installed into all scrollbars that are used by the Jot text package. This **/HandleMotion** just describes the motion that occurred and passes that information to Jot, and lets Jot calculate a new set of parameters for the scrollbar.

This **/HandleMotion** produces two arguments for the notifier, which is allowed, because it was Jot that supplied both the **/HandleMotion** and the notifier. Because this **/HandleMotion** doesn't actually change the value of the scrollbar, Jot issues either a **/setvalue** or **/setparameters** once the new values are calculated. And at that time Jot will also warp the cursor with the **/warpcursor** scrollbar method.

```
/HandleMotion { event => <notify-args>
        % Calculate a new current value for the scrollbar base on the kind of
        % motion that just occurred.
        %
        /motion self send {                                     % event motion
                /TopAnchor { pop 0 }                            % 0
                /BottomAnchor {
                        pop /parameters self send               % value viewsize max
                        3 1 roll pop pop                        % newvalue
                }
                /ElevatorUp { pop /value self send 1 sub }      % newvalue
                /ElevatorDown { pop /value self send 1 add }    % newvalue
                /PageUp {
                        pop /parameters self send pop           % value viewsize
                        sub                                     % newvalue
                }
                /PageDown {
                        pop /parameters self send pop           % value viewsize
                        add                                     % newvalue
                }
                /ElevatorDrag {                                 % event
                        /EventToValue self send                 % newvalue
                }
        } case                                                  % newvalue

        % Now set the scrollbars value to the newvalue. This moves the elevator automatically.
        % Then warp the cursor to the new elevator position. Finally, leave the new value on the
        % stack for the notifier.
        %
        /setvalue self send                                     % -
        /warpcursor self send
        /value self send                                        % value
} def
```

Figure 18-6 The default /HandleMotion definition for scrollbars

```
/HandleMotion { event => <notify-args>
        /motion self send {                                     % event motion
                /ElevatorDrag {                                 % event
                        /EventToValue self send                 % newvalue
                        /Absolute                               % newvalue /Absolute
                }
                /TopAnchor { pop -1 /Document }
                /BottomAnchor { pop 1 /Document }
                /ElevatorUp { pop -1 /Line }
                /ElevatorDown { pop 1 /Line }
                /PageUp { pop -1 /Page }
                /PageDown { pop 1 /Page }
        } case
} /installmethod jotscrollbar send
```

Figure 18-7 Example of how /HandleMotion can be modified.

ClassScrollList

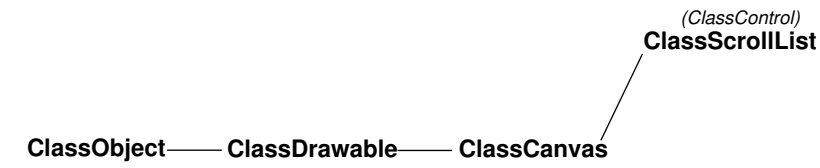


Figure 19-1 The ClassScrollList subtree

Introduction

A scrolling list is a list of items that you can scroll through and in most cases choose one or more of the items. However, you have to add a scrollbar to the scroll list.

The items in scrolling lists are similar to settings: they can be exclusive or nonexclusive. The NeWS Toolkit (TNT) provides the following kinds of scrolling lists:

- /Exclusive
  A scrolling list from which you can choose one item.
- /ExclusiveVariation
  A scrolling list from which you can choose one or none of the items.
- NonExclusive
  A scrolling list from which you can choose none, one, or multiple items.

A ClassScrollList instance presents a list of items in its canvas; the scroll list is said to manage its items. These items are specified using TNT display items; therefore the following types are available:

- strings with font modifiers
- strings with color modifiers
- strings with font and color modifiers
- canvases (including orphans)
- PostScript fragments that can respond to **/paint** and **/size** messages

Prior to display, the list is validated to determine the maximum item width and height. Using the maximum height, the number of visible items and the row height is calculated. Because items can be different sizes, each item is centered within the row height. As the maximum item height changes, the layout also changes.[1]

When scrolling occurs, a minimum number of items will be repainted. Total repaint happens when three or fewer items are visible in the canvas or no items overlap.

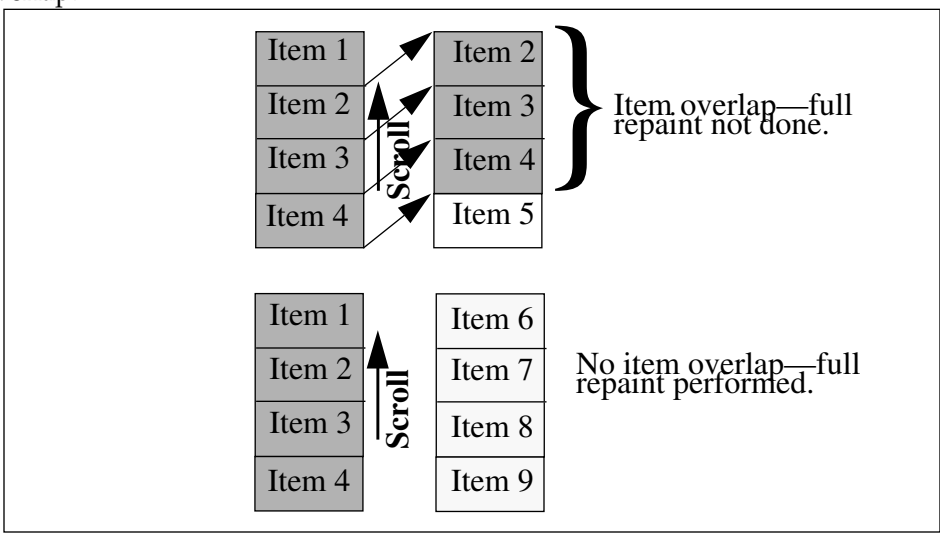


Figure 19-2 Interaction between scrolling and and painting a list.

1. The maximum width is calculated to limit the number of bits affected during a scroll operation. The scroll operation is implemented using copyarea.

Each item is accessed through a zero relative integer (called the item-index) representing its position in the list. The item-index is not constant, because items can be inserted or deleted in the list. *No error handling is performed when an index is out of range.*

When an item in the list is chosen, the rectangle surrounding the item is recessed as indicated in the *OPEN LOOK Graphical User Interface Specification*.

Items

/setitemlist

[display-item display-item ...] /**setitemlist** -

Sets the list of items to be managed in the scrolling list. Invalidates the list. To unset the current list you can use nullarray. Unsetting the item list also initializes (i.e., clears) the list of choices. If you call /**setitemlist**, with a non-null array, on a list that already has items, the item list is reset and the list of choices is cleared.

/itemlist

- /**itemlist** [display-item display-item ...]

Returns an array that contains the list of items managed in the scrolling list. Due to the nature of composite objects in PostScript, you must not modify the array[1].

/item

item-index /**item** display-item

Returns the display item at item-index.

/itemcount

/**itemcount** number-of-items

Returns the number of items managed in the scrolling list.

/insertitem

item-index display-item /**insertitem** -

Inserts display-item into the list at the specified item-index. Invalidates the scrolling list and no automatic repainting occurs. When the instance is validated, the row height and visible items might change.

1. If you need to modify the returned array you should make a copy of it and modify the copy. One way to copy the array is:
dup length array copy

/replaceitem

item-index display-item /**replaceitem** -

Replaces the item referenced through item-index with display-item. Invalidates the scrolling list and no automatic repainting occurs. When the list is validated, the row height and visible items might change.

/deleteitem

item-index /**deleteitem** -

Deletes the item referenced through item-index. Invalidates the scrolling list and no automatic repainting occurs. When the list is validated, the row height and visible items might change. If the item you are deleting is chosen, its item-index is also deleted from the choice list.

If you delete an item that precedes chosen item, the chosen item, stays chosen and it is assigned an item-index appropriate to its new position. The chosen item's new item-index is added to the choice list.

/appenditem

display-item /**appenditem** -

Appends the item to the end of the list. Invalidates the scrolling list and no automatic repainting occurs.

/itemvisible?

item-index /**itemvisible?** boolean

Returns a boolean that indicates whether the item referenced through item-index is visible. This method may return incorrect results when the instance is invalid.

Choices

/setchoicemode

mode /**setchoicemode** -

Sets the choice mode for the scrolling list. mode must be defined as /Exclusive, /NonExclusive, or /ExclusiveVariation. The default mode is /NonExclusive.

/choicemode

- /**choicemode** mode

Returns the choice mode for the scrolling list.

/locatechoice

- /locatechoice -

Provides an interface to the OPEN LOOK menu commands “Locate Choice” in exclusive lists and “Locate Next Choice” in nonexclusive lists. When the instance is valid, the list is scrolled to the next choice that is not currently visible in the canvas. This search does not *wrap around*; if there is no choice below the last visible item, no scrolling occurs.

/clearchoice

- /clearchoice -

Provides an interface to the menu command “Clear All Choices” in scroll lists. When the instance is valid, the choice list is initialized and visible choices will be *raised*. This method is ignored when the choice mode is /Exclusive.

/chosen?

item-index **/chosen?** boolean

Indicates whether the item referenced though item-index is included in the choice list.

/setvalue

[item-index ...] **/setvalue -**

Sets the value of the scroll list. **/setvalue** raises visible choices *if* the old choice is not in the new array and recesses visible choices *if* the new choice is not in the old array. Choices contained in both arrays are not altered. Recessing and raising occurs only if the instance is valid.

When the choicemode is /Exclusive, the array passed to **/setvalue** should contain 1 element; otherwise, the behavior of this operation is undefined. Similarly, when the choicemode is /ExclusiveVariation, the array passed to /setvalue should contain 0 or 1 element.

/value

- /**value** [item-index item-index ...]

Returns an array that contains the list of choices. Due to the nature of composite objects in PostScript, you must not modify the array.

Scrolling and scrollbars

Though not mandatory, most applications will use a scrollbar to scroll the scrolling list. For a explanation of using scrollbars see *Scrollbars* on page 151. the following sequence is one way you could associate a scroll list with a scrollbar.

1. A scrolling list is instantiated:

```
/list framebuffer /new ClassScrollList send def
```

2. A vertical scrollbar is instantiated.

```
/scrollbar framebuffer /new ClassVScrollbar send def
```

3. Using ClassControl target mechanism the scrolling list becomes the "target" for notification from the scrollbar.

```
list /settarget scrollbar send
```

4. The /**scroll** method in the scrolling list is executed during scrollbar notification.

```
/scroll /setnotifier scrollbar send
```

5. The /**scroll** method in the scrolling list is also executed during scrollbar previewing.

```
/scroll /setpreviewer scrollbar send
```

6. The scrolling list is informed that it has an attached scrollbar. The scrolling list will use this reference to set and update the scrollbar's parameters.

```
scrollbar /setscrollbar list send
```

/setscrollbar — scrollbar-instance /**setscrollbar** -

Associates scrollbar-instance with the scrolling list and sets the parameters of the scrollbar (see /**setparameters** in Scrollbars on page 154). You still must make the list the target of the scrollbar. To unset the scrollbar you can define scrollbar-instance as null.

/scroll — item-index scrollbar /**scroll** -

The notifier for the list's scrollbar.

/scrolltohere — item-index /**scrolltohere** -

Scrolls the list to put item-index at the top.

List geometry

/rows — **-** /**rows** rows

Returns the number of rows that fit in the visible portion of the list.

/heightfromrows

rows /**heightfromrows** height

Calculates the appropriate height for the specified number of rows. The returned height can be used with /**reshape**, /**minsize**, and /**preferredsize** to ensure an appropriate number of visible items (rows) in the scrolling list.

/setrowgap

points-between-rows /**setrowgap** -

Sets the amount of space (in points) between scroll list items. Invalidates the scrolling list.

/rowgap

- /**rowgap** points-between-rows

Returns the amount of space (in points) between items and between the visible item list and the scroll list itself.

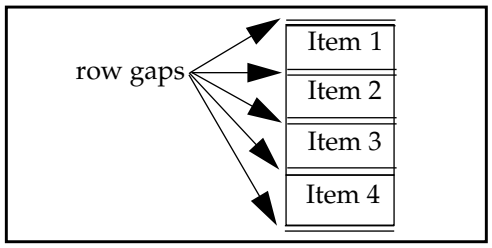


Figure 19-3 Row gaps in scroll lists.

Ability to interact with scroll list items

/setvisualstate

state /**setvisualstate** -

Sets the state of the list. state is one of /Active, /Inactive, or /Busy. /Active is the default. The list is not repainted to reflect the new state unless it is valid. The state of a list determines not only its visual presentation but also whether users can interact with the scroll list. Users can interact with /Active lists; /Inactive and /Busy lists ignore user interaction.

/visualstate

- /**visualstate** state

Returns the visual state of the list. state is one of /Active, /Inactive, or /Busy.

Notification

/setnotifier — notifier /**setnotifier** -

Sets the notifier in the scrolling list; notifier can be either a PostScript name or a PostScript code fragment.When notifier is specified as a PostScript name type, it is used in conjunction with the target interfaces to dispatch notification to the appropriate target. Whenever an item is chosen, notification occurs. During notification the following objects are pushed onto the stack:

- An array describing the change to the list. The array contains either the item-index(es) of the items chosen and a boolean that is true if the was chosen or or false if unchosen.
- An array of choices for /NonExclusive; an item-index and a bool for an /ExclusiveVariation; a single item-index for /Exclusive.
- the scrolling list instance, and the notifier name are pushed on the stack, prior to invoking **/sendtarget**.

Although not recommended, notifier can also be specified as a PostScript code fragment. The above mentioned array, the scrolling list instance, and the fragment are pushed on the stack, then the fragment is executed.

Notification is performed within the context of the Local Event Manager (LEM).

/notifier — - /**notifier** notifier

Returns the notifier for the scrolling list.

/setpreviewer — previewer | null /**setpreviewer** -

Sets the previewer procedure for the list. null removes the previewer from the list.

/previewer — - /**previewer** previewer

Returns the previewer for the list.

The target interface

Use targets with scroll lists to make the scroll list the target of its scrollbar and to make other objects targets of scroll list notifiers.

/settarget — object /**settarget** -

Sets object to be the target of the scroll list.

/cleartarget — null | object /**cleartarget** -

Clears the target. If object is specified the target is cleared only if the target and object are the same.

/target — - /**target** null | object

Returns the target object.

/sendtarget — args /method /**sendtarget** results

Sends /method and any necessary arguments to the target and returns the method's results, if any.

Miscellaneous

/settextfont — font /**settextfont** -

Changes **TextFont** to font. Simple string items are affected. The instance is invalidated and no automatic repainting occurs. When the instance is validated, the row height and visible items might change.

ClassSelection

Introduction

A selection is an indication of some data of interest to the user, almost always some information visible on the screen that is about to be used in an operation. The most common example is text that is to be moved or copied from one place to another. Many other objects can be selected, and many operations besides move/copy are possible; for instance, a window may be selected so that its properties may be inquired or manipulated. The NeWS Toolkit provides a ClassSelection, whose instances (*Selections*) describe such a selected chunk of data.

The window system has a global registry that keeps track of a few selections; registering a selection causes any previously registered value to be deselected, and makes the current selection available to all clients of the window system. This registry is implemented inside ClassSelection through utility procedures defined in systemdict.

The instance variables for ClassSelection contain attributes of the selection. Some of these are required by ClassSelection's processing: *Holder* is the object responsible for the selection (which is, by default, the canvas on which the selection appears), and *Rank* is a global identifier that may be any non-null PostScript object The standard ranks are /PrimarySelection, /SecondarySelection, and /ShelfSelection. /ShelfSelection is also commonly referred to as the ClipBoard. Other variables are attributes that support the user interface for making selections, see Table 20-2 on page 182. Finally, a Selection usually also contains information stored by the client to identify the selection.

Note – There is a file in the demo/bin directory on the TNT release tape called `circles` that demonstrates many of the concepts discussed in this chapter. It is strongly advised that you use the circles demo in conjunction with this chapter when you are trying to use selections in your application.

How this chapter is organized

This chapter is divided into three main categories of information:

The section *Retrieving selection values* on page 176 tells you how to ask an existing selection for its value.

The section *Making selections* on page 181 tells you how to make a selection. Making selections is somewhat more complicated, connected as it is to issues of user interface and UI independence.

The section *Responding to selection requests* on page 185 tells you how your selection should respond to requests posed to it.

Definition of "selection client"

The "client" and "selection client" mentioned in this chapter both refer to the canvas in which there is a selection. In most cases this is the **/Holder** but that is not a requirement because there can be more than one canvas associated with a single **/Holder** (see **/Holder** on page 66 in ClassCanvas).

How applications get information about selections

Some users of ClassSelection will make selections, other users will want to get information about selections. ClassSelection provides the means for both creating selections and getting information from them. For example, a series of selection-related transactions involving a dialog window might look like[1]:

1. A user presses the "Load" button in an application window.

1. If the dialog box wanted to make its own selections rather than just use another window's selections then it would have to be Selectable. See *Selectables* on page 65 in ClassCanvas.

2. A dialog window opens. The dialog window tries to fill in the name of the file using the current /PrimarySelection, e.g., some text selected in a command tool.

3. The dialog box first must determine if a /PrimarySelection exists. It would use /**getselection**. (See /**getselection** on page 176.)

4. Now that it has the selection, the dialog window must determine if it can use it. The dialog window is expecting text; ClassSelection "labels" a selection that contains only text, /ContentsAscii (see Table 20-1 on page 178). The dialog box would use the ClassSelection method /**query** to determine if the current selection was text. (See /**query** on page 176.)

5. The dialog processes the selection and performs the appropriate action; in this case the text is installed as the default value in the dialog's text field.

The application that is going to create selections and respond to queries has its own associated series of transactions; see *Making selections* on page 181.

The context of selection processing

It is important to recognize that a selection can exist without being registered in the global database—instances of (subclasses of) ClassSelection are used privately in several parts of the system before being made available to the world at large.

A second important point is that much of the processing described in this chapter is initiated outside the application's context, i.e., its userdict and stdin/stdout are not available.

For instance, global UI code will recognize that a function key has been released, or a drag action performed, signaling that a selection transfer should take place. Similarly, the UI layer, not the application, is responsible for determining which user actions indicate a selection is to be made or adjusted.

One implication of this second point is that the selection's methods are often invoked in some foreign process (the global event manager, or even in another client's process). They must, consequently, be self-contained—if they need some data such as the connection to the C-side client, that must be reachable from the selection instance. Context is important when communicating with a

c-client because of the way NeWS communicates with the client-side (see the *NeWS 2.1 Programmer's Guide* for an explanation of NeWS-Client communication).

Retrieving selection values

Applications can retrieve the value of a selection by sending a message to it. This may require that the application first find that selection in the global registry. To find the selection in the registry use /**getselection**. To get information about a selection's attributes use /**query** or /**request**; /**request** is the more general (and complex) method.

getselection — rank **getselection** selection | null

Returns selection currently registered under rank, or null if none exists. The standard ranks are: /PrimarySelection, /SecondarySelection, and /ShelfSelection, but rank may be any non-null PostScript object.

getselection is a utility procedure in systemdict, you don't send it to an object but just call it. For example to determine if a /PrimarySelection exists you could do:

```
/PrimarySelection getselection dup null ne          % selection bool
```

/query — key /**query** false | value true

Retrieves a single attribute of a selection. key is the name of the attribute desired (e.g., /ContentsAscii; see Table 20-2 on page 182 for a list of attribute keys) and returns the associated value and true. If there is no such value, it returns false.

For example, to use /**query** to retrieve just the characters of a selection you could do the following:

```
/PrimarySelection getselection dup null ne{   % sel
    /ContentsAscii /query 3 -1 roll send {    % val
        ...process the value...               % -
    } if
} {
    pop
} ifelse                                      % -
```

/request

request-dictionary **/request** response-dictionary

Retrieves one or more attributes of a selection at a time[1]. request-dictionary is a dictionary that contains the complete request. Each key in the dictionary names a selection attribute or an operation the selection should perform. For an operation, the corresponding value in the dictionary may be a parameter or array of parameters. For requested attributes, the original value in the dictionary doesn't matter. **/request** should be used when any of the following are true:

- You are asking a selection to do an operation that takes arguments to complete.
- The requester and holder must negotiate the form of the requested data (described below).
- When the cost of communicating with the holder of the selection is high (e.g., the holder must communicate with its C-side client through a slow communication link in order to respond to any request), it may be advantageous to batch queries in a single call to **/request**.

The selection will return a similar dictionary (or modified copy of the same dictionary, as convenient), with results and requested attributes in the value for each key wherever possible; if it cannot store a result, it will store the value /UnknownRequest.

1. The default implementation for /request in ClassSelection is in terms of /SingleRequest: the request-dict is enumerated with forall, and each request/value pair is passed to /SingleRequest. (If /RequestSequence is found, it is passed to its own enumerator, which in turn calls /SingleRequest. When /RequestChoice is implemented it will work in the same way.) If a selection holder wishes to do batch processing of requests, it should override the /request method. Any such override is then responsible for supporting /RequestSequence and /RequestChoice (see Table 20-1 on page 178).

The following example illustrates use of /**request** to ask an editor where the selection is and in what source file. A debugger might do this so that it could set a breakpoint there.

```
/PrimarySelection getselection dup null ne {% sel
        dictbegin               % construct request dict
            /FileName null def
            /StartIndex null def
        dictend                 % sel request
        /request 3 -1 roll send % response
        begin
                ... process response ...
        end
        } {                     % no such selection; null
            pop
        } ifelse
```

The set of request keys passed to the selection holder is open-ended; any set of clients that can agree on the interpretation of a new key, may use that key to communicate among themselves. A convention for the most common requests has not yet been established; but a number of the most useful are suggested here. In general, request names should develop parallel to the conventions of the X11 Window System, as documented in David Rosenthal's *Inter-Client Communication Conventions Manual* (distributed by the X Consortium).

The following table summarizes the conventional request keys. Certain keys request that the client modify the selection in some way. Two "action-type" keys are specified here: /DeleteContents and /ReplaceContents.Most keys represent requests for the Selection to render its value in a named format.

Table 20-1 Request keys

Key Name[1]	Argument	Result
/Canvas	none	the selected object, if it is a NeWS canvas.
/ContentsAscii	none	a PostScript string containing the selected text, without text attributes (font, typeface, etc.)

1. Those keys marked with * are not currently implemented by The NeWS ToolKit, but are defined so that clients who wish to use such requests will have a common interface.

/ContentsPostScript	none	a PostScript object, which, when executed, will recreate the selected value. (This is likely to be most useful for graphical objects, which can be redrawn in a new environment.)
/DeleteContents	none	the contents of the selection are deleted. Tells the client to delete the contents of the selection. This is not the same as merely deselecting or destroying the Selection instance; e.g., in a text item /DeleteContents means remove the selected span of characters from the text. Since there are no parameters required for this operation, either of **/query** or **/request** will work for it. Assuming the holder is willing and able to comply, a null value will be returned.
/ReplaceContents*	none	/ReplaceContents involves a deletion, just like /DeleteContents; but then new data passed as an argument to the request should be stored in place of the deleted material, and the replacement should be selected. In this case (where the requester must be able to pass an argument to the request), the **/query** method will not work. Instead, the **/request** method is used, with a request dict for its argument. In the request dict, the key /ReplaceContents is defined, with the replacement contents as its value. This style of passing parameters enables a consistent interface to be maintained between requester and selection holder, regardless of the particular requests.[1]
/FirstIndex *	none	the count of how many objects of size Level precede the first object in the selection.
/Level *	none	the multi-click level of the selection.
/LastIndex *	none	the count of how many objects of size Level precede the last such object in the selection.
/RequestChoice *	[request arg ...]	a list of alternative requests (with parameters for each) of which the holder should respond to one.
/RequestSequence	[request arg ...]	a list of requests (with parameters for each) which the holder should respond to in order.

1. Note: No Toolkit selections currently support (or attempt to use) the /ReplaceContents request. It is specified here so that clients who may choose to implement it will have a consistent protocol. The protocol described in the table matches that in the ICCCM.

/SelectionObjsize	none	the size of the selection as measured in bytes. Note the lower-case 's' in "Objsize". In order to paste TNT selections into XView applications you must include /SelectionObjsize in your /**SingleRequest** definition (page 186).
/TransferSelection *	dict[.../Source: Selection...]	one selection is requested to perform a transfer between itself and another; see the note at the end of the next section.

Since /**query** retrieves only one selection attribute at a time, the requester can easily control the order in which requests are processed. This is not so easy with /**request**: The order of objects in a dictionary is undefined, so if there is a required order to the requests, the requester must take special pains. It should define only one key in the request dictionary, /RequestSequence, and its value should be an array. The 0th, 2nd, etc. elements of the array will be taken as requests, and the following (odd-numbered) element for each will be the corresponding parameter/value. In the dict returned by /**request**, the value associated with /RequestSequence will be an array in which the odd-numbered elements are the values returned by the selection holder.

A similar mechanism allows the requester and holder to negotiate over the form of response. The requester uses the key /RequestChoice, which is defined to an array similar to the one used with /RequestSequence. In this case, the keys in the even-numbered positions of the array are included in the order of the requester's preference. The holder may then choose any of the requests in the RequestChoice to respond to; the key /RequestChoice is redefined to a new array containing the single key responded to and its corresponding value. If none of the choices is acceptable, the array should be replaced by /UnknownRequest. (If one of the choices in the /RequestChoice array is in turn a /RequestSequence, it is deemed responded to only if all the requests in the sequence are acceptable.)

Note – Like /ReplaceContents, the /RequestChoice key is not currently supported by the NeWS Toolkit. Individual clients may choose to implement it if they are willing to run the risk of having their code become obsolete.

The full details of request processing are described below under "Responding to Requests."

When and how to transfer a selection value

Clients will occasionally decide on their own initiative that they should retrieve a selection value; for instance, the debugger example above would probably be triggered by invocation of a "Breakpoint" panel button or menu command. But most of the time, global user interface code will determine that a selection transfer is called for. If a client can accept input from the user (keystrokes or mouse drawings, for instance), then it should generally also be ready to accept the contents of a selection. Whenever the UI code determines this is appropriate, it will send an event to the destination of the transfer.

In order for your canvas to receive these transfer events you must make it Receptible. See Chapter 5, *Drag and Drop—receptible canvases on page 71.*

Making selections

Creation

/new

rank holder /**new** selection

Creates an instance of ClassSelection.

A selection client is a canvas that can contain a selection, or be one itself (e.g. a text window or a frame).

Selection clients will subclass two classes: ClassSelection and ClassCanvas. In both, there are subclass responsibility methods that the client must implement in its own subclass. If your canvas is handling text you may want to subclass ClassTextCanvas instead of ClassCanvas.

Below is a typical sequence of operations for a selection-client application:

1. Subclass ClassCanvas to be Selectable and to provide definitions of the selectable subclass responsibility methods and **/SelectableType** (see *Selectables* on page 65 in ClassCanvas).
2. Subclass ClassSelection to provide subclass responsibility methods.
3. Activate an instance of your subclass of ClassCanvas (see *Activation and deactivation* on page 46 in ClassCanvas). Because **/Selectable?** is true the canvas is added to a global interest managed by UI-specific code.

4. The UI code matches certain events (which events get matched depends on the particular UI), and decides a selection action has occurred, e.g., SELECT goes down over the canvas.

5. The UI code sends the **/NewSelection** message to the canvas. **/NewSelection** returns a new Selection of the appropriate class (i.e., an instance of the client's Selection subclass created in step 2). See **/NewSelection** on page 67 in ClassCanvas.

6. The UI code then sets instance variables within that selection (selection attributes) to indicate what's going on (multiclick level, etc.). See Table 20-2 for a list of the relevant instance variables. You should realize that it is the UI code that sets these variables. When the selection is delivered to your canvas you can query the selection as to the value of the variables you want to know about.

7. Next the UI code calls more canvas subclass responsibility methods to get the client to finish resolving the operation. E.g., the ADJUST button may be used to extend the selection, causing the UI code to call the *canvas's* **/SelectionStart** method (see **/SelectionStart** on page 69 in ClassCanvas).

8. When the UI code decides that the user action is complete (e.g., a mouse button is released), it notifies the canvas that the selection is complete (see **/SelectionStop** on page 70 in ClassCanvas) and then registers the Selection in the global dictionary so that other applications can access it, via **getselection**.

Note that most of the selection actions are driven by the UI code. Clients need write relatively little code for their particular selections. In particular, clients should *not* try to interpret raw device events—some are not readily accessible to the client; most are subject to user-modification (some left-handers swap the meaning of the mouse buttons and function keys for left-handed use); and user interfaces are subject to many changes an application will do well to ignore if it can.

Table 20-2 Class Selection attributes used in making selections.

Key (Name)	Value Type	Interpretation
/DeleteSource?	boolean	Meaningful only if the selection is being passed to a **/DragStart** or **/DragAdjust** canvas method, in which case the key is true if the operation is a Move rather than a Copy.
/ToggleSelected?	boolean	This key is defined only if the SelectableType of the canvas that made the selection is /Canvas or /Graphic. For /Canvas Selectable

types, it shows a change in the previewed-state (as distinguished from the true state) of the selection. E.g., Adjust-down on an icon [de]- hilites it immediately, and then toggles its hiliting as you slide off and on the icon.

In /Graphic selections, it is set from **/SelectionStart** (page 69 in ClassCanvas) through **/SelectionStop** (page 70 in ClassCanvas). Its interpretation depends on the form of the adjustment, indicated by the Pin value at SelectionStart time:

/Pin=/NoPin	The object under the cursor is being toggled. If the client is doing any previewing on **/SelectionAdjust** (page 69 in ClassCanvas), the last object toggled must be cached so that it may be restored.
/Pin=/AtPoint	There is no object under the cursor. A bounding box should be started, and all selectable objects enclosed by it at **/SelectionStop** should be toggled.

/Holder	any	Determines who is the selection client.
/InsertionPoint	name	Tells the client the selection's insertion point if some value is copied to the selection. In particular, if this is a primary selection the canvas should change its input focus location. name is one of:

/LowEnd	The low end of the existing selection (first)
/HighEnd	the high end of the existing selection (last)
/NearEnd	whichever of first/last is closest to the current point
/FarEnd	whichever of first/last is furthest from the current point
/AtPoint	the cursor position (current)

/Level	int	The "size" of objects to be selected. For instance, in text, 1 may indicate a character, 2 a word, 3 a line or sentence, etc. Essentially, for OPEN LOOK, /Level is a multi-click count.
/PendingDelete?	boolean	true if the selection should be replaced by the next user input action (always, for primary selections in OPEN LOOK).
/Pin	any	Tells the client where to "anchor" the selection during an **/SelectionAdjust** operation. The pin can be a name, in which case it can be evaluated using **/ComputeNamedPosition** (described below

under "Utilities"), or it may be an arbitrary value (typically an int) representing a previously computed anchor point. The possible names are:

/LowEnd	The low end of the existing selection (first)
/HighEnd	The high end of the existing selection (last)
/NearEnd	Whichever of first/last is closest to the current point
/FarEnd	Whichever of first/last is furthest from the current point
/AtPoint	The cursor position (current)
/NoPin	No pin. One use for /NoPin is toggling the selected state of the object under the cursor.

/Rank	any	Most selections have Rank eq /PrimarySelection. Selections made while some function-keys are down have Rank eq /SecondarySelection. (These get reflected differently, and have special uses.) The Clipboard has Rank eq /Shelf-Selection. Other Ranks are possible.
/Registered?	boolean	true if the selection has been registered in the global database via /**setselection**.
/SelectResult?	boolean	Determines whether data that has been transferred via a selection operation (e.g., Cut and Paste) should be selected after the transfer.
/Style	any	The style of highlighting recommended by the UI manager. Currently defined values are /Default, /Invert, /Outline, /StrikeThru, and /Underscore. Clients may ignore this value if they think it does not apply to their selection type, basing their highlighting instead on /Rank and /PendingDelete?.

Finally, to assist clients in correctly reflecting changes to the selection, every time the /**SelectionAdjust** message is sent, the accompanying selection will have the name /Changed defined to an array of some of the above names; each key in the array has changed value since the previous adjustment.

Registering a new selection; unregistering an old one

When the UI layer decides that a selection specification has been completed, it sends **/setselection** to the selection; the default implementation registers the instance in the global database. Clients may override **/setselection** if they need to adjust state or maintain any additional information when one of their selections becomes publicly available. Of course, it is also possible for a client to create a new selection on its own initiative and send it **/setselection**; it will get registered just the same.

Just as a client may register a selection on its own initiative, it can unregister one by sending **/unsetselection** to it. A warning given above bears repeating here: It often happens that a **/Deselect** message is generated in the client which is making a new selection (the cause of this one being Deselected); the message-send in this context is in danger of communicating with the wrong client. For instance, printing data over the current process's standard out is likely to send it over the wrong connection. Clients must take care to store some access back to their generating client in selections they create, so they can communicate with it reliably when that selection receives a message while running in another process. Other messages that may be sent to a selection face analogous dangers: **/SingleRequest** and **/destroy** in particular are liable to be sent to a selection while running in some process other than the selection's Holder's client. Equivalent care is required in these cases to respond in the proper context.

/Deselect

- /Deselect -

When a new selection is registered, any old selection already registered under that rank is sent a **/Deselect** message. **/Deselect** is a strict subclassresponsibility method: there is no implementation in ClassSelection. The subclasser's method should at least dehighlight the selection; most selections will also destroy themselves. However, a deselected selection might be retained, for instance to support restoring the selection in an Undo operation.

Responding to selection requests

Selections respond to requests through either of two methods, **/request** or **/SingleRequest**, and you must override ride *at least one* of them. Because the default implementation of **/request** is defined using **/SingleRequest**, most subclassers of ClassSelection can simply define **/SingleRequest** to meet their needs.

In this section only **/SingleRequest** is discussed. For information on **/request** see page 177.

/SingleRequest

oldvalue request-key **/SingleRequest** newvalue

The key passed to a selection's **/SingleRequest** method identifies the nature of the request. Most keys represent requests for the selection to render its value in a named format (see Table 20-1). For these requests, the value currently on the stack (oldvalue) is to be discarded, and the client should put the requested value on the stack.

Certain keys request that the client modify the selection in some way. For these requests, the oldvalue on the stack contains arguments, if any, to be used in the operation. Even if no arguments are needed, the **/SingleRequest** method must be sure to remove the value from the stack; likewise, even if there is no return value, **/SingleRequest** must store something (typically 'null') on the stack. This ensures a uniform interface to **/SingleRequest**. (If neither the oldvalue nor the newvalue is meaningful, the oldvalue can simply be left on the stack as the returned value. See the /DeleteContents case in the example below.)

Note – In order to paste TNT selections into XView applications you *must* define /SelectionObjsize in your **/SingleRequest** override. See Table 20-1 on page 178 for a definition of /SelectionObjsize.

For any request (value or action), the client may choose not to support the requested key. If so, **/SingleRequest** should pop oldvalue and return /UnknownRequest.

A typical **/SingleRequest** method might look like:

```
/SingleRequest { % oldvalue request-key => newvalue
        {    /ContentsAscii      {pop Rank <get value> Holder send}
             /SelectionObjsize   {pop Rank <get value> Holder send length}
             /Level              {pop Level}
             /Canvas             {pop Holder}
             /DeleteContents     {Rank /DeleteSel Holder send} % leave junkval
             /Default            {pop /UnknownRequest}
        } case
        } def
```

Note that /**SingleRequest** need not support /RequestSequence or /RequestChoice. Since clients making such requests must always go through the /**request** method (rather than /**query**), it is left to the /**request** method to handle breaking up the /RequestSequence or /RequestChoice into a series of calls to /**SingleRequest**.

Utilities

Utility selection class

The NeWS Toolkit defines a subclass of particular interest to clients and/or implementors of selections. The subclasses is StringSelection.

A StringSelection only knows how to render itself as ContentsAscii. StringSelection's value never changes; it responds to queries by looking up the queried keys in a constant dict. This is to make it easy for clients to wrap a string inside a selection preparatory to handing it to a canvas via a /TransferSelection request. (See /**sendtocanvas** on page 188.) A StringSelection is intended to be created directly via **/new** instead of via the /**NewSelection** method, and thus does not expect rank/holder arguments; its /**new** method takes just a string.

Utility methods

/ComputeNamedPosition — first last current position-name /**ComputeNamedPosition** position

first, last, current, and position are numeric values referring to the location of a selection (in the client's interpretation); first/last are the endpoints of an existing selection, while current is typically the position corresponding to the coordinates of a recent event. position-name is one of the values defined for the Pin in a Selection, and is interpreted as follows:

/LowEnd	the low end of the existing selection (first)
/HighEnd	the high end of the existing selection (last)
/NearEnd	whichever of first/last is closer to current
/FarEnd	whichever of first/last is further from current
/AtPoint	the cursor position (current)

This is used for interpreting the Pin to establish one endpoint in preparation for subsequent /**SelectionAdjust** messages.

/computepin

first last current /**computepin** pin-point

If the Selection's Pin is a name, /**computepin** calls /**ComputeNamedPosition** and stores the result as the new value of Pin. Otherwise the three input values are discarded and the previously computed Pin value is returned unchanged. This should be done on every /**SelectionStart** or /**SelectionAdjust**; the UI manager will override Pin again if the anchor is to change.

/computerange

first last current /**computerange** newfirst newlast

Same as /**computepin**, but it returns the pinned position and the current position, in sorted order.

/CanRenderAs

- /**CanRenderAs** dictionary (Variable)

Determines which data-rendering request keys, e.g., /ContentsAscii (see Table 20-1 for a list of the keys) a selection responds to; the dictionary should not contain any of the action-generating keys (e.g., /DeleteContents). The dictionary is of the form: /request-key:any.

The cut and copy operations use /**CanRenderAs** to figure out what data formats the selection supports. The Toolkit's cut/copy code then queries the selection for each of those formats and stores the values on the clipboard.

Note – The list should NOT include /Canvas, even though the selection's /**SingleRequest** method might handle such a request. This is because copying a /Canvas value to the ClipBoard would result in the canvas staying on the screen after its application has been destroyed.

/sendtocanvas

canvas [delete?] /**sendtocanvas** -

Sends the selection to the given canvas if the canvas is receptible. The selection need not be registered with the global manager. The optional boolean says whether the selection should be deleted after the transfer. (Default is false.) The canvas can be null to send the event to the canvas(es) currently under the pointer. For example, the following would send a string to the current input focus:

```
(random string) /new StringSelection send
currentinputfocus /sendtocanvas 3 -1 roll send
```

clearselection

rank **clearselection** -

Removes the selection (if any) currently registered for the given rank from the global registry and deselects it. Defined in systemdict.

ClassSettings and ClassCheckBoxes

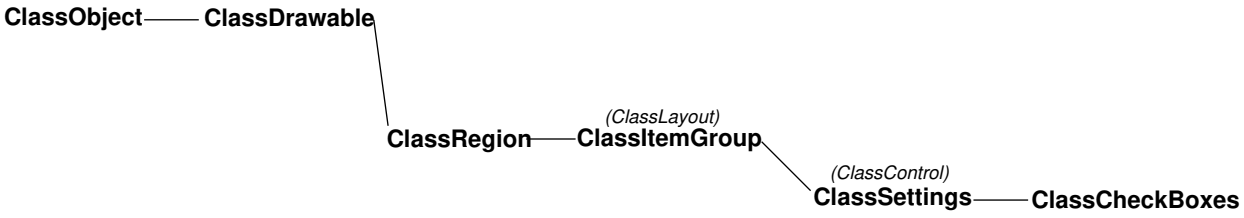


Figure 21-1 The ClassSettings and ClassCheckBoxes subtree

ClassSettings

ClassSettings is a single class implementing both exclusive and non-exclusive settings, including the special variation of exclusive settings which toggles the currently selected item on clicking on it a second time, thus allowing for no item being currently chosen. Settings can be: /Exclusive, /NonExclusive, or /ExclusiveVariation. The settings in a group are items and are added, deleted and accessed using the methods found in ClassItemGroup (see ClassItemGroup on page 87). Some of the ClassItemGroup methods inherited by ClassButtons are:

/setitemlist	/itemlist	/itemcount
/insertitem	/replaceitem	/deleteitem
/appenditem	/itemsize	/itemlocation
/itembbox	/pointinitem?	/pointtoitem

In addition, ClassControl is mixed-in to ClassSettings. The control interfaces documented in ClassSettings are: /setnotifier, /notifier, /setvisualstate /visualstate and the target interface.

Creation

Instances and items follow the item group interface exactly, overriding only **/NewItem** to store the display item and to initialize the "chosen" state to be false.

/new — placement parent /**new** instance

Creates a new instance of ClassSettings. Placement is one of /Spaced, /Absolute, /Calculated or /Grid.

Justification

/setjustification — /Left | /Centered /**setjustification** -

Sets the justification of the settings.

/justification — - /**justification** /Left | /Centered

Returns the settings justification.

Choices

/setchoicemode — /Exclusive | /NonExclusive | /ExclusiveVariation /**setchoicemode** -

Sets the style of the setting. You should call /**setchoicemode** immediately after /**new**, and before adding any items to the setting.

/choicemode — **-** /**choicemode** /Exclusive | /NonExclusive | /ExclusiveVariation

Returns the style of the setting.

The default is /NonExclusive.

/chosen? — item-index /**chosen?** boolean

Indicates whether the item at item-index is chosen.

Value of settings

The "value" of a setting is an array of the chosen items' indices. No special case is made for exclusive choices.

/setvalue — [item-index item-index . . .] /**setvalue** -

Sets the items whose indices appear in the array as the current set of chosen items.

For settings of mode /Exclusive this array must have one and only one entry.

For settings of mode /ExclusiveVariation the array may have either one entry or be an empty array.

/value — - /**value** [item-index item-index]

Returns an array of indices of the chosen items.

Notification

The notification value for settings and checkboxes is an array that contains:

[item-index boolean]

boolean indicates whether the item has been chosen or "unchosen."

Different types of settings notify at different user actions:

/Exclusive	notifies *only* when an item is turned on. This means you are not notified about the item being unchosen, and the boolean in the array is always true.
/ExclusiveVariation	notifies when an item is turned on as /Exclusive settings do and notifies when the item currently turned on is turned off and the setting has no chosen item.
/NonExclusive	notifies when any item is turned on or off.

/setnotifier — notifier | null /**setnotifier** -

Sets the notifier for the settings. When notifier is specified as a PostScript name type, it is used in conjunction with the target interfaces to dispatch notification to the appropriate target. During notification, the notification value, the setting instance, and the notifier name are pushed on the stack, prior to invoking /**sendtarget**. (see below).

Although not recommended, notifier can also be specified as a PostScript code fragment. The current value, the control instance, and the fragment are pushed on the stack, then the fragment is executed.

/notifier — - /**notifier** notifier | null

Returns the notifier for the settings.

/setpreviewer — previewer | null /**setpreviewer** -

Sets the previewer for settings. If null is specified the previewer is set to null, i.e., previewing is turned off.

/previewer — - /**previewer** previewer | null

Returns the settings previewer.

ClassCheckBoxes

ClassCheckBoxes, implements the special check box selection list by subclassing ClassSettings. Although it typically uses /Grid layout, it is not limited to that and may use /Calculated, /Absolute, or /Spaced layouts. Only /NonExclusive check boxes are supported.

Creation

/new — placement parent /**new** instance

Creates a new instance of ClassCheckBoxes. placement is one of /Spaced, /Absolute, /Calculated or /Grid

Sliders

Figure 22-1 The slider subtree

The NeWS Toolkit implements OPEN LOOK horizontal sliders by ClassHSlider and OPEN LOOK vertical sliders by ClassVSlider.

A slider controls a numerical value bounded by minimum and maximum values. A user may affect the value of the slider in the following ways:

- moving the drag box to the desired location to set an absolute value.
- clicking on the end boxes to select the minimum or maximum values.
- clicking to the left or the right of the drag box to change the value by a relative delta.

Values are also constrained to be between the minimum and maximum values, after being normalized.

Tick marks can be drawn at regular intervals below a horizontal slider, or alongside a vertical one, by specifying the distance in slider value units between consecutive tick marks. If you don't want tick marks, you can specify a distance of 0, which is the default.

Creation

/new — parentcanvas /**new** instance

Creates a new slider instance. Sent to ClassHSlider to create a horizontal slider and ClassVSlider to create a vertical slider.

Slider auto repeat

You can control when the slider starts to repeat and how fast it repeats by defining two variables in your UserProfile. If you don't define values for the variables, the Toolkit uses default values.

/SliderThresh — /**SliderThresh** milliseconds (UserProfile Variable)

Determines how many milliseconds SELECT has to be down to either side of the drag box before the slider begins repeating. Default is 500 milliseconds

/SliderDelay — /**SliderDelay** milliseconds (UserProfile Variable)

Determines how many milliseconds delay there is between slider repeats. Default is 20 milliseconds.

Values

/setvalue — value /**setvalue** -

Sets the value of the slider. Any number can be given to /**setvalue**; it is normalized and forced in range. The slider is painted to reflect the new value if the slider is valid.

/value — - /**value** value

Returns the value of the slider. The number returned by /**value** is always normalized and in range.

/setrange — minimum-value maximum-value /**setrange** -

Sets the minimum and maximum values of the slider. Invalidates the slider. The slider is updated to reflect the new range. You should reshape the slider to its preferredsize after changing the range if you want the slider to be large enough to select every whole number in its range. The default range is 0 to 100.

/range - /**range** minimum-value maximum-value

Returns the minimum and maximum values of the slider.

Notification and previewing

/setnotifier notifier | null /**setnotifier** -

Sets the notifier in the slider. When notifier is specified as a PostScript name type, it is used in conjunction with the target interfaces to dispatch notification to the appropriate target. During notification, the current value, the slider instance, and the notifier name are pushed on the stack, prior to invoking /**sendtarget**. In other words, your notifier should be written to take a value and an instance as arguments. Using null makes the notifier a no-op, i.e., it turns off notification. The default is for the notifier to be null.

Although not recommended, notifier can also be specified as a PostScript code fragment. The current value, the control instance, and the fragment are pushed on the stack, then the fragment is executed.

/notifier - /**notifier** notifier

Returns the notifier for the slider.

/setpreviewer previewer | null /**setpreviewer** -

Sets the previewer procedure for the slider. null removes the previewer from the slider. Previewing involves returning intermediate values before getting the final value, e.g., dragging the sliders drag box could preview.

/previewer - /**previewer** previewer

Returns the slider's previewer procedure.

Target Interface

/settarget object /**settarget** -

Sets object as the target of the slider's notifier. If a previous target exists it is overwritten.

/cleartarget — null | object /**cleartarget** -

Clears the target. If null is given the target is cleared. If object is specified then the target is cleared only if object and the target are the same. This latter specification ensures that the target is not incorrectly cleared.

/sendtarget — arguments /method /**sendtarget** results

Sends /method and any required arguments to the target.

/target — - /**target** null | object

Returns the target.

Granularity

Slider granularity is controlled by a normalizer procedure. The normalizer procedure consumes the unconstrained value on the stack and leaves a constrained value on the stack.

/setnormalizer — proc /**setnormalizer** -

Sets the slider's normalization procedure. Invalidates the slider.

/normalizer — - /**normalizer** proc

Returns the procedure that normalizes the value of the slider.

A typical normalizer would be "{round cvi}", to constrain the value to rounded integers. By default the normalizer does not change the value.

/setdelta — step /**setdelta** -

Sets the slider's step delta. step is the amount the slider's value changes when the user presses POINT to the left or right of the drag box. To the left of the drag box the slider's value is decremented by step; to the right of the drag box the slider's value is incremented by step. The default is 1.

/delta — - /**delta** step

Returns the slider's step delta.

Visual presentation

/settickmarks

distance /**settickmarks** -

Specifies that you want tick marks drawn below or alongside the slider. Invalidates the slider. distance is the space, in slider value units, between consecutive tick marks. 0 specifies no tick marks. “Slider value units” requires some explanation. For example, if a slider has a value that ranges from 0 to 100 then it has 101 slider value units.

/tickmarks

- /**tickmarks** distance

Returns the distance (in slider value units) between the tickmarks. Returns 0 if the slider has no tick marks.

/setendboxes

boolean /**setendboxes** -

Determines whether the slider has endboxes. The default is for no endboxes. Invalidates the slider. You should reshape the slider to its preferred size after changing the endboxes.

/endboxes?

- /**endboxes?** boolean

Returns whether the slider has endboxes.

Geometry

/minsize

- /**minsize** width height

Returns the minimum size of the slider.

/preferredsize

- /**preferredsize** width height

Returns a size that makes the slider larger enough to select every whole number in its range.

Slider label positioning

/offset

name /**offset** x y

Returns the (x,y) offset from the slider’s lower left corner to the point where a label should be justified.

name is one of:

/MinEnd	At left end of hslider, or bottom of vslider.
/MaxEnd	At right end of hslider, or top of vslider.
/MinTick	At left end below hslider, or top to right of vslider.
/MaxTick	At right end below hslider, or bottom to right of vslider.

The offsets are useful when positioning the slider's minimum and maximum labels and labels for the tick marks, if any. Table 22-1 shows the various ways you could use offsets to position your slider labels using calculated layout and the slider offsets. Figure 22-2 shows the resulting slider.

Table 22-1 Code using offsets for calculated layout of horizontal slider labels

Label	Label Offset name	Slider Offset name	Slider Client name		Gap offset calculation
MinEnd	[/East	{ /MinEnd	/hslider	POSITION	0 5 xysub }]
MaxEnd	[/West	{ /MaxEnd	/hslider	POSITION	0 5 xyadd }]
MinTick	[/North	{ /MinTick	/hslider	POSITION	5 0 xysub }]
MaxTick	[/North	{ /MaxTick	/hslider	POSITION	5 0 xysub }]

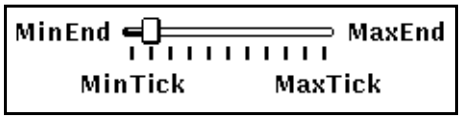


Figure 22-2 Horizontal slider with its labels positioned using slider offsets

Similarly you could use the slider offsets to layout a vertical slider (Table 22-2, Figure 22-3). The label offset names change to reflect the change in orientation and the gap offset calculation changes slightly but otherwise the calculations are the same.

Table 22-2 Code using offsets for calculated layout of vertical slider labels

Label	Label Offset name	Slider Offset name	Slider Client name		Gap offset calculation
MinEnd	[/North	{ /MinEnd	/vslider	POSITION	5 0 xysub }]
MaxEnd	[/South	{ /MaxEnd	/vslider	POSITION	5 0 xyadd }]
MinTick	[/West	{ /MinTick	/vslider	POSITION	0 5 xyadd }]
MaxTick	[/West	{ /MaxTick	/vslider	POSITION	0 5 xyadd }]

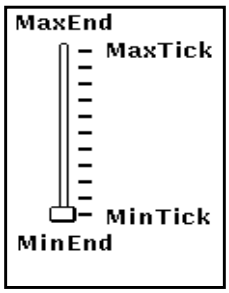


Figure 22-3 Vertical slider with its labels positioned using slider offsets.

ClassTextCanvas

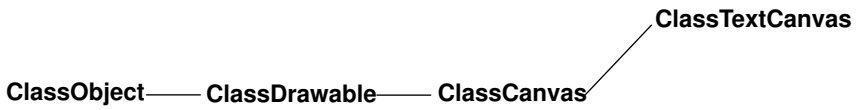


Figure 23-1 The ClassTextCanvas subtree

ClassTextCanvas subclasses ClassCanvas to provide assistance for clients whose selections are character strings, and who want to use an overlay canvas to display the selection during a drag-move or drag-copy operation. It fills in the **/DragStart** and **/DragAdjust** methods for you. You still must provide the definitions for the other selectable subclass responsibility methods in ClassCanvas (see *Selectables* on page 65 in ClassCanvas).

/DragStart — event selection **/DragStart** -

Creates an overlay canvas (see the *NeWS 2.1 Programmer's Guide* for information on overlay canvases).

/DragAdjust — event selection **/DragAdjust** -

Called when the cursor is moved (as long as SELECT stays pressed); it causes the overlay canvas to follow the cursor and displays some of the text that was displayed in the window when dragging began.

/DragStop — event selection **/DragStop** -

Destroys the overlay canvas. Sent to the text canvas when SELECT is released.

/CurrentText

selection /**CurrentText** string

Obtains the text to display in the overlay canvas. Subclassers will generally wish to override the /**CurrentText** method for greater efficiency (the default uses the normal /**query** mechanism (see /**query** page 174 in ClassSelection), whereas individual subclasses can usually obtain the text by more direct methods).

ClassTextField

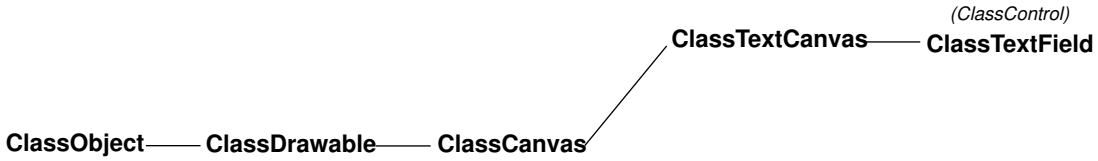


Figure 24-1 The ClassTextField subtree

Creation

/new — parentcanvas /**new** instance

Creates a textfield instance. Unlike some controls, text fields are not "grouped" and so do not have a placement parameter.

State of Text Fields

/setvisualstate — state /**setvisualstate** -

Sets the state of the field to one of /Active, /Inactive, or /Busy. The field is not painted unless it is valid.

Active fields accept the focus, drag and drops, and selections.

Inactive fields are dimmed, and do not accept the input focus, drag and drops, or selections. The text can still be modified programmatically via calls to **/inserttext**, **/setvalue**, etc.

Busy fields do not accept input, drag and drops, or selections.

/visualstate — - **/visualstate** state

Returns the state of the field. State is one of /Active, /Inactive, or /Busy (see **/setvisualstate**).

/ReadOnly? — **/ReadOnly?** boolean (Variable)

Determines if the text field is read only. A read only text field does not display a caret and does not accept the input focus or drag and drops. However, it displays normally (i.e., not dimmed) and the text in it can be selected. As with inactive fields, the text can still be modified programmatically via calls to **/inserttext**, **/setvalue**, etc.

/setreadonly — boolean **/setreadonly** -

Sets the value of the variable **/ReadOnly?**.

/readonly? — - **/readonly?** boolean

Returns the value of the variable **/ReadOnly?** (see **/ReadOnly?** for an explanation).

Value of Text Fields

/setvalue — string **/setvalue** -

Replaces the entire contents of the text field with the given string. Any selections in the old text are cleared, and the caret is placed at the end of the new text. The string is copied, so subsequent changes to the original string or to the text field's value do not affect each other. The field is not painted unless it is valid.

To clear the text completely, use the global constant "nullstring", i.e.,

```
nullstring /setvalue mytextfield send.
```

/value - /**value** string

Returns the current contents of the text field. Clients who intend to modify the contents of the string (via the PostScript operators **put** or **putinterval**) must instead make their own copy of the string.

Characters

/characters - /**characters** number-of-characters

Returns the number of characters in the text. Equivalent to, but much faster than

```
/value mytextfield send length.
```

/MinimumVisible /**MinimumVisible** number-of-characters (Variable)

Determines the minimum number of characters to display in the field. Setting /**MinimumVisible** does *not* automatically reshape the text field; all it does is change the value returned by /**minsize**. Default setting is 5 (see /**minsize**).

/setminimumvisible number-of-characters /**setminimumvisible** -

Sets the minimum number of characters to display.

/minimumvisible - /**minimumvisible** number-of-characters

Returns the minimum number of characters to display.

/minsize - /**minsize** width height

Specifies a width for the text field large enough to fit /**MinimumVisible** number of capital-M characters in the text field's font, plus the scrolling buttons.

Notification

The Toolkit's text fields notify *only* on TAB or RETURN (or LINEFEED), *not* on loss of focus. This means that, unlike for other controls, the client must explicitly obtain the current text if it needs to be certain of having the most recent value, such as when performing an "Apply" function.

This notification scheme can create a problem for an application if it assumes that it will be notified when a text field is changed. If a user makes changes to a textfield and then types ^N (or ^P) to move to another object, no notification will occur. You should use the notifier to perform some specific action, like doing a "Load File."

/setnotifier — notifier /**setnotifier** -

Sets the text field's notifier. notifier is either a method name that is sent to the target (as set by /**settarget**), or it is a PostScript code fragment. (The latter is not recommended but is permitted for those clients who need the generality or efficiency.) See *Notification and previewing* on page 77 in ClassControl.

/notifier — - /**notifier** notifier

Returns the text field's notifier.

/setpreviewer — previewer /**setpreviewer** -

Sets the previewer. Like the notifier (above), the previewer is either a method name that is sent to the target, or it is a PostScript code fragment. It can also be null to obtain default behavior. Setting the previewer allows a client to respond to each character as it is typed into the text field, unlike the notifier which is called only when the user is finished. See /**insertcharacter** for more details.

/previewer — - /**previewer** previewer

Returns the text field's previewer.

The text insertion point (the caret)

/setcaret — caret-position /**setcaret** -

Sets the position of the caret within the text. The caret position determines the location for inserting characters, as well as for various other operations. The caret goes between two characters. If caret-position is 0 the caret is placed to the left of the first character. If the position exceeds the length of the text, an error will result.

/caret — - /**caret** caret-position

Returns the current caret position. The number returned is the number of characters to the left of the caret, including characters not visible due to having scrolled off the left edge of the field.

Painting

/PaintText

character-number /**PaintText** -

Paints the text, starting at character-number. If character-number is less than 0, /**PaintText** paints all the text and also clears the rest of the canvas. The scrolling buttons are also repainted as necessary. Assumes that the canvas and font are set.

Note – /**PaintText** is a utility used to write other methods. Most clients will call the ClassCanvas /**paint** method, which ends up doing a -1 /PaintText self send.

Manipulating the text

/deletecharacters

number-of-characters /**deletecharacters** -

Deletes one or more characters from the text, starting at the caret. If number-of-characters is negative, deletes characters preceding the caret; otherwise /**deletecharacters** deletes after the caret. (If number-of-characters is zero, the text is unchanged.)

/deletespan

1st-character-position number-of-characters /**deletespan**-

Subclasser method: /DeleteSpan

Deletes an arbitrary span of characters, starting after 1st-character-position and extending for number-of-characters. If 1st-character-position + number-of-characters exceeds the length of the text, an error may occur. Any selections within the text are adjusted accordingly. For example:

(Sample Text) /setvalue txt send 4 5 /deletespan txt send /value txt send == (Sampxt)

/DeleteSpan

1st-character-position number-of-characters /**DeleteSpan** -

Deletes the span of characters defined by 1st-character-position and number-of-characters. It assumes the canvas and the font have been set. It does not do any painting, but just updates the text value and adjusts the selections.

/deletewords

number-of-words /**deletewords** -

Deletes number-of-words, starting at the caret. If number-of-words is negative, deletes words preceding the caret; otherwise deletes words after the caret. To delete each word, characters are deleted until (a) at least one alphanumeric has been

deleted and (b) the next character is not alphanumeric, or until the caret reaches the start of the text (or end, if deleting forward). The **/AlphaNumeric?** method is used to test individual characters.

Example:

```
(Here, so they say, is some sample text.) /setvalue txt send 11 /setcaret txt send
% puts it after the “th” of “they”
2 /deletewords txt send /value txt send == (Here, so th, is some sample text.)
```

/insertcharacter

character **/insertcharacter** -

Inserts a single character into the text at the current caret location. If the caret is within (or at one end of) a pending-delete primary selection, the selected text is deleted.

If the previewer is non-null (see **/setpreviewer**), then **/insertcharacter** does not modify the text. Instead it sends the previewer method to the target, or executes the previewer code fragment. The text field instance is first pushed onto the stack; the previewer should consume both the instance and the character. It is up to the previewer to make any desired modifications to the text, e.g. by calling **/insertstring**. NOTE: The previewer must not call **/insertcharacter**, because this will likely cause infinite recursion.

If the previewer is null, the character given to **/insertcharacter** is matched against a dictionary of special characters (see **/SpecialActions**); characters found in that dict perform the special action and are not inserted (and do not delete the selection).

/insertstring

string **/insertstring** -

Subclasser Method: /InsertString

Inserts a string of characters into the text at the current caret location. If the caret is within (or at one end of) a pending-delete primary selection, the selected text is deleted.

Characters in the string given to **/insertstring** are NOT checked for special actions and do NOT go through the previewer. Ordinary type-in uses **/insertcharacter**; text received via cut and paste or drag and drop uses **/insertstring**.

/InsertString

string /**InsertString** -

Performs the actual insertion of the text. It does NOT do any painting. /**InsertString** assumes that the canvas and the font have been set, that the caret has been turned off, and the pending-delete selection has been dealt with. It also assumes that the caller will take care of scrolling the text if the position where the caret is to be restored turns out to be off the edge of the canvas.

Note – The /**insertcharacter** operation is actually quite different from Jot's /**insertcharacters**. The Jot method inserts multiple characters specified as a buffer and length (as distinct from a null-terminated string). PostScript makes no distinction for null-terminated strings, so /**insertstring** handles all multi-character cases. Jot does not have a single-character /**insert** method.

/settextfont

font /**settextfont** -

Sets the font of the textfield. Invalidates the field.

Selections

/setselection

1st-character-position number-of-characters /**setselection** -

Causes the specified range of characters to become the primary selection. The selection is single-click level, pending delete. "Single-click" means that if the user subsequently clicks ADJUST, the selection adjusts by characters, not words. Pending delete means if the user types, the selection is deleted. OPEN LOOK specifies that selections are pending-delete.

/selection

- /**selection** false|1st-character-position number-of-characters true

Returns the indices of the selected range and true if the text field currently contains the primary selection; otherwise it returns false.

Moving between textfields and other textfields or canvases

Text fields support the notion of "where to go next"; i.e., where should the input focus go when the user presses RETURN or some other key (per the OPENLOOK specification). Usually the next focus is another text field, but it can be any Keyable canvas. (E.g., in Mail Tool, pressing RETURN in the composition window eventually puts the focus into the message text area.)

/setnextfocus

canvas /**setnextfocus** -

Sets the canvas that gets the focus from the text field. /setnextfocus doesn't actually set the focus but just specifies the focus order. It's generally called when you are creating a control panel, not in a notifier. To specify that the focus should not move out of the text field when RETURN is pressed, call /**setnextfocus** with 'null'. The default is null.

/nextfocus

- /**nextfocus** canvas

Returns the canvas that was set by /**setnextfocus**.

/previousfocus

- /**previousfocus** canvas

Returns the text field for which /**nextfocus** is this one. (If more than one text field calls /**setnextfocus** with the same destination, the results are undefined.)

/gotonextfocus

- /**gotonextfocus** -

Moves the focus to the canvas specified via /**setnextfocus** after calling the notifier (if any) with the current value. If /**nextfocus** is null, the input focus is not moved, but the notifier is still called.

/gotonextfield

- /**gotonextfield** -

Moves the focus to the text field specified in /**setnextfocus**. If some other type of canvas (i.e., not a text field) was specified in /**setnextfocus**, the focus goes to the first text field in the chain set up by sequential calls to /**setnextfocus**. I.e., the focus wraps around from the last field back to the first. Does not call the text field's notifier.

/gotopreviousfield

- /gotopreviousfield -

Moves the focus to the text field that is returned by **/previousfocus**. If there is no such text field the focus is moved to the last text field in a chain set up by sequential calls to **/setnextfocus**. I.e., the focus wraps around from the first field back to the last.

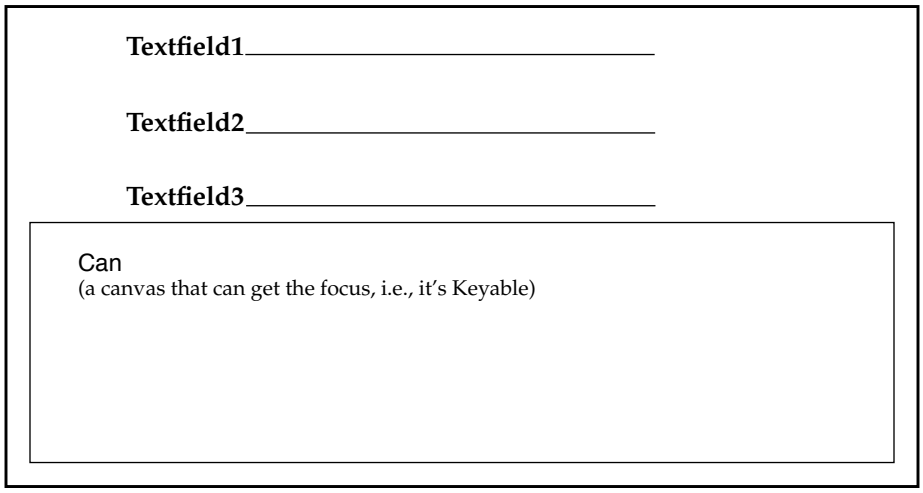


Figure 24-2 Moving the focus between text fields.

Figure 24-2 illustrates a sample application that has linked text fields so the focus can be moved between textfields. You could establish the chain of focus passing by doing the following:

```
Textfield2 /setnextfocus Textfield1 send
Textfield3 /setnextfocus Textfield2 send
Can /setnextfocus Textfield3 send
```

With this chain established if **/gotonextfocus** is called when Textfield3 has the focus the focus moves to Can. However, if **/gotonextfield** is called when Textfield3 has the focus then the focus moves to Textfield1 (i.e., the focus "wraps"). Similarly if Textfield1 has the focus and you call **/gotopreviousfocus**, the focus "wraps" around to Textfield3.

Scrolling

/scroll

number-of-characters /**scroll** -

Subclasser method: /Scroll

Scrolls the text left or right, as number-of-characters is positive or negative, respectively. If you try to scroll right and the first character is already visible, no scrolling occurs; it does not cause an error. Similarly trying to scroll left while no character is visible, does not cause an error.

/Scroll

number-of-characters /**Scroll** -

Works the same as /**scroll** except it does not perform any painting. It is used internally by various methods that may perform several scrolls before updating the screen. The /**scroll** method also uses /**Scroll**.

/fitcaret

- /**fitcaret** -

Subclasser method: /**FitCaret**

Scrolls the text left or right as necessary to cause the current caret position to be within the visible portion of the text. Usually you do not need to call this method explicitly, because other methods (/**insertcharacter**, /**deletespan** and /**setposition**) scroll the text automatically.

If the caret is already visible, no scrolling occurs. Otherwise the text scrolls so as to place the caret at approximately the position specified by /**AutoScrollPosition**.

/FitCaret

/method /**FitCaret** boolean

Adjusts the text field's left margin as necessary, but without doing any painting, and returns true if any scrolling was performed (in which case the caller is expected to do a full repaint instead of whatever partial repaint might otherwise have been required). The method name given to /**FitCaret** is one of: /**fitcaret**, /**setposition**, /**deletespan**, and /**insertstring**; the default implementation ignores it, but a subclasser might choose to scroll differently (or not at all) after certain operations.

/AutoScrollPosition — **- /AutoScrollPosition** position (Variable)

Determines where the caret is placed, by **/FitCaret**. It is a fraction between 0 (to put the caret at the left edge of the field) and 1 (to put it at the right). The default value is 2/3. This value is not expected to be changed "on the fly," but only via subclassing (or UserProfile overrides), so there are no accessor methods.

/InvisibleCaret — **- /InvisibleCaret -**

Makes the caret so that it is not currently painted, blinking, or otherwise active. Assumes the canvas and font are set.

/VisibleCaret — boolean **/VisibleCaret -**

Causes the caret to become visible at its current position. Assumes the canvas and font are set. If boolean is true, the caret is activated immediately. Otherwise an event is queued that causes the caret to become visible after a short delay (determined by **/CaretDelay**) unless additional /*Caret calls occur.[1]

/CaretDelay — **/CaretDelay** milliseconds (Variable)

Determines how long the text field waits to make the caret visible/Active. The default is 100 milliseconds.

1. Implementation note: The reason for the delay is to avoid spending time painting and unpainting the caret when the user is typing many characters. Eliminating this extra painting saves a significant amount of processor overhead.

/ResolveToChar

event /**ResolveToChar** caret-position

Returns the caret position nearest to the coordinates of the event. Caret positions are between characters; the position returned is before/after the character at the coordinates as the coordinates are in the left/right half of the character (see Figure 24-3).

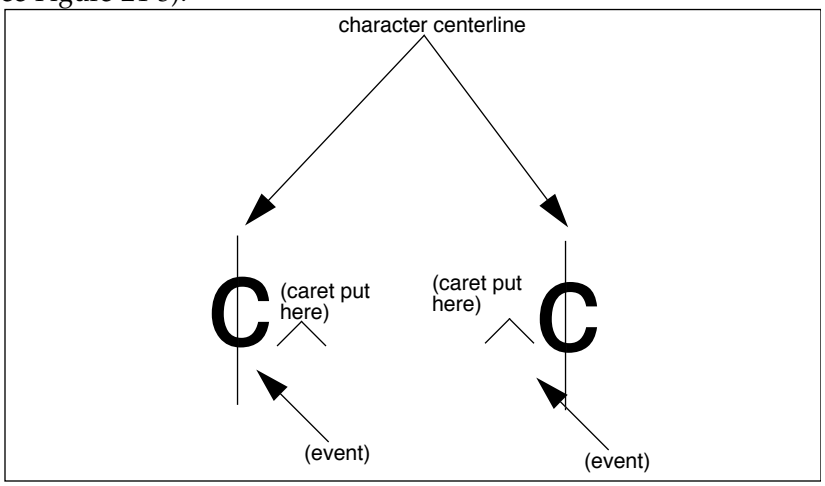


Figure 24-3 How the caret position is resolved.

/AlphaNumeric?

character-position /**AlphaNumeric?** boolean

Returns "true" if the character at the specified position is part of a word. The default method returns true for A-Z, a-z, 0-9, and underscore. Subclassers may override this to change the behavior of word-selection and word-deletion operations. See also /**AlphaNumericTable**, below.

/AlphaNumericTable

/**AlphaNumericTable** dictionary (Variable)

Determines which characters can be part of a word; used by /**AlphaNumeric?**. It is a dict whose keys are the alphanumeric characters (with null values).

If you want to modify this dictionary to change the behavior of the /**AlphaNumeric?** method for an instance or a subclass, you must be careful to make a copy of it and modify the copy. If you simply define new keys into the existing dict, it affects *all* text fields.

/SpecialActions

/SpecialActions dictionary (Variable)

Defines the special actions performed by **/insertcharacter** in response to certain characters. As with **/AlphaNumericTable**, subclassers who wish to modify this dict should make a copy of it. The default set of special actions is:

Table 24-1 The keys and values of the /SpecialActions dictionary.

Dict Key	Dict Value	Keyboard Key	Explanation
1	{0 setcaret} def	^A	move to beginning
2	{caret 1 sub 0 max setcaret} def	^B	move backward
4	{1 deletecharacters} def	^D	delete forward
5	{characters setcaret} def	^E	move to end
6	{caret 1 add characters min setcaret} def	^F	move forward
8	{-1 deletecharacters} def	^H	delete backward
9	{gotonextfocus} def	TAB	go to next focus canvas
10	{gotonextfocus} def	LF	go to next focus canvas
13	{gotonextfocus} def	CR	go to next focus canvas
14	{gotonextfield} def	^N	go to next field
16	{gotopreviousfield} def	^P	go to previous field
21	{caret neg deletecharacters}def	^U	delete to beginning
23	{-1 deletewords} def	^W	delete word
127	{-1 deletecharacters} def	% DEL	delete backward

Miscellaneous

/movebaseline

x y **/movebaseline** -

Moves the text field (within its parent canvas) so as to place the left edge of the baseline at the given coordinates. This method is intended for application writers who have raw text painted on a canvas and want to replace that text with a text field with minimal visual disturbance.

For example the following two lines of code:

```
/Parent yourtextfield send setcanvas x y moveto (some text) show
```

```
(some text) /setvalue yourtextfield send x y /movebaseline yourtextfield send
```

positions the text “some text” at the same position on the screen.

/reshape

x y width height /**reshape** -

Reshapes the field to fit the bounding box defined by the arguments. Invalidates the field.

invalidate

-/**invalidate** -

Invalidates the text field. You can use /**invalidate** to control how many times the field paints when you are changing both its value and its visual state. Both /**setvalue** (page 204) and /**setvisualstate** (page 203) only repaint the field if it’s valid. So, if you’re planning to call /**setvalue** and /**setvisualstate** for a text field to avoid having it repaint twice you could call:

/**invalidate** (marks field as invalid)

/**setvalue** (sets the value, field is invalid, it doesn’t paint)

/**setvisualstate** (sets state, field is still invalid, it doesn’t paint)

/**paint** (paints and validates the field)

in that order.

Windows

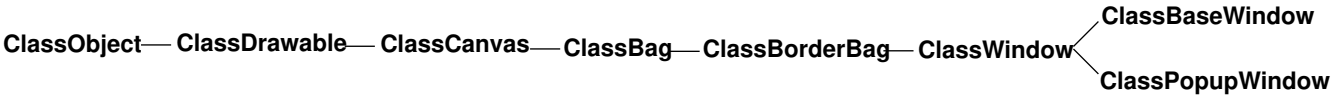


Figure 25-1 The windows subtree

ClassWindow

A window is a descendent of ClassBorderBag which defaults to having a single "client" canvas. (It is the /Center client of the borderbag, see Chapter 3, ClassBorderBag.) The client canvas operates like any other ClassCanvas object: it knows how to /paint, /fix, /reshape, /move itself.

A window also has five standard "attributes" that determine its style (Table 25-1).

Table 25-1 Window attributes and their associated class variables

Window Variable	Explanation
/Close?	True to display the close button
/Footer?	True to display a footer area
/Label?	True to display a label area
/Pin?	True to display a pin

/Reshape? True to display reshape corners

Each attribute is associated with a window decoration that may be an active area.

While some combinations of attributes are not explicitly errors, they give rise to undefined behavior. For example the pin and close button are displayed in the same part of the window. If you set both /Pin? and /Close? to true both are displayed but the one that notices the mouse down first is undefined.

Table 25-2 Default control usage by OPEN LOOK

Window Type	/Label?	/Footer?	/ReShape?	/Close?	/Pin?	Menu
Base Window	y	o	o	y	n	y†
Property Window	y	o	o	n	y	y††
Command Window	y	o	o	n	y	y††
Help Window	y	n	o	n	y	y††
Notice Window[1]	n	n	n	n	n	n
Icon Window	y	n	n	n	n	y†

Legend: y: yes n:no o:optional †Uses base window's menu; ,†† uses pop-up window's menu

[1] Notices are not actually a subclass of ClassWindow and are included here only to complete the OPEN LOOK model.

Creation and initialization

/new client | null parent /**new** instance

Creates an instance of ClassWindow. If client is non-null, it becomes the central client (i.e., it's name is /Center) in the border bag (see ClassBorderBag).

/setattribute attribute-name boolean /**setattribute** -

Determines whether a window has the named attribute. name is one of: /Close?, /Footer?, /Label?, /Pin?, or /Reshape?. These are initially class variables, promoted by /**setattribute**. The attributes default to false in ClassWindow, and are overridden to OPEN LOOK defaults by ClassBaseWindow and ClassPopupWindow.

Your window can have a label area and/or a footer area but not have anything displayed in them. /**setattribute** can be seen as changing a window's structure (by allocating part of the window for a label and/or a footer) but says nothing about the label's or the footer's contents. For example, if you add a footer area by doing:

```
/Footer? true /setattribute win send
```

you have altered the structure of the window to have an area at the bottom of the window where you can put display items. In order to have anything actually displayed in the footer you have to use /**setfooter**. /**setfooter** can be seen as putting some content into the footer structure. Thus, if you want a window to have a footer that is sometimes empty you should do:

```
nullstring nullstring /setfooter win send
```

instead of:

```
/Footer? false /setattribute win send
```

/**setattribute** invalidates the window. (See Validation on page 44 in ClassCanvas.)

/attribute — name /**attribute** boolean

Returns whether name (a window attribute) is present in the window.

For example, to see if there is a label area allocated:

```
/Label? /attribute win send
```

Labels and footers

Windows can have header and footer areas into which the application can put arbitrary display items.

/setlabel — display-item /**setlabel** -

Sets the label to be the given display item.

For example, to set the label to be "Hi World!" in blue:

```
[(Hi World!) 0 0 1 rgbcolor] /setlabel win send
```

/label — - /**label** display-item

Returns the current label.

/setfooter — left right /**setfooter** -

Sets the left and right footer display items.

/footer — - /**footer** left right

Returns the left and right footer display items.

Painting

There are also two public window painting methods, /**paintwindow** and /**flashframe**.

/flashframe — - /**flashframe** -

Changes the window's appearance, then back to the current appearance. Use /**flashframe** to draw attention to the window. Uses /**paintwindow**.

/paintwindow — - /**paintwindow** -

Paints just the window; does not paint the window's client.

/TextFont — /**TextFont** font (Variable)

Font used by the window label.

/FooterFont — /**FooterFont** font (Variable)

Font used by the window footer.

Freezing windows

Windows may be "frozen" for input when in an error situation. You freeze and unfreeze a window using /setvisualstate. This allows an application in an error state to ignore input until the error condition is handled. See /**FreezeFilter** below.

/Freezable? — /**Freezable?** boolean (Variable)

Determines whether the window can be frozen. Windows are Freezable by default.

/setvisualstate — state /**setvisualstate** -

Sets the visual presentation of the window. state can be one of /Active, /Inactive, /Busy. /Active is the default. /**setvisualstate** also determines whether users can interact with the window. Users can interact with /Active windows; /Inactive and /Busy windows ignore user interactions, i.e., they are frozen.

/visualstate **-** /**visualstate** state

Returns the state of the window.

/FreezeFilter /**FreezeFilter** dictionary (Variable)

Dictionary used to determine what should happen when the window gets an event when it is /Busy (frozen).The dictionary consists of event name-procedure pairs used to process the events of a frozen window. Any event name not found in the dictionary uses the /Default entry, which ignores the event. The procedures are executed when a key matches an event. You may supply your own custom FreezeFilter by defining one as either a class variable or an instance variable. The standard filter is:

```
/FreezeFilter dictbegin
    /Damaged {redistributeevent} def
    /LoseFocus {redistributeevent} def
    /LoseSelection {redistributeevent} def
    /Default {pop} def
dictend def
```

The standard filter allows the window to repaint damaged areas if new portions are exposed, and to respond to loss (but not acquisition) of focus and selections.

Window placement

Windows implement a simple default placement policy for bringing up windows for which the user has specified no explicit location.

/place - /**place** -

Positions the window at the next location in the placement scheme. If the window is already reshaped, /**place** uses its current size, otherwise it uses the window's preferredsize.

Subwindows

Windows may have nested subwindows. These close together, and if closed together, open together. Generally subwindows are popups managed by base windows. The subwindow tree is bidirectional: the subwindow has a SuperWindow reference, and the super window has an array of subwindows.

Windows have no accessor methods for subwindows; you must keep a handle to any subwindows you might want to access.

/SubWindows — /**Subwindows** array (Variable)

The window's subwindows.

/SubWindows — /**Subwindows** array (Variable)

The window's subwindows.

/subwindows — - /**subwindows** [subwindow1 subwindow2 . . .]

Returns the value of /SubWindows, i.e., an array containing the window's subwindows.

/addsubwindow — subwindow /**addsubwindow** -

Adds a subwindow to this window. If the subwindows is not active, it is activate using the superwindow's event manager.

/removesubwindow — subwindow /**removesubwindow** -

Removes a subwindow from this window. Deactivates the subwindow if it has the same event manager as the superwindow.

/opensubwindows — - /**opensubwindows** -

Opens the subwindows previously closed by this window.

/closesubwindows — **-** /**closesubwindows** -

Closes the subwindows currently open and marks them as having been closed by this window.

/SuperWindow — /**SuperWindow** super-window (Variable)

The window's super window.

/RootWindow — **-** /**RootWindow** root-window (Variable)

The root window of this set of subwindows.

Viewing states

Three viewing states of windows are handled in a similar manner: pinned, opened, and zoomed. The interfaces are similar: /foo, /unfoo, /fooed?, and /togglefoo.

Opening and closing windows

/open - /**open** -

Opens the window, does nothing if the window is already open. The opened state is an abstract notion that for base windows means non-iconic, and for other windows means mapped. If /**open** is sent to a base window, any subwindows that the window may have previously closed are opened.

/close - /**close** -

Closes all open windows, does nothing if the window is already closed. If /**close** is sent to a base window, it closes opened subwindows.

/opened? - /**opened?** boolean

Returns whether the window is mapped.

/toggleopened - /**toggleopened** -

Reverses the opened state of the window.

Pinning windows

/pin - /**pin** -

Pins the window.

/pinned? - /**pinned?** boolean

Returns whether the window is pinned.

/togglepinned - /**togglepinned** -

Reverses the pinned state of the window.

/unpin - /**unpin** -

Unpins the window. Causes the window to close itself.

Zooming windows

/zoom — **- /zoom -**

Zooms the window to full size. Remembers the unzoomed size.

/zoomed? — **- /zoomed?** boolean

Returns whether the window is zoomed.

/togglezoomed — **- /togglezoomed -**

Reverses the zoomed state of the window.

/unzoom — **- /unzoom -**

Reverts to unzoomed size.

Miscellaneous

/QuitFromUser — control | event /**QuitFromUser** -

Responds to a request to quit. When the user chooses "Quit" from a base window menu the message /**QuitFromUser** is sent to that window. The default implementation of /**QuitFromUser** provided by TNT kills your program. If you need to inform the client side that it is about to be killed, or you want to confirm this with the user, then override /**QuitFromUser** and insert your code there.

ClassBaseWindow

ClassBaseWindow is a subclass of ClassWindow, overrides a few ClassWindow methods, provides an icon, and provides a menu that the base window and its icon share. In addition, ClassBaseWindow overrides some of ClassWindow's painting methods in order to paint the icon.

Creation

/new — client | null parentcanvas /**new** instance

Creates an instance of ClassBaseWindow. By default a window is open when it is created and has a close box, a footer area, a label area, reshape corners but no pin.

Opening and closing base windows

/opened? - /**opened?** boolean

Returns whether the window is open or iconified.

Base window icons

The icon painting method, /**PaintIcon**, typically paints an image then a label. Both are arbitrary display items. In addition, the image size determines the size of the icon. If the icon image is null (i.e. no image has been set), the icon defaults to 64x64.

/IconFont - /**IconFont** font (Variable)

Font used by the icon label. Defaults to FooterFont.

/IconSize - /**IconSize** width height (Variable)

Returns the size of the icon's image, or 64x64 if the image is null.

/seticonimage display-item | hexstring /**seticonimage** -

Sets the icon image and repaints the icon. If a string is passed in, it is converted into an "image" display item whose image is composed of the hex data in the string. The hexstring is assumed to be square, i.e. an NxN array of bits. Because of the space considerations, it is a good idea for applications to share the icon image among all instances of their base windows.

/iconimage display-item /**iconimage** -

Returns the image display item.

/seticonlabel display-item /**seticonlabel** -

Sets the display item to be the icon's label and repaints the icon.

```
Example [(Hi) 1 0 0 rgbcolor] /seticonlabel win send
```

/iconlabel - /**iconlabel** display-item

Returns the icon's label.

Geometry

/minsize — - /**minsize** width height

Returns the window's minimum size. If the window is iconified the size of the icon is returned.

/preferredsize — - /**preferredsize** width height

Returns the "ideal size" for the window. If the window is iconified the size of the icon is returned.

Painting

/Paint — - /**Paint** -

Paints the window.

/**PaintChildren** and /**FixChildren** are no-ops when iconic.

Menus

By default base windows are Menuable and all the basewindows and their icons share the same menu. The iconic state of the window is tracked and the menu reflects the state by either having Close or Open in the menu depending on whether the window is open or closed. For a discussion of menus see Canvas menus on page 53 in ClassCanvas.

ClassPopupWindow

ClassPopupWindow currently overrides three window attributes to be true: /Label?, /Pin?, and /Reshape? It also handles a shared menu exactly like base windows, switching between "Cancel" and "Dismissed" based on the state of the /Changed? state of the window.

Creation

/new — client | null parentcanvas /**new** instance

Creates an instance of ClassPopupWindow.

The Wire Service

The purpose of the NeWS Wire Service is a server-client communications package of sufficient generality that diverse client applications and toolkits can use it.

The Wire Service is almost completely independent of the PostScript components of The NeWS Toolkit; it does not presume the existence of any particular class. However, in order to use the Wire Service you must know the NeWS client-side facility CPS and how to use. See the *NeWS 2.1 Programmer's Guide*, for an explanation of CPS.

Also included in this chapter are the client-side functions that you use to implement help facilities.

At the end of the chapter there is an example called `wire-demo`. This demo is also included as part of the TNT distribution CD and can be found in the demo directory.

Error handling

Most Wire Service interface functions return a value that can be coerced to an integer and tested for a 0 return value. Many of them return a boolean; `TRUE` for success, `FALSE` for failure.

`wire_Errno`

```
int wire_Errno;
```

`wire_ErrorString`

```
char * wire_ErrorString();
```

`wire_Perror`

```
void
wire_Perror(prefix)
char *prefix;
```

When an error has occurred, its type is available in the `wire_Errno` global variable, and a descriptive string is pointed to by `wire_ErrorString`. Like its UNIX equivalent, the error condition is not cleared immediately after an error. It remains set until the next error. The function `wire_Perror` prints the current error string to standard error, prefixed by the user-supplied string.

Components

The components of the Wire Service are:

- A connection manager that handles multiple connections to one or more servers (discussed on page 228).
- Handle allocation and registration procedures that allow items on one side of the wire to be referred to from the other side (discussed on page 234).
- A notifier on the C-side to allow asynchronous messages from the server(s) to be dispatched to client functions (discussed on page 238).
- A synchronization package to allow server-based code to make RPC-style calls across the wire (discussed on page 241).

Connection management

The connection management functions support multiple connections.

`wire_Open`

```
wire_Wire
wire_Open(server)
char *server;
```

Opens a connection to a server[1] which makes the wire the current connection and enables it. All libcps calls use the current connection. `wire_Open` takes an argument to specify the particular server to connect to. If you use `NULL` as an argument, `wire_Open` attempts to use the `NEWSSERVER` environment variable. If `NEWSSERVER` is unset, the environmental variable, `DISPLAY` is used. If `DISPLAY` is not set, then the current host is used with default port 2000. If the argument is not `NULL` then it should be a hostname, a `NEWSSERVER`-style string, or a `DISPLAY`-style string. For more information on the `NEWSSERVER` and `DISPLAY` variables and the correct format of their string values, see the *X11/NeWS Version 2 Server Guide*.

`wire_Close`

```
boolean
wire_Close(w)
wire_Wire w;
```

Closes an opened connection. If the argument is `wire_ALLWIRES`, all the connections are closed, however, `FALSE` is returned if there is an error with *any* of the connections.

`wire_SetCurrent`

```
boolean
wire_SetCurrent(w)
wire_Wire w;
```

Sets the current wire, i.e., it directs CPS to and from a the connection from now on. `wire_SetCurrent` has the effect of moving the appropriate file pointers into the CPS global variables, “PostScript” and “PostScriptInput”. All libcps calls will thereafter use this connection. The act of opening a wire sets it to be the current wire.

1. Implementation Note: The current implementation uses 2 file descriptors per connection. Thus, the number of available wires is determined by the number of available file descriptors in the system. This is highly implementation-specific and may be changed in the future.

`wire_Current`

```
wire_Wire
wire_Current()
```

Returns the current connection. Your program may need to determine what the current wire is because the notifier (see The Notifier on page 238) may *itself* change the current connection depending on where the next message has come from. Clients that do not want to reply down the same connection as their upcoming message will have to call `wire_SetCurrent` again before they write.

`wire_Valid`

```
boolean
wire_Valid(w)
wire_Wire w;
```

Indicates whether wire `w` is valid. A valid wire represents a connection to a NeWS server that is open and active.

`wire_SetData`

```
boolean
wire_SetData(w, data)
wire_Wire w;
caddr_t data;
```

Associates a client data pointer with a connection. Applications may associate client data with each connection via the `wire_SetData`. The most common use of `wire_SetData` will be to reestablish some per-connection application context when processing a message from a particular connection.

`wire_Data`

```
caddr_t
wire_Data(w)
wire_Wire w;
```

Returns the client data pointer for a connection.

`wire_Disable`

```
boolean
wire_Disable(w)
wire_Wire w;
```

Disables a connection. You can use `wire_AllWires` as the wire argument to disable all the wires in a single call. An error is reported if there is a problem with any one of the connections. `wire_Disable` allows input from a particular connection to be ignored temporarily, and later the wire is reactivated via `wire_Enable` (see below)**.** While a connection is disabled, the notifier (see *The Notifier* on page 238) does not read any messages from it, and no functions are called on its behalf. The purpose of this function is to allow a client to negotiate with one server, and guarantee that the client won't be interrupted by messages from another. Disabling a wire only affects its input side; writes to a disabled wire will succeed.

`wire_Enable`

```
boolean
wire_Enable(w)
wire_Wire w;
```

Enables a connection. When first opened, a connection is enabled. You can use `wire_AllWires` as the wire argument to enable all the wires in a single call. An error is reported if there is a problem with any one of the connections.

`wire_Enabled`

```
boolean
wire_Enabled(w)
wire_Wire w;
```

Returns whether a connection is enabled.

Handling abnormal events on a connection

The wire service has three default functions that handle certain abnormal events (e.g., an unexpected termination of a connection). You may "override" any or all of the default functions by supplying functions that the notifier will call after these abnormal events. You register your functions with the notifier using the function `wire_Problems`.The abnormal events and their associated user-defined functions are:

- If the connection is terminated, other than by a call to `wire_Close`, `death` is called. Your definition of this function should *not* attempt to close the offending wire.
- If the notifier finds data at the head of an input queue that is not recognizable as a dispatching tag, the current connection is preset to the offending one and `disease` is called. `disease` must consume the leading non-tag values from the stream.
- If the notifier finds a dispatching tag which has not been registered using `wire_RegisterTag`, `unknowtag` is called.

User-supplied functions

`wire_Problems`

```
wire_Problems(w, death, disease, unknowntag)
wire_Wire w;
void (*death)();
void (*disease)();
void (*unknowntag)();
```

Registers functions that are called on connection death or reading errors. If `wire_Problems` is called with `wire_ALLWIRES` as the first parameter, then the same set of callbacks will be used for all of the existing connections. A `NULL` argument to any of these three arguments to `wire_Problems` leaves that function unchanged.

```
void
death(w)
wire_Wire w;
```

Called on abnormal connection termination.

```
void
disease(w)
wire_Wire w;
```

Called after connection protocol error.

```
void
unknowntag(w)
wire_Wire w;
```

Called when an unregistered tag is found.

Default functions

`wire_DeathDefault`

```
void
wire_DeathDefault();
```

Prints a message to `stderr`.

`wire_DiseaseDefault`

```
void
wire_DiseaseDefault();
```

Cleans up the queue and prints a message to `stderr`.

`wire_UnknownTagDefault`

```
void
wire_UnknownTagDefault();
```

Consumes the tag and any following arguments, and prints a message to `stderr`.

`wire_SkipEvent`

```
boolean
wire_SkipEvent()
```

Consumes a token and skips to the next tag. `wire_SkipEvent` consumes the initial token on the current wire and any remaining input up to, but not including, the next tag. If there is no next tag on the current wire, `wire_SkipEvent` will not block waiting for one. This function is useful when writing 'disease' and 'unknowntag' functions.

Indexing data structures by wires

Certain clients of the Wire Service may want to build data structures that are indexed by a wire. For this reason a pair of procedures (currently macros) are provided that map a wire into a unique small integer and back again. This is meant to be used in those cases where a client does not want to use the client data field associated with the connection.

`wire_WireToInt`

```
int
wire_WireToInt(w)
wire_Wire w;
```

Maps a wire to a small integer.

`wire_IntToWire`

```
wire_Wire
wire_IntToWire(i)
int i;
```

Maps a small integer into a wire.

Accessing the psio files

`wire_PSinput` and `wire_PSoutput` are accessor functions to the psio file pointers. Use these functions if your program needs to access the psio files. (For an explanation of psio see the *X11/NeWS Version 2 Server Guide* or the psio(3) man page.)

`wire_PSinput`

```
PSFILE *
wire_PSinput(w)
wire_Wire w;
```

Returns a pointer to the psio input file pointer.

wire_PSoutput

```
PSFILE *
wire_PSoutput(w)
wire_Wire w;
```

Returns a pointer to the psio output file pointer.

Handle allocation and registration

Both C and PostScript components of the Toolkit need to reference remote objects. The C programmer may need to modify or query some PostScript object created earlier. Similarly, any PostScript object that wishes to notify the client of a user event needs some way to specify the appropriate C function to invoke. Because references to PostScript objects cannot be passed across the wire and C pointers cannot easily be stored in the server, the Toolkit provides two "handle allocators" that generate and remember unique identifiers.

Tags

The Wire Service uses "tags," as provided by CPS, to drive its notifier. Before you can register a callback with the notifier, you must obtain a tag to associate with it.

wire_AllocateTags

```
int
wire_AllocateTags(count)
int count;
```

Reserves a range of client tags. `wire_AllocateTags` takes a number N and returns another M, such that none of the integers M, M+1, ... M+N-1 are already allocated, i.e. it returns the number of tags allocated. These integers are handles whose primary use is to dispatch messages from the server to client functions.

wire_AllocateNamedTags

```
boolean
wire_AllocateNamedTags(names)
int *names[];
```

Assigns client tags to an array of addresses. `wire_AllocateNamedTags` is a thin wrapper around `wire_AllocateTags`. It takes a `NULL` terminated array of pointers to integers, and assigns a tag through each of these pointers. A typical use might look like:

```
int menu_tag, resize_tag;
int *tag_pointers[] = {&menu_tag,&resize_tag, NULL};

wire_AllocateNamedTags(tag_pointers);
wire_RegisterTag(menu_tag, my_menu_callback, data);
wire_RegisterTag(resize_tag, my_resize_callback,data);
```

`wire_ReserveTags`

```
boolean
wire_ReserveTags(largest)
int largest;
```

Makes the tag allocator ignore a range of integers. `wire_ReserveTags` is provided to allow dynamically-allocated tags to coexist with a previous version of CPS that used constant tags. If you know that some piece of code uses tag values 1..50, then before calling `wire_AllocateTags` you should call `wire_ReserveTags(50)`. This facility can also be used to leave space for your own private tag allocator if the one provided by the Wire Service doesn't meet your needs. `wire_ReserveTags` must be called before any connections are opened.

`wire_RegisterTag`

```
boolean
wire_RegisterTag(tag, proc, data)
int tag;
void (*proc)();
caddr_t data;
```

Registers a client handler. `wire_RegisterTag` allows you to associate a procedure pointer and a user data pointer with a tag. If this tag is ever found on the wire by the notifier, your procedure will be called.

```
void
(*proc)(tag, data)
int tag;
caddr_t data;
```

`wire_TagProc`

```
void (*)()
wire_TagProc(tag)
int tag;
```

Retrieves a client function pointer.

`wire_TagData`

```
caddr_t
wire_TagData(tag)
int tag;
```

Retrieves a client data pointer.

The environment in which your callback function (*proc above) is called is as follows:

- The current wire will be the wire on which the given tag arrived. This implies that normal CPS techniques for reading values from a connection can be used.
- By the time your function is called the tag will have already been removed from the wire (and handed to you as an argument).

It is your responsibility to know how many arguments follow this particular tag, and to read them all from the wire before returning from this function. You may do this either via CPS input functions, or the "ease of use" procedures described in *Ease-of-use functions*. If you do not remove all the arguments from your wire, the disease handling function will be called.

Since the Wire Service is reentrant it is admissable to call `Wire_Notify` or `Wire_EnterNotifier` from within your callback function. This is sometimes useful when synchronizing with the user.

Tokens

The Wire Service uses "usertokens", as provided by CPS, as handles to PostScript objects. These tokens are allocated on a per connection basis. The application is responsible for the registration of the usertoken in the server. These calls are similar to the calls for tag allocation, except that they are done on a per-wire basis and there is an additional call to free up a range of user tokens (`Wire_DeallocateTokens`).

Note – It is still up to the user to remove the reference to the server object in the user token array (by assigning null to that user token). Unlike `wire_ReserveTags`, `wire_ReserveTokens` is called after the connection has been opened.

`wire_AllocateTokens`

```
int
wire_AllocateTokens(w, count)
wire_Wire  w;
int count;
```

Reserves a range of client tokens on a specific wire.

`wire_AllocateNamedTokens`

```
boolean
wire_AllocateNamedTokens(w, names)
wire_Wire w;
int *names[];
```

Assigns tokens for a specific wire to an array of addresses.

`wire_ReserveTokens`

```
boolean
wire_ReserveTokens(w, largest)
wire_Wire w;
int largest;
```

Makes the token allocator for a specific wire ignore a range of integers.

`wire_RegisterToken`

```
boolean
wire_RegisterToken(w, token, obj)
wire_Wire w;
int token;
wire_RefAny obj;
```

Associates a server-side object (`token`) with a client-side one (`obj`). `wire_RegisterToken` is a way to associate a client-side object (or any piece of data) with a usertoken. For example, a usertoken might represent a server canvas, and the client-side object that parallels that server object would be registered with `wire_RegisterToken`. Then when that canvas makes a callback, it will include the usertoken as an argument, and the client side will look that up with `wire_TokenData` (see below). `wire_RegisterToken` does not register the usertoken in the server because this cannot be done from the C process—you need a reference to the PostScript object to make it a usertoken.

`wire_TokenData`

```
wire_RefAny
wire_TokenData(w, token)
wire_Wire w;
int token;
```

Given a server side object (`token`); retrieve the client-side one.

`wire_DeallocateTokens`

```
boolean
wire_DeallocateTokens(w, first, count)
wire_Wire w;
int first, count;
```

Frees up a range of allocated tokens.

An example of the way you should use the usertoken facility for registering your server-side objects is:

In your C file:

```
struct canvas *c;

c = canvas_create();
c->token = wire_AllocateTokens(wire_Current(), 1);
wire_RegisterToken(wire_Current(), c->token, (caddr_t) c);
ps_CreateMyCanvas(c->token);
```

And in your CPS file:

```
cdef ps_CreateMyCanvas(int token)
     ... /new MyCanvas send           % canvas
% /SaveToken defined in canvas subclass
     token /SaveToken 2 index send    % canvas ; save token
        token setfileinputtoken
```

Now if the canvas makes a callback to the client side, it sends the token number over the wire so the client side knows which canvas is making the callback. For example, one of the canvas's methods might send a mouse event across the wire.

In your window class:

```
/sendmouseevent { % event => -
    MOUSE_EVENT tagprint  % event
    /GetToken self send tagprint % event ; send token; /GetToken defined in canvas subcl.
    /EventToXY self send exch % y x
    tagprint tagprint   % - ; send coordinates
} def
```

And in your client callback:

```
static void
mouse_callback()
{
    struct canvas *c;
    int  token_id;

    ... /* get token_id from callback */
    c = (struct canvas *) wire_TokenData(wire_Current(), token_id);
}
```

The Notifier

The purpose of the notifier is to read tags from one or more server connections, and depending on their value, call the client functions that were previously registered using `wire_RegisterTag`. Two styles of notification are provided (The two styles can also be mixed in the same application.):

1. The notifier itself can handle the main loop.

2. The client program can repeatedly request the dispatching of a single incoming message.

`wire_Notify`

```
boolean
wire_Notify(timeout)
struct timeval *timeout;
```

Reads and processes one message. `wire_Notify` causes a single tag to be read from one of the active connections. (Round-robin scheduling is used when more than one connection with data ready for reading exists.) The tag read is used to look up the registered procedures. The procedure is then called with the handle and registered data as arguments. (See the function `my_slider_handler` in the example at the end of the chapter.) `wire_Notify` has the side-effect of setting the current connection, which allows registered functions to read further arguments from the wire using CPS and psio functions. If no data is available on any of the active connections, `wire_Notify` blocks until a message arrives or the period specified in the timeout parameter expires.

If a timeout occurs `wire_Notify` returns `FALSE`. When `wire_Notify` is blocked waiting for input, and a signal interrupts it before the input arrives, `wire_Notify` returns `FALSE` and `wire_Errno` is set to `wire_EINTR`. This corresponds to the UNIX system call error `EINTR` (interrupted system call). Programs are often interrupted, but that doesn't mean the program should exit. Therefore, when `wire_Notify` returns false, `wire_Errno` should be checked to see if it is `wire_EINTR`. If `wire_Errno` is set to `wire_EINTR` the program should continue in the notify loop. `wire_EnterNotifer()` ignores interrupted system calls, but will return if there is an error for another reason.

`wire_WouldNotify`

```
boolean
wire_WouldNotify(w)
wire_Wire w;
```

Indicates whether there are any messages to read on this wire. `wire_WouldNotify` does not block. If you use `wire_ALLWIRES` as the wire argument, `wire_WouldNotify` indicates whether there are messages on any of the active connections.

`wire_EnterNotifier`

```
void
wire_EnterNotifier()
```

Descends into the main loop. `wire_EnterNotifier` will be the main-loop for many client applications. It will be reentrant, and can be intermixed with calls to `wire_Notify`. In fact, it will do little more than repeatedly call `wire_Notify` itself. A call to `wire_EnterNotifier` does not return until the corresponding `wire_ExitNotifier` is executed.

`wire_ExitNotifier`

```
void
wire_ExitNotifier()
```

Emerges from main loop. `wire_ExitNotifier` is called when the application programmer wants to exit from a (possibly nested) notifier loop. The corresponding `wire_EnterNotifier` will return as soon as the registered procedure that called `wire_ExitNotifier` itself returns. Pending messages are not processed in any way.

`wire_AddFileHandler`

```
boolean
wire_AddFileHandler(file, callback, data)
FILE *file;
void (*callback)();
caddr_t data;
```

Adds a file to the notifier's list of files to check. When data is detected on the file, `(*callback)()` is called and passed the data pointer. No restrictions are imposed on the file and it is up to the client to handle all operations within the callback. `wire_AddFileHandler` takes a file pointer but if a file descriptor is desired in the application program, a call to `fdopen` can be made with no adverse side-effects.

`wire_RemoveFileHandler`

```
boolean
wire_RemoveFileHandler(file)
FILE *file;
```

Removes a file from the notifier's list of files to check. The file is not a `wire_Wire` and cannot be enabled/disabled, etc. There are no restrictions imposed on the file and it is up to the client to handle all operations within the callback. `wire_RemoveFileHandler` takes a file pointer, but If a file descriptor is desired in the application program, a call to `fdopen` can be made with no adverse side-effects.

Ease-of-use functions

The following functions are provided to enable data to be easily read from the current connection. It is assumed that the user of these functions knows the type of the data on the wire. Thus there is no type checking or error reporting. If the data is of the wrong type, garbage may be returned and the wire may be left in an undetermined state. The caveat to this is numeric arguments are converted by CPS (floats to ints, ints to floats, etc.).

`wire_ReadInt`

```
int
 wire_ReadInt()
```

`wire_ReadFloat`

```
float
 wire_ReadFloat()
```

`wire_ReadString`

```
char *
wire_ReadString(str)
char *str;
```

`wire_GobbleAny`

```
void
wire_GobbleAny()
```

Synchronization

CPS provides a mechanism for a client process to block pending notification from a server process. The wire service provides a complementary mechanism that allows a server process to block pending notification from a client process. This provides symmetric facilities for synchronous communications.

Server interface:

wire_Sync

```
proc wire_Sync -
```

The proc is executed, and `wire_Sync` guarantees it will not return until the C client has acknowledged dealing with any data sent to it by the proc. That is, `wire_Sync` execs the proc, and sends a marker to the C client, then waits for the client to acknowledge having seen the marker. For example, PostScript can ask C to send it some value, or to do some painting, etc., and be sure that C has responded to the request before trying to do any more PostScript code. Naturally, this makes some assumptions about the client; hence the requirement that the client be built on the Wire Service.

Things can get rather tricky if `wire_Sync` is called from the "listener" process, i.e., the process that is reading from the client connection. `wire_Sync` guarantees that anything the client ships in response to the proc will be executed (a) before `wire_Sync` returns and (b) before any other tokens that the client may have previously shipped down the wire.

Client interface:

`wire_InSync`

```
boolean
wire_InSync(w)
wire_Wire w;
```

Indicates whether the wire is responding to a synchronized request.

`wire_InSync` simply checks to see if wire `w` is responding to a synchronized request from the server, which means that PostScript code sent now may be executed before PostScript code sent earlier. This is a subtle but important point.

Wire Service synchronization caveat.

The Wire Service has some very obscure problems with client-server synchronization. These problems are attributable to the flexibility NeWS gives you for having code on both sides of the wire. In most cases the Wire Service will work fine but the following problems are indications that you are encountering synchronization problems:

- You start having symptoms like deadlock, e.g., your program is stuck in a cps call that waits for return values.

or

- You find that when you use a user token, the value it turns into is a null.

If you are experiencing these symptoms then you should revisit this section and read it carefully.

The problem

When the server-side code does a synchronized callback to the client, it can result in data being read from the wire without that data being immediately executed. In other words, the server-side callback wants to get the response from the client before continuing. After getting the response the callback wants to execute anything else that came down the wire while it was waiting.

There are two cases where the client needs to be sure that the server is executing what it receives, i.e., data isn't being cached for "later execution". These two cases are:

1. The client is about to do a "blocking cps call", i.e., one that returns a value.

The client is about to send a snippet that includes a usertoken, and wants to be sure that the definition of that usertoken will have been executed before To ensure the server is executing what it receives, the application writer should use the two functions: `wire_ExpectSync` and `wire_DrainSync`.

`wire_ExpectSync`

```
boolean
wire_ExpectSync(wire)
wire_Wire wire;
```

`wire_ExpectSync()` is called after calling a cps function that might result in the server executing some wire_Sync callbacks.

`wire_DrainSync`

```
boolean
wire_DrainSync(wire, notify_proc)
wire_Wire wire;
void (*notify_proc)();
```

`wire_DrainSync` is called when the client needs to make sure the server is actually executing what it receives. Thus you call it before making a blocking cps call, and you call it somewhere between sending the definition of a usertoken and the first time you try to use that token.

In both cases, the purpose of `wire_DrainSync` is to make sure there are no pending wire_Sync calls yet to come from the Listener process. `wire_DrainSync` repeatedly executes *notify_proc* in a loop until the number of expected syncs goes to zero. It is the responsibility of the *notify_proc* to execute at least one call to `wire_Notify()`, in addition to anything else it wants to do (like update some other information).

Help facilities

The TNT help facilities assume that you will use the Wire Service, in particular the notifiers provided by the Wire Service. In addition there are server-side methods used in conjunction with these client-side functions. You can find the applicable methods in ClassCanvas, Help facilities on page 63.

`Help_Initialize`

```
void
Help_Initialize(w);
wire_Wire w:
```

Initializes the help system. The wire passed to `Help_Initialize` should have been opened with `wire_Open()`. The connection used to initialize the help library should be the same connection used to load the definitions of all windows containing Helpable objects into the server. The userdict associated with this connection will contain information necessary for TNT objects to pass help requests back to the client for message lookup and display.

The Help library currently looks in the directory where the application was launched, and in `$OPENWINHOME/lib/help` for help message files. If you want the message file to reside elsewhere, you are responsible for setting up `HELPPATH` correctly *before* calling `Help_HelpInitialize`.

`Help_UpdateView`

```
void
Help_UpdateView ( );
```

Updates the canvas that displays the help message. The recommended style for a notification loop involving help uses `Help_UpdateView`. For example:

```
while ( !quit && wire_Notify((struct timeval *) NULL) ) {
          if (wire_WouldNotify(thiswiire))
                 continue
       . . .
          Help_UpdateView();
}
```

`Help_HelpRequestHandler`

```
void
Help_RequestHelpHandler(tag, data);
int tag;
caddr_t data;
```

Handles help requests transmitted from the server. Its default behavior sets up the pane, adds it to the help window, looks for the help text and displays it.

`Help_RequestHelpHandler` is the callback function that is registered with the Wire Service. Replace it *only* if you want some kind of custom help display.

The help message file

The help library expects messages in a file to follow this format:

- Keywords are lines that start with a colon (:) character.
- Any line beginning with a pound-sign (#) is a comment line, and ignored.
- Messages are one or more lines in between keywords and/or comment lines. Message files are ordinary text files following the format in the example below. Messages containing no newline characters will be word-wrapped inside the help window.

Thus a help file would contain entries similar to:

```
# Help for myslider
:myslider
Moving the slider adjusts the value of the gauge.
#
#Help for Save button
:savebutton
Pressing this button will save your work before quitting.
#
# Help for Discard button
:discardbutton
Pressing this button will quit without saving changes.
#
# Help for Cancel button
:cancelbutton
Pressing this button will cancel the Quit operation.
#
```

Constants

The following constants are defined in the wire service:

Boolean values:	`TRUE  FALSE`
Special Wires:	`wire_INVALID_WIRE`
	`wire_ALLWIRES`
Error values:	`wire_EUNKNOWNHOST`
	`wire_ENOSUCHSERVER`
	`wire_EBADWIRE`
	`wire_ECONNECTIONDIED`

```
wire_ENOWIRES
wire_ETIMEOUT
wire_EINTR
wire_ERANGECHECK
wire_EBADRESERVE
wire_EFILEINUSE
wire_ENOFILEHANDLER
wire_ECONNECTIONREFUSED
```

Sample Program

```
/* wire.c
 * Sample program to illustrate the proposed NeWS Wire Service.
 * A slider is placed on the framebuffer, and values are notified
 * back to the client. If the slider is dragged to 0, the program
 * terminates.
 * The program consists of two files: this one, called wire.c and
 * the file called wire_demo_cps.cps, which starts at the top of
 * the page following the last line of wire.c.
 */

#include <NeWS/wire.h>
#include "wire_demo_cps.h"

#if !defined (lint)
static char sccsid[] = "@(#)wire_demo.c 1.5 91/02/11 Copyright 1985 Sun Micro";
#endif



/*
 * Slider callback function.
 */
void
my_slider_handler(tag, data)
int     tag;
caddr_t  data;
{
      /*
      * Let's assume that the client wants to know the new value of the
      * slider when it changes. He must read it himself in this procedure.
      * The data argument in this case holds the server handle provided to
```

```
      * wire_RegisterTag().
      */
      float   value;

      /* Take the value off the wire */
      value = wire_ReadFloat();

      printf("The value of the slider is now %f\n", value);

      if (value == 10.0) {
         /*
          * Modify the slider if its value is now 10.0!
          */
         ps_grow_slider((int) data, 10);
      }
} /* my_slider_handler() */


/*
 * Quit callback function.
 * Triggered from the Quit window menu item.
 */
void
my_quit_handler(tag, data)
int     tag;
caddr_t  data;
{
      /*
      * Assume that there is no application specific cleanup required.
      */
      wire_ExitNotifier();
```

```
}    /* my_slider_handler() */


/*
 * Wire demo main program.
 */
main()
{
     wire_Wire  w;                  /* My connection */
     int        slider_tag;         /* Client-side handle */
     int        quit_tag;
     int        slider_token;       /* Server-side handle */

     /* Open a connection to your default NeWS server */
     w = wire_Open(NULL);
    /*
     * Reserve a set of tokens (1 in this case) by which
     * the client can refer to objects on the server. This will
     * only be necessary if 1) you need to refer to an object after
     * it has been created, and 2) you don't want to provide your
     * own set of names.
     */
      slider_token = wire_AllocateTokens(w, 1);

     /*
      * Reserve a set of tags (2 in this case) by which
      * server-based objects can call functions on the client-side.
      * Then register my functions to be called when this tag is received.
      * The registration function also takes an arbitrary client data
      * value, which in this case we choose to use to store the server
      * handle.
```

```
     */
     slider_tag = wire_AllocateTags(1);
     wire_RegisterTag(slider_tag, my_slider_handler, (caddr_t) slider_token);

     quit_tag = wire_AllocateTags(1);
     wire_RegisterTag(quit_tag, my_quit_handler, (caddr_t) slider_token);


    /*
     * Send PS to the server to create a slider on the framebuffer.
     * THIS IS NOT A FUNCTION PROVIDED BY THE WIRE SERVICE.
     * (Implementation below).
     */
 ps_create_slider(slider_tag, quit_tag, slider_token, 100, 100, 200);

    /*
     * Enter the read/dispatch loop. This function will not return
     * until after wire_ExitNotifier() has been called from inside
     * some callback function -- in this case my_quit_handler()
     */
 wire_EnterNotifier();

 /* Close and exit gracefully */
 wire_Close(w);
 exit(0);
}
```

```
% This is the file wire_demo_cps.cps
% */
%
% Create the slider and position it at (x,y). Give it the standard
% height and make it "width" wide. The executable array is the
% slider's notify proc. In this case it simply tagprints the
% client tag, and typedprints its value.
%
% Note that the server_handle is passed in as an int because the
% usertoken hasn't been initialized yet.
%
cdef ps_create_slider(int client_handle, int quit_tag, int server_handle,
     int x, int y, int width)

      % Make a window with a calculated pane.
      /pan /Calculated framebuffer /new ClassPanel send def
      /win pan framebuffer /new ClassBaseWindow send def
      (Wire Service Demo) /setlabel win send
      (Value 10) (extends slider) /setfooter win send

      % Make a slider.
      /slider framebuffer /new ClassHSlider send def

      % Turn on the end boxes. Don't bother with end labels.
      true /setendboxes slider send

      % Let's start out with a range of 0..20
      0 20 /setrange slider send

      % And let's set tic marks every two units.
      2 /settickmarks slider send
```

```
     % Make values integers.
     { round cvi } /setnormalizer slider send

% Set up slider target, notifier, and previewer.
win /settarget slider send

{ % Notifier:                              % value slider
    client_handle tagprint
    1 index typedprint

    % Tell target (our window) to set its footer.
    exch (%) sprintf                      % s (v)
    (Slider Notify) exch                  % s (SN) (v)
    /setfooter /sendtarget 5 -1 roll      % (SN) (v) /sf /st s
    send                                  % -
} /setnotifier slider send

{ % Previewer:                             % value slider
    % Tell target (our window) to set its footer.
    exch (%) sprintf                      % s (v)
    (Slider Preview) exch                 % s (SN) (v)
    /setfooter /sendtarget 5 -1 roll      % (SN) (v) /sf /st s
    send                                  %
} /setpreviewer slider send

% Put the slider in the center of the panel.
/slider slider [/Center {/Center PARENT POSITION}]
/addclient pan send
```

```
 % Size it
 x y width /minsize slider send exch pop
 /reshape slider send

 /QuitFromUser { % CallingControl => -
     pop
     quit_tag tagprint
     /QuitFromUser super send
 } /installmethod win send

 % Reshape and activate the window.
 20 40 20 40 /setgaps win send
 /place win send
 /new ClassEventMgr send /activate win send
 /map win send

 % Now associate the server_handle token with the slider
 slider server_handle setfileinputtoken
%
% Grow the slider. Don't ask me why.
%
% Note that server_handle is declared as a token so that the usertoken
% lookup will be performed.
%
cdef ps_grow_slider(token server_handle, int delta)
       /bbox server_handle send
       exch delta add exch
       /reshape server_handle send

       /bbox win send /preferredsize win send
       xymax /reshape win send
```

Jot

Introduction to Jot

Jot is the text package included with the NeWS toolkit. Jot defines both the data structures and procedures required to manage presentation and interaction with text. You can use either C or C++ when using the Jot interface.

How Jot functionality is organized

Jot's functionality is organized according to the model-view-controller paradigm. This paradigm decomposes an application into three components. The model is a data object, representing application information. The view presents its associated model in a graphical fashion. The controller provides an interface between input devices and the model that allows the user to interact with the model through its view.

For example, a model consisting of numeric data might be viewed as a spreadsheet, a pie chart, or a line graph. When the user enters a number in a spreadsheet cell, the controller notifies the model, which stores the new information. The model then requests its views to update their presentations.

The arrangement of Jot functionality does not quite map into the model-view-controller paradigm. The text model is called JotText and all the procedures that deal with the model are prefaced with `JotText_`. The view and controller procedures are combined and are prefaced by `JotView_`.

In addition, Jot uses spans to reference specific pieces of the text buffer (`JotSpan_`) and has functions for implementing a text search facility (`JotSearch_`) and for handling selections (`JotSelection_`).

Position definition

In Jot, *position* indicates a point between characters so that the location of operations like inserting characters is unambiguous. A characters said to be located at a position is actually the character to the right of that position. The first character in a JotText is at position zero (0).

Global error descriptions

Whenever a Jot function indicates an error condition, the global error variable, `Jot_Errno` is also set. You can examine this variable for a more detailed explanation of the failure.

Table 27-1 Jot errors

Error name	Description
`Jot_ECONSTRAIN`	The characters in the JotText buffer could not be constrained to the JotView. This error occurs only when a JotView is constrained.
`Jot_EMEMORY`	An attempt to allocate memory failed.
`Jot_ERANGECHECK`	Position and/or length parameters exceeded the range of the associated JotText buffer. For JotSelections, the rank or level were inappropriate.
`Jot_ESELECTION`	An operation was attempted on a selection that didn't exist.
`Jot_ESYNTAX`	An invalid regular expression was detected.
`Jot_ETEXT`	An operation was attempted on a JotSpan or JotView not associated with a JotText instance.

Jot initialization

`Jot_Initialize` must be performed on each Wire Service *connection* prior to executing other Jot procedures.

`Jot_Initialize`

```
void
Jot_Initialize(connection)
wire_Wire connection;
```

Initializes a Jot connection. Information is placed on the server associated with the *connection*. In addition, Jot internal data structures are initialized to default values.

JotText procedures—the text model

JotText is the text model managed in Jot. This model includes a character buffer and procedures to access and change this buffer. The developer can insert or delete characters, read characters from a file stream, or write characters to a file stream. In addition, the developer can search for specific patterns and perform inquiries on the buffer state.

`JotText_New`

```
JotText *
JotText_New(length)
int length;
```

Allocates and returns a pointer to a new JotText instance. A `NULL` (0) pointer is returned if the operation fails. The initial size of the JotText buffer, *length*, can be specified. If *length* is negative, the size defaults to zero.

The *length* parameter is declared for performance reasons. When the initial size is zero, sequential insertions might result in the buffer space being reallocated several times. For example, when a large file is to be edited in an application, you can define a reasonable *length* for the buffer during text creation. The performance penalty associated with numerous allocations is then avoided.

Errors: `Jot_EMEMORY`

`JotText_Free`

```
void
JotText_Free(text)
JotText *text;
```

Deallocates *text* and its internal resources. Any JotViews or JotSpans associated with *text* are returned to a pristine state for reuse.

`JotText_Clear`

```
void
JotText_Clear(text)
JotText *text;
```

Deletes all the characters in the *text* buffer. Any JotSpans associated with *text* reset their length and position to zero.

`JotText_CharacterAt`

```
int
JotText_CharacterAt(text, position)
JotText *text;
int position;
```

Returns the character following *position* in the *text* buffer. A -1 is returned if the operation fails.

Errors: `Jot_ERANGECHECK`

`JotText_FastCharacterAt`

```
int
JotText_FastCharacterAt(text, position)
JotText *text;
int position;
```

Returns the character following *position* in the *text* buffer. `JotText_FastCharacterAt` performs no range checking. Although the macro is faster than the related `JotText_CharacterAt` function, it is also dangerous unless used with care. In addition, constructs such as `JotText_FastCharacterAt(text, position++)` should be avoided, because position will be incremented twice.

`JotText_InsertCharacters`

```
int
JotText_InsertCharacters(text, position, buffer, length)
JotText *text;
int position;
char *buffer;
int length;
```

Inserts *length* characters from *buffer* into the *text* buffer, at *position*. When the operation is successful, the number of characters inserted is returned; otherwise, a -1 is returned.

Errors: `Jot_ECONSTRAIN Jot_EMEMORY Jot_ERANGECHECK`

`JotText_ScanCharacter`

```
int
JotText_ScanCharacter(text, position, c, n)
JotText *text;
int pos, c, n;
```

Returns the position of the nth occurrence of *c* in *text*. It starts at *position* and searches for the Nth occurrence of *c*. If *n* is a negative number, it searches backwards from `position` starting from the character just before `position`. `JotText_ScanCharacter` returns -1 if there are not *n* occurrences.

`JotText_InsertString`

```
int
JotText_InsertString(text, position, string)
JotText *text;
int position;
char *string;
```

Inserts a null terminated *string* into the *text* buffer, at *position*. When the operation is successful, the number of characters inserted is returned; otherwise, a -1 is returned.

Errors: `Jot_ECONSTRAIN Jot_EMEMORY Jot_ERANGECHECK`

`JotText_ReplaceCharacters`

```
int
JotText_ReplaceCharacters(text, position, length, buffer, buffer_length)
JotText *text;
int position;
int length;
char *buffer;
int buffer_length;
```

Replaces *length* characters in the *text* buffer, at *position*, with *buffer_length* characters from *buffer*. When the operation is successful, the number of characters inserted is returned; otherwise, a -1 is returned.

Errors: `Jot_ECONSTRAIN Jot_EMEMORY Jot_ERANGECHECK`

`JotText_DeleteCharacters`

```
int
JotText_DeleteCharacters(text,position,length)
JotText *text;
int position;
int length;
```

Deletes *length* characters from the *text* buffer, after *position*. When *length* is negative, *length* characters before *position* are deleted. When the operation is successful, the number of characters deleted is returned; otherwise no deletion is performed and -1 is returned.

Errors: `Jot_ERANGECHECK`

`JotText_Read`

```
int
JotText_Read(text, position, file)
JotText *text;
int position;
int file;
```

Reads from the *file* descriptor, inserting characters into the *text* buffer at *position* until EOF is detected. When the operation is successful, the number of characters read is returned; otherwise, a -1 is returned. You are responsible for opening, positioning, and closing *file*.

Note – The *file* parameter should not be set to non-blocking mode. Applications that require this feature can use `read(2)` or `fread(3)` to read characters into a private buffer and then use `JotText_InsertCharacters` or `JotText_InsertString` to transfer the characters into a JotText buffer.

Errors: `Jot_ECONSTRAIN` `Jot_EMEMORY` `Jot_ERANGECHECK` (plus all the errors that you can get when using `read(2)`)

`JotText_Write`

```
int
JotText_Write(text,position,length,file)
JotText *text;
int position;
int length;
int file;
```

Writes *length* characters, beginning at *position*, from *text* buffer to the *file* descriptor. When the operation is successful, the number of characters written is returned; otherwise, a -1 is returned. You are responsible for opening, positioning, and closing file.

Note – If the *file* parameter is set to non-blocking mode and refers to a pipe or socket, `JotText_Write` may write fewer bytes than requested. The return value should be noted and the operation retried.

Errors: `Jot_ERANGECHECK` (plus all the errors that you can get when using `write(2)`)

`JotText_Modified`

```
int
JotText_Modified(text)
JotText *text;
```

Returns the total number of modifications performed on *text* since its creation. The modification total is incremented each time the buffer is changed. To determine whether *text* has been modified, you compare the return values from two successive calls to `JotText_Modified`. `JotText_Modified` is the foundation for developing a checkpoint service.

`JotText_Characters`

```
int
JotText_Characters(text)
JotText *text;
```

Returns the number of characters in the *text* buffer.

`JotText_Newlines`

```
int
JotText_Newlines(text)
JotText *text;
```

Returns the number of newline characters in the *text* buffer. Jot does not cache this information in a JotText instance; the number of newline characters is recalculated each time `JotText_Newlines` is called. Therefore, a slight performance penalty might result from locating and counting newline characters in a large buffer.

`JotText_FirstView`

```
JotView *
JotText_FirstView(text)
JotText *text;
```

Returns a pointer to the first JotView instance in the *text*'s view list. When the list is empty, a `NULL` (0) pointer is returned.

`JotText_NextView`

```
JotView *
JotText_NextView(view)
JotView *view;
```

Returns a pointer to the JotView instance following *view* in the list managed by the owner of *view*. When *view* is the last JotView in the list, a `NULL` (0) pointer is returned. You can use this function and `JotText_FirstView` to enumerate all the views on a text:

```
for (V = JotText_FirstView(text);
     V != 0; V=JotText_NextView(V) ) {
   . . .
}
```

`JotText_FirstSpan`

```
JotSpan *
JotText_FirstSpan(text)
JotText *text;
```

Returns a pointer to the first JotSpan instance in the list associated with *text*. When the list is empty, a `NULL` (0) pointer is returned.

`JotText_NextSpan`

```
JotSpan *
JotText_NextSpan(span)
JotSpan *span;
```

Returns a pointer to the JotSpan instance following *span* in the list managed by the owner of *span*. When *span* is the last JotSpan in the list, a `NULL` (0) pointer is returned.

`JotText_SetCaret`

```
boolean
JotText_SetCaret(text, position)
JotText *text;
int position;
```

Moves the caret (insertion point) to *position* in the *text* buffer. When the operation is successful, `TRUE` is returned; otherwise, `FALSE` is returned.

If the caret should be visible following the next JotView refresh (see JotView procedures—the view and controller on page 269), you must invoke:

```
if (JotText_SetCaret(aText, aPosition)) {
  JotView_EnsurePositionVisible(aView, JotText_Caret (aText));
}
```

Errors: `Jot_ERANGECHECK`

`JotText_Caret`

```
int
JotText_Caret(text)
JotText *text;
```

Returns the caret position (insertion point) in the *text* buffer.

Undoing and redoing JotText operations

Jot has an undo facility built into it. The undo operations are part of the JotText interface.

The way any undo works is by keeping a history of editing operations. To undo an insertion, the corresponding deletion is done. To undo a deletion, the deleted characters are reinserted. Two possible approaches to the undo operation are:

1. The accountant's approach—an operation that is undone becomes a part of the edit history.

2. The 1984 approach—back up in time, thus rewriting history.

Jot uses the 1984 approach. In this approach undoing an operation can be thought of as going backward in time through the edit history. Redoing an operation can be though of as simply moving forward through time, or "back to the future." Once back in time, starting new editing operations erases the future. Jot uses an N level undo, which means it will remember an arbitrary number of editing operations on a per JotText basis.

In Jot, consecutive insertions (or deletions) are grouped into one undoable editing operation. Jot provides functions to break up the default grouping, or to group together operations that are normally considered separate.

`JotText_SetUndo`

```
void
JotText_SetUndo(text, level)
JotText *text;
int     level;
```

Sets the undo level in *text* to *level*. If the specified level is <= 0 undo is disabled in *text*. Otherwise, Jot remembers the last *level* changes to *text*. When `JotText_SetUndo` is used to shorten the edit history, some number of the oldest operations are immediately forgotten.

`JotText_Undo`

```
boolean
JotText_Undo(text)
JotText *text;
```

Effectively backs up in time, undoing one operation. JotText_Undo returns `FALSE` if you are already at the beginning of recorded history. If `JotText_Undo` is issued again before any other changes are made, time is backed up even further.

JotText_Redo

```
boolean
JotText_Redo(text)
JotText *text;
```

Redoes the last undone operation, effectively moving forward through the edit history. `JotText_Redo` returns `FALSE` if there are no more undone operations that can be redone.

JotText_RedoCount

```
int
JotText_RedoCount(text)
JotText *text;
```

Returns the number of redoable operations for *text*. This is useful for an editor which wants to guard against a naive user erasing a bunch of redoable operations.

JotText_UndoCount

```
int
JotText_UndoCount(text)
JotText *text;
```

Returns the number of undoable operations for *text*. The integer returned is a number between 0 and the level specified in `JotText_SetUndo`.

JotText_UndoBreak

```
void
JotText_UndoBreak(text)
JotText *text;
```

Specifies that the current grouping of edit operations should be stopped, and a new one started. The default grouping combines consecutive insertions or deletions at the same position in the text as one undoable operation. You use `JotText_UndoBreak` to split up that default grouping.

JotText_UndoBegin

```
void
JotText_UndoBegin(text)
JotText *text;
```

JotText_UndoEnd

```
void
JotText_UndoEnd(text)
JotText *text;
```

These two routines are used to group together operations that are normally considered separate. For instance, an EMACS fill-paragraph operation is implemented by inserting and deleting many Newline characters. The default grouping separates insertions and deletions into separate pieces of history, but that would not be the correct behavior for undoing a fill paragraph. So, EMACS implementers would use `JotText_UndoBegin` at the beginning their fill routine and `JotText_UndoEnd` at the end.

Another use for the begin and end pair might be in implementing a global replace operation. You could wrap the global replace operation in a begin/end pair so the user could undo the entire replacement operation at once. Contrast this behavior with query replace, which you want to allow the user to undo one replacement at-a-time. Thus, you wouldn't wrap a query replace in a begin/end pair.

Text spans

A JotSpan describes a specific region or location in a text buffer through a beginning position and length. (The definition of "position is on page 256.) Each JotText instance maintains a list of the JotSpans that are associated with it. The position field is adjusted whenever modifications occur before the beginning of a JotSpan. The length field is adjusted whenever modifications occur within the JotSpan boundaries. These adjustments require no intervention from the developer.

Figure 27-1 illustrates the adjustments to the JotSpan, as strings are inserted within and before its scope.

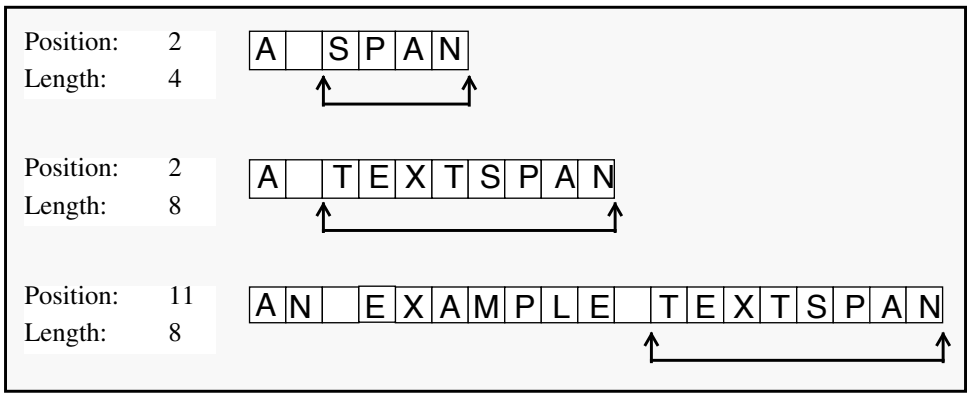


Figure 27-1 Inserting strings into a Jot span

JotSpan_New

```
JotSpan *
JotSpan_New(text, position, length)
JotText *text;
int position;
int length;
```

Allocates and returns a pointer to a new JotSpan instance. When *text* is a pointer to a JotText instance, the JotSpan is inserted into the JotSpan list managed by *text*. When *text* is a NULL (0) pointer, the JotSpan is not initially associated with a JotText instance. The position and length variables in the new JotSpan are set to the specified *position* and *length*. If the operation fails, a NULL (0) pointer is returned.

Errors: Jot_EMEMORY Jot_ERANGECHECK

JotSpan_Free

```
void
JotSpan_Free(span)
JotSpan *span;
```

Deallocates *span* and its internal resources. If *span* is owned by a JotText instance it is also removed from the JotSpan list in its owner.

JotSpan_SetText

```
boolean
JotSpan_SetText(span, text, position, length)
JotSpan *span;
JotText *text;
int position;
int length;
```

Inserts *span* into the JotSpan list managed by *text*. If *span* is already associated with a JotText instance, it will be removed from the previous list. In addition, *span* can be explicitly disassociated from its current owner by defining *text* as a NULL (0) pointer. The position and length variables in *span* are set to the specified *position* and *length*. When the operation is successful, TRUE is returned; otherwise, FALSE is returned.

Errors: Jot_ERANGECHECK

JotSpan_Text

```
JotText *
JotSpan_Text(span)
JotSpan *span;
```

Returns a pointer to the JotText instance associated with *span*. When *span* is not associated with a JotText instance, a NULL (0) pointer is returned.

JotSpan_DeleteContents

```
int
JotSpan_DeleteContents(span)
JotSpan *span;
```

Deletes the characters from the buffer managed by the owner of span using the position and length of *span*. Because spans adjust their length as they are modified the length of *span* becomes 0. When the operation is successful, the number of characters deleted is returned; otherwise, no deletion is performed and -1 is returned.

Errors: `Jot_ETEXT`

JotSpan_Contents

```
int
JotSpan_Contents(span, contents)
JotSpan *span;
char *contents;
```

Copies the characters defined by *span*, from the owner of *span* to contents. The characters are placed in *contents*. You are responsible for allocating adequate space to hold the characters. When the operation is successful, the number of characters copied is returned; otherwise, -1 is returned.

```
if(((length = JotSpan_Length(aSpan)) > 0) &&
    ((contents = (char *) malloc(length)) != 0)) {
   JotSpan_Contents(aSpan, contents);
   /* process the contents */
   free(contents);
}
```

Errors: `Jot_ETEXT`

JotSpan_Replace

```
boolean
JotSpan_Replace(old, new)
JotSpan *old;
JotSpan *new;
```

Replaces the characters described in *old* with the characters described in *new*. Both *old* and *new* must be associated with a JotText instance. When the operation is successful, `TRUE` is returned; otherwise, `FALSE` is returned.

Errors: `Jot_ECONSTRAIN Jot_EMEMORY Jot_ETEXT`

JotSpan_Position

```
int
JotSpan_Position(span)
JotSpan *span;
```

Returns the value of the position variable in *span*. If the operation fails, a -1 is returned.

Errors: `Jot_ETEXT`

`JotSpan_SetPosition`

```
boolean
JotSpan_SetPosition(span, position)
JotSpan *span
int position;
```

Sets the position variable in *span* to the specified *position*. The *span* must be associated with a JotText instance. When the operation is successful, `TRUE` is returned; otherwise, `FALSE` is returned.

Errors: `Jot_ERANGECHECK Jot_ETEXT`

`JotSpan_Length`

```
int
JotSpan_Length(span)
JotSpan *span;
```

Returns *span*'s length. If the operation fails, a -1 is returned.

Errors: `Jot_ETEXT`

`JotSpan_SetLength`

```
boolean
JotSpan_SetLength(span, length)
JotSpan *span;
int length;
```

Sets the length variable in *span* to the specified *length*. The *span* must be associated with a JotText instance. When the operation is successful, `TRUE` is returned; otherwise, `FALSE` is returned.

Errors: `Jot_ERANGECHECK Jot_ETEXT`

`JotSpan_Set`

```
boolean
JotSpan_Set(span, position, length)
JotSpan *span;
int position;
int length;
```

Sets the position and length variables in span to the specified *position* and *length*. The *span* must be associated with a JotText instance. When the operation is successful, `TRUE` is returned; otherwise, `FALSE` is returned.

Errors: `Jot_ERANGECHECK Jot_ETEXT`

`JotSpan_Modified`

```
boolean
JotSpan_Modified(span)
JotSpan *span;
```

Returns the modified state of *span*. Whenever a text operation occurs in a span, the span is marked as modified and stays marked until `JotSpan_ClearModified` is called. When *span* is not associated with a JotText instance, `FALSE` is returned.

`JotSpan_ClearModified`

```
void
JotSpan_ClearModified(span)
JotSpan *span;
```

Sets the modified state of *span* to `FALSE`.

JotView procedures—the view and controller

The view associated with the text model is the JotView. This view reads characters from a text buffer, applies formatting information, and displays the results in a rectangular canvas on the screen. A JotText instance can be associated with multiple JotViews.

The JotView interface also encompasses the controller interface. The user interacts with text using the mouse and keyboard. Mouse events such as selections or caret movement are handled in the controller. Jot provides no-op default controllers.

`JotView_New`

```
JotView *
JotView_New(text)
JotText *text;
```

Allocates and returns a pointer to a new JotView instance. In addition, a canvas is created on the window server to view text. This canvas is neither mapped nor activated. A `NULL` (0) pointer is returned if the operation fails.

You can use `JotView_New` to create and associate a JotView with an existing JotText instance in one operation. When this behavior is not desired, *text* should be defined as a `NULL` (0) pointer.

Errors: `Jot_EMEMORY`

`JotView_Free`

```
void
JotView_Free(view)
JotView *view;
```

Deallocates *view* and its internal resources, including the canvas created in the window server. As necessary, *view* is removed from the JotView list in its associated JotText.

`JotView_SetText`

```
void
JotView_SetText(view, text)
JotView *view;
JotText *text;
```

Inserts *view* into the JotView list managed by *text*. This operation permits *text* to be presented and edited in the canvas controlled through *view*. If *view* is already associated with a JotText instance, it is removed from the previous list. In addition, *view* can be explicitly disassociated from its current owner by defining *text* as a `NULL` (0) pointer.

`JotView_Text`

```
JotText *
JotView_Text(view)
JotView *view;
```

Returns the JotText instance associated with *view*. When *view* is not associated with a JotText instance, a `NULL` (0) pointer is returned.

`JotView_Characters`

```
int
JotView_Characters(view)
JotView *view;
```

Returns the number of characters in *view*. Should the operation fail, a -1 is returned.

The information returned is consistent with the last change to the text buffer presented in *view*, i.e., if the buffer's contents have changed, Jot formats it to calculate the answer. However, *view* is not redisplayed, which may result in the returned value being inconsistent with what is displayed in *view*.

`JotView_EnsurePositionVisible`

```
boolean
JotView_EnsurePositionVisible(view, position)
JotView *view;
int position;
```

Forces the character at *position* to be visible after the next update in *view*. When the operation is successful, `TRUE` is returned; otherwise, `FALSE` is returned.

Errors: `Jot_ERANGECHECK`

`JotView_ConstrainText`

```
void
JotView_ConstrainText(view, constrain)
JotView *view;
boolean constrain;
```

Prevents scrolling and limits character insertion to the amount of space in *view,* when *constrain* is `TRUE`. `JotView_ConstrainText` is useful in forms-based applications where each JotView represents a field in a form.

At some future time, during a insertion into the text buffer (e.g., using `JotText_InsertCharacters`), the insertion may fail and the insertion function will return -1 and the error number will be `Jot_ECONSTRAIN`.

`JotView_Update`

```
boolean
JotView_Update(view)
JotView *view;
```

Forces *view* to update its display based on the current state of the associated JotText buffer and the caret location (the insertion point). If the buffer hasn't changed and the caret hasn't moved, no action occurs. When the operation is successful, `TRUE` is returned; otherwise, `FALSE` is returned.

In a editor with a C language mode, the caret might be moved to the matching delimiter when a close brace is entered. (This is commonly referred to as "paren flashing.") After a brief interval, the cursor would then restored to its original position. The following code demonstrates how Jot functions could be used to implement such behavior:

```
/* A simple paren flashing example*/
aText = JotView_Text(aView);
CurrentLocation = JotText_Cursor(aText);
JotText_SetCursor(aText, AnotherLocation);
JotView_EnsurePositionVisible(aView, AnotherLocation);
JotView_Update(aView);
sleep(1);
JotText_SetCursor(aText, CurrentLocation);
JotView_EnsurePositionVisible(aView, CurrentLocation);
JotView_Update(aView);
```

`JotView_UpdateViews`

```
void
JotView_UpdateViews()
```

Updates all the views that need updating. Changes are made to text buffers, they are all batched together and the screen is refreshed once. Every time through the main loop, `JotView_UpdateViews` should be called to ensure that all views that need updating for one reason or another *are*, in fact, updated.

It is necessary for applications using Jot to own the main loop and call `JotView_UpdateViews()`. E.g.,

```
while (wire_Notify()) {
     /* If there is something else coming over the wire
        don't bother updating the views this time, so we
        can batch keystrokes, etc. */
     if (wire_WouldNotify())
          continue;
     JotView_UpdateViews();
    /* maybe do something else*/
}
```

Errors: `Jot_ETEXT`

`JotView_Lines`

```
int
JotView_Lines(view)
JotView *view;
```

Returns the number of lines in *view*. Should the operation fail, a -1 is returned.

The information returned is consistent with the last change to the text buffer presented in *view*, i.e., if the buffer's contents have changed, Jot formats it to calculate the answer. However, *view* is not redisplayed, which may result in the returned value being inconsistent with what is displayed in *view*.

Errors: `Jot_ETEXT`

`JotView_PositionFromLine`

```
int
JotView_PositionFromLine(view, line)
JotView *view;
int line;
```

Returns the position preceding the first character displayed on *line* in *view*. If you ask what this character is you get the *first* character in line; remember position is defined to be *between* characters. *line* must be greater than or equal to zero and less than the value returned from `JotView_Lines` (page 272). Should the operation fail, a -1 is returned.

```
/* First position in view */
First = JotView_PositionFromLine(aView, 0)
/* Last position in view */
Last = (First + JotView_Characters(aView)) - 1;
```

Errors: `Jot_ERANGE`, `Jot_ETEXT`

The JotBoundingBox data type

A new data type, JotBoundingBox, is used to determine the size of a view. Although characters are presented on a canvas managed in a JotView instance, if you use margins, the view size and origin might not be equivalent to those of the canvas. The `JotView_Height`, `JotView_Width`, `JotView_LineBoundingBox`, and `JotView_BoundingBox` functions take the margins into account. In addition, the values are relative to the Current Transformation Matrix (CTM) for the canvas.

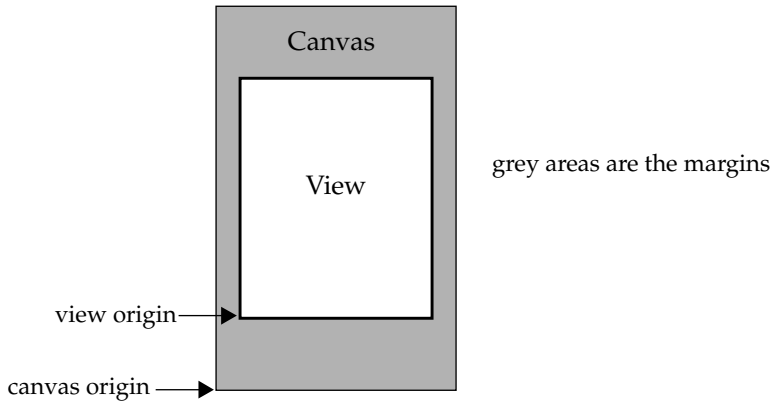


Figure 27-2 Relationship of a Jot view to a canvas

The origin for JotView is the lower-left corner. The declaration for `JotBoundingBox` is:

```
typedef struct {
   int x;
   int y;
   int width;
   int height;
} JotBoundingBox;
```

`JotView_Height`

```
int
JotView_Height(view)
Jotview *view;
```

Returns the *view* height in points

`JotView_Width`

```
int
JotView_Width(view)
JotView *view;
```

Returns the *view* width in points.

`JotView_SetMargins`

```
void
JotView_SetMargins(view, left, right, top, bottom)
JotView *view;
int left, right, top, bottom;
```

Sets the margins for a view. *left*, *right*, *top* and *bottom* are specified in points.

`JotView_BoundingBox`

```
void
JotView_BoundingBox(view, box)
JotView *view;
JotBoundingBox *box;
```

Copies the values for the origin, width and height of *view* into *box*. You are responsible for allocating *box*.

`JotView_LineBoundingBox`

```
boolean
JotView_LineBoundingBox(view, box, line)
JotView *view;
JotBoundingBox *box;
int line;
```

Copies the values for the origin, width and height of *line*, in *view*, to *box*. *line* must be greater than or equal to zero and less than the value returned from `JotView_Lines` (page 272). When the operation is successful, `TRUE` is returned; otherwise, `FALSE` is returned. You are responsible for allocating *box*.

The information returned is consistent with the last change to the text buffer presented in *view*, i.e., if the buffer's contents have changed, Jot formats it to calculate the answer. However, *view* is not redisplayed, which may result in the returned value being inconsistent with what is displayed in *view.*

Errors: `Jot_ERANGECHECK`, `Jot_ETEXT`

`JotView_RelativeLineFromPosition`

```
int
JotView_RelativeLineFromPostion(view, position)
JotView *view;
int position;
```

Returns the line number containing the character at *position* in the text buffer being presented in *view*. This line number is relative to *view,* so its value is greater than or equal to zero and less than the value returned from `JotView_Lines`. When the character is not contained in *view* or the operation fails for other reasons, a -1 is returned.

Errors: `Jot_ERANGECHECK`, `Jot_ETEXT`

`JotView_LineFromPosition`

```
int
JotView_LineFromPosition(view, position)
JotView *view;
int position;
```

Returns the number of the line containing the character located at *position*. `JotView_LineFromPosition` uses *view*'s width to calculate line width but pretends that view's height is adequate to present the entire JotText buffer. The first character in the buffer appears at the top of view. Should the operation fail, a -1 is returned.

`JotView_LineFromPosition` needs to format text from the beginning of the buffer to calculate the line number; performance will be degraded during this process.

Errors: `Jot_ERANGE`, `Jot_ETEXT`

`JotView_ScrollAutomatic`

```
void
JotView_ScrollAutomatic(view, height)
JotView *view;
int height;
```

Specifies how you want the view to scroll when the caret would be positioned outside *view* without scrolling. Scrolling is specified as follows:

- When *height* is positive, the caret is positioned *height* points below the top or above the bottom of *view*.
- When *height* is negative, the caret is positioned *height* points from the opposite end of *view*.
- When the absolute value of *height* is zero or greater than the value returned by `JotView_Height`, the caret is centered in *view*.

The direction to scroll is determined by whether the caret is moving forward or backward. For automatic scrolling to occur, `JotView_EnsurePositionVisible` must also be invoked.

`JotView_ScrollRelative`

```
boolean
JotView_ScrollRelative(view, units, count)
JotView *view;
int units;
int count;
```

Scrolls text in *view* based on line or page *units*. The *count* parameter determines the number of *units* to scroll. When *count* is positive, the text is scrolled forward. When *count* is negative, the text is scrolled backward. When *count* is zero, the operation is ignored and `TRUE` is returned. If the absolute value of *count* exceeds the actual number of lines or pages, *count* is regarded as the maximum number of *units*. The *units* parameter must be defined as `Jot_LINE` or `Jot_PAGE`. If further scrolling is not possible, `FALSE` is returned; otherwise, `TRUE` is returned.

```
while (JotView_ScrollRelative(view, Jot_PAGE, 1))
  ;
```

Errors: `Jot_ETEXT`

`JotView_ScrollAbsolute`

```
boolean
JotView_ScrollAbsolute(view, position, YCoordinate)
JotView *view;
int position;
int YCoordinate;
```

Scrolls text in *view* until the line containing the character at *position* is visible. The baseline for this line is positioned as close as possible to the point defined in *YCoordinate*. To calculate appropriate values for *YCoordinate*, you should reference the values returned from `JotView_BoundingBox` or `JotView_Height`, e.g.,

```
JotView_BoundingBox (aView, &aBox);
Bottom = aBox.y;
Top = Bottom + aBox.height;
```

```
JotView_ScrollAbsolute (aView, 1000, Top);
```

Errors: `Jot_ETEXT`

`JotView_SetReadOnly`

```
void
JotView_SetReadOnly(view, browse)
JotView *view;
boolean browse;
```

Determines whether the view is protected from user input. Jot does not enforce the read-only state. Your keyboard controller will have to check to see if read only is turned on and then enforce it. When *browse* is `TRUE` the view is read only and the caret is not visible. Scrolling and selections remain available in a read only view with the following limitations on manipulating the selection:

- selections cannot be cut.
- selections cannot be pasted into the view.
- selections can be copied out of the view.

`JotView_ReadOnly`

```
boolean
JotView_ReadOnly(view)
JotView *view;
```

Indicates whether *view* is protected from input.

`JotView_SetData`

```
void
JotView_SetData(view, pointer)
JotView *view;
caddr_t pointer;
```

Associates a client data *pointer* with *view.*

`JotView_Data`

```
caddr_t
JotView_Data(view)
JotView *view;
```

Returns the client data pointer previously associated with *view.* If no client data has been associated with *view,* a `NULL` (0) pointer is returned.

The Wire Service and JotViews

Please see the example at the end of this chapter for information on the relationship between the Wire Service and JotViews, including how to associate a scrollbar to the JotView.

`JotView_Canvas`

```
int
JotView_Canvas(view)
JotView *view;
```

Returns the Wire Service user token representing the canvas managed by *view*.

`JotView_View`

```
JotView *
JotView_View(connection, token)
wire_Wire connection;
int token;
```

Returns a pointer to the JotView instance associated with the Wire Service user *token* on the Wire Service *connection*. Should the operation fail, a `NULL` (0) pointer is returned.

`JotView_Wire`

```
wire_Wire
JotView_Wire(view)
JotView *view;
```

Returns the Wire Service connection associated with *view*. Should the operation fail, `wire_INVALID_WIRE` is returned.

View Controllers

Jot provides a mechanism to register controllers that control user input into the JotView. The keyboard and mouse default controllers are implemented as no-ops. The selection controllers have implement Toolkit semantics for selections. You can install the default if you no longer want to get input from the associated device. For example, if you don't want to get input from the mouse you can use the default mouse controller.

`JotView_SetControllers`

```
void
JotView_SetControllers(view, keyboard, mouse, SelectionStart, SelectionAlter)
          JotView *view;
          void (*keyboard) ();
          void (*mouse) ();
          boolean (*SelectionStart) ();
          void (*SelectionAlter) ();
```

Registers the controllers that respond to various inputs in *view*. Jot provides default controllers that are used when you do not define and register a private controller. To unregister your private controller, you should execute `JotView_SetControllers` using the appropriate default controller name.

When a controller parameter is defined as NULL, it is ignored. This permits you to change one or more controllers without registering an existing controller multiple times.

```
void
keyboard(view, character)
JotView *view;
int character;
```

The *keyboard* controller is executed whenever keyboard events are detected in *view*.

```
void
mouse(view, button, action, position)
JotView *view;
int button;
int action;
int position;
```

The *mouse* controller is called for the entire time between mouse down and mouse up on either SELECT or ADJUST and includes mouse drags. The default controller is JotView_MouseDefault.

button is defined as either Jot_SELECTBUTTON or Jot_ADJUSTBUTTON. *action* is defined as Jot_BUTTONUP, Jot_BUTTONDOWN, or Jot_MOUSEDRAGGED. *position* corresponds to the point preceding a character in the text buffer being presented in *view*.

```
boolean
SelectionStart(view, position)
JotView *view;
int position;
```

The SelectionStart controller is executed whenever a request to begin a selection is made. Your SelectionStart controller determines whether a selection should be made and returns true if it should be started. The default controller is JotView_SelectionStartDefault.

position corresponds to the point preceding a character in the text buffer being presented in *view*. When JotView_SelectionStart returns TRUE, a new selection is created.

Note – In an editing application, a click on SELECT button might reset the insertion point or start a selection. A hypertext application might interpret this action in a different manner. For instance, a text link could be activated in the Help Viewer.

```
void
SelectionAlter(view, rank)
JotView *view;
int rank;
```

The `SelectionAlter` controller is executed whenever the selection is adjusted in *view*. The default controller is `JotView_SelectionAlterDefault`.

rank is defined as either `Jot_PRIMARY` or `Jot_SECONDARY`.

JotSearch procedures

JotSearch uses the same syntax as regular expressions to conduct its text searches. Regular expressions are passed to `JotSearch_CompileExpression`, which compiles them into a form usable for pattern matching.

`JotSearch_New`

```
JotSearch *
JotSearch_New()
```

Allocates and returns a pointer to a new JotSearch instance. If the operation fails a `NULL` (0) pointer is returned.

Errors: `Jot_EMEMORY`

`JotSearch_Free`

```
void
JotSearch_Free(search)
JotSearch *search;
```

Deallocates *search* and its internal resources.

JotSearch_CompileExpression

```
boolean
JotSearch_CompileExpression(search, string, regular)
JotSearch *search;
char *string;
boolean regular;
```

Compiles *string* and copies the results to *search*. When *regular* is TRUE, *string* is considered a regular expression search string, and certain characters will be treated specially (See ed(1) for a list of these characters.). When *regular* is TRUE, the special characters must be escaped if they are to be treated as literal. When the operation is successful, TRUE is returned; otherwise, FALSE is returned.

A new JotSearch instance does not have to be created for each search; an existing instance can be passed repeatedly to JotSearch_CompileExpression.

Errors: Jot_EMEMORY, Jot_ESYNTAX

JotSearch_MatchPattern

```
boolean
JotSearch_MatchPattern(search, range, match, direction, case)
JotSearch *search;
JotSpan *range;
JotSpan *match;
int direction;
boolean case;
```

Matches the compiled expression in *search* against characters in the text buffer managed by the owner of *range*. The position and length variables of *range* are examined to determine the search region. The *direction* must be defined as Jot_FORWARD or Jot_BACKWARD. *case* determines whether case should be considered during the search. When the operation is successful, the text, position, and length variables describing the matched characters are copied into *match* and TRUE is returned; otherwise, FALSE is returned.

JotSearch_MatchString

```
boolean
JotSearch_MatchString(string, range, match, direction, case)
char *string;
JotSpan *range;
JotSpan *match;
int direction;
boolean case;
```

Matches *string* against characters in the text buffer managed by the owner of *range*. The position and length variables in *range* are examined to determine the search region. *direction* must be defined as Jot_FORWARD or Jot_BACKWARD. *case* determines whether case should be considered during the search. When the

operation is successful, the text, position, and length variables describing the matched characters are copied into *match* and `TRUE` is returned; otherwise, `FALSE` is returned.

Note – `JotSearch_MatchString` does not support regular expressions. It is used for simple searches.

`JotSearch_Substring`

```
boolean
JotSearch_Substring(search, substring, match)
JotSearch *search;
int substring;
JotSpan *match;
```

Accesses the *substring* register from the current regular expression in *search*. The *search* parameter must have been passed to a successful execution of `JotSearch_MatchPattern`. When the operation is successful, the text, position, and length variables describing the contents of *substring* register are copied into *match* and `TRUE` is returned; otherwise, `FALSE` is returned.

Notes: From the ed(1) manual pages:

\n - Match the contents of the nth substring register from the current RE. This provides a mechanism for extracting matched substrings. For example, the expression \(.*\)\1$ matches a line consisting of two repeated appearances of the same string. When nested parenthesized substrings are present, n is determined by counting occurrences of \(starting from the left.]

Errors: `Jot_ERANGECHECK`

JotFont procedures

`JotFont_New`

```
JotFont *
JotFont_New(name, size, printermatched)
char *name;
int size;
boolean printermatched;
```

Returns a pointer to a JotFont instance. The JotFont cache is searched for a font matching the specified parameters. When the search is successful, the pointer to an existing JotFont instance is returned; otherwise, the window server is requested to locate the font corresponding to *name*. If the font is not located, a

NULL (0) pointer is returned; otherwise, the font is scaled to *size*. When *printermatched* is TRUE, printer metrics are imposed on the font. The pointer to the new JotFont instance is returned and the instance is cached for reuse.

The Times-Roman 12 is the default font.

Errors: Jot_EMEMORY

JotFont_Free

```
void
JotFont_Free(font)
JotFont *font;
```

Deallocates *font* and its internal resources. In addition, *font* is removed from the JotFont cache.

JotView_SetFont

```
void
JotView_SetFont(view, font)
JotView *view;
JotFont *font;
```

Sets the font in view to *font*.

JotSelection procedures

Selections are a mechanism that permit client applications to exchange information. This service allows text to be cut, copied, and pasted between applications. Although the window server might support an arbitrary number of selections, the Jot interface the Jot interface supports only primary and secondary selections.

JotSelection_Set

```
boolean
JotSelection_Set(span, rank, pendingdelete)
JotSpan *span;
int rank;
boolean pendingdelete;
```

Uses the position and length variables in *span* to set the *rank* selection. The *rank* must be defined as either Jot_PRIMARY or Jot_SECONDARY. When *pendingdelete* is TRUE, the selection is to be pending delete and will be replaced with the next inserted character. The characters defined by *span* are highlighted when visible in associated JotViews. When the operation is successful, TRUE is returned; otherwise, FALSE is returned.

Errors: Jot_ERANGECHECK Jot_ETEXT

`JotSelection_SetLevel`

```
boolean
JotSelection_SetLevel(view, rank, level)
JotView *view;
int rank;
int level;
```

Sets the *rank* selection in *view* to the specified *level*. The *rank* must be defined as `Jot_PRIMARY` or `Jot_SECONDARY`. The *level* must be defined as `Jot_CHARACTER`, `Jot_WORD`, `Jot_LINE`, or `Jot_BUFFER`. When the operation is successful, `TRUE` is returned; otherwise, `FALSE` is returned.

Errors: `Jot_ERANGECHECK`

`JotSelection_Clear`

```
boolean
JotSelection_Clear(text, rank)
JotText *text;
int rank;
```

Clears the *rank* selection in *text*. The *rank* must be defined as `Jot_PRIMARY` or `Jot_SECONDARY`. When *text* does not own the selection, the operation is ignored and `FALSE` is returned; otherwise, `TRUE` is returned. The characters addressed in the selection are unhighlighted when visible in associated JotViews.

Errors: `Jot_ERANGECHECK`

`JotSelection_Span`

```
JotSpan *
JotSelection_Span(text, rank)
JotText *text;
int rank;
```

Returns a pointer to a JotSpan that represents the selection of rank *rank*. The *rank* must be defined as `Jot_PRIMARY` or `Jot_SECONDARY`. `JotSelection_Span` supports only primary and secondary selections. In addition, *text* must own the *rank* selection. When the operation is not successful a `NULL` (0) pointer is returned.

A JOT Example

This is a Jot example that consists of three files: joe.c, joe_cps.cps, and Makefile. This is an example of a very simple text editor.

```
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <stdio.h>
#include <strings.h>
#include "NeWS/wire/wire.h"
#include "NeWS/jot/jot.h"
#include "NeWS/jot/view.h"
#include "NeWS/jot/font.h"
#include "joe_cps.h"

#define CTL(c) (c & 037)

JotView*TheView;

/*
 * Keyboard callback for the JotView. This handles:
 *             Control-A moves to the beginning of the document.
 *             Control-B moves backward one character.
 *             Control-D deletes the character after the caret.
 *             Control-E moves to the end of the document.
 *             Control-F moves forward one character.
 *             Delete deletes the character before the caret.
 *
 *
 * All other characters are inserted in the text buffer based on the
 * caret position. Following the insertion, the caret is
 * advanced.
 */
```

```
static void
keyboard_callback (view, ch)
JotView *view;
char    ch;
{
    JotText *text;
    int   caret_pos;

    text = JotView_Text(view);
    caret_pos = JotText_Caret(text);

    switch (ch) {
    case CTL('D'):
    case '\177':
        (void) JotText_DeleteCharacters(text, caret_pos, (ch == CTL('D')) ? 1 : -1);
        break;

    case CTL('B'):
        if (caret_pos > 0)
            JotText_SetCaret (text, caret_pos - 1);
        break;

    case CTL('F'):
        if (caret_pos < JotText_Characters(text))
            JotText_SetCaret(text, caret_pos + 1);
        break;

    case CTL('E'):
        JotText_SetCaret(text, JotText_Characters(text));
        break;
```

```
    case CTL('A'):
        JotText_SetCaret(text, 0);
        break;

    case '\r':
        ch = '\n';
        /* Fall into */

    default:
        JotText_InsertCharacters(text, caret_pos, &ch, 1);
        break;
    }

    JotView_EnsurePositionVisible(view, JotText_Caret(text));
}

/*
 * A menu callback has been made. We snarf the font family and try to create that font.
 * If we succeed, the global font in view is
 * set to the new font; otherwise, an error message is printed.
 */


void
menu_callback(tag, data)
{
    JotFont *f;
    char name[128];

    wire_ReadString(name);
```

```
    if ((f = JotFont_New(name, 12, FALSE)) == NULL) {
        fprintf(stderr, "Error finding font %s.\n", name);
        return;
    }
    JotView_SetFont(TheView, f);
}

main(argc, argv)
int     argc;
char    **argv;
{
    JotText    *text;
    wire_Wire  wire;
    struct statstbuf;
    int        fd, menu_tag, nbytes;
    char       *filename;

 if (argc < 2) {
     fprintf(stderr, "%s: usage: %s file\n", argv[0], argv[0]);
     exit(1);
    }
    filename = argv[1];

    /* A server connection is opened and initialized. */
    if ((wire = wire_Open(NULL)) == wire_INVALID_WIRE) {
     wire_Perror();
     exit(1);
    }

    Jot_Initialize(wire);
```

```
/* The file is opened and its length is acquired. A text
    object is created with a little extra room and the file is
    read into the text object. */

if ((fd = open(filename, 0)) == -1) {
 perror(filename);
 exit(1);
}

if (fstat(fd, &stbuf) == -1) {
 perror(filename);
 exit(1);
}

if ((text = JotText_New(stbuf.st_size + 1024)) == NULL) {
 fprintf(stderr, "Cannot create text object\n");
 exit(1);
}

nbytes = JotText_Read(text, 0, fd);
printf("Read %d bytes from file %s.\n", nbytes, filename);
close(fd);

/*
* Now allocate a tag for the menu. The menu callback will read a font family
* name from the wire and make that the font for the view. When the view is
* created, we associate the menu with that canvas using the /setmenu method
* in ClassCanvas. The ps_initialize routine downloads a couple of useful
* functions into the server, as well as
* creating the font menu.
*/
```

```
    menu_tag = wire_AllocateTags(1);
    wire_RegisterTag(menu_tag, menu_callback, 0);
    ps_joe_initialize(menu_tag);

    /*
    * The view is created and the text object is associated with it. The canvas
    * managed by the view is made a client of a baseframe through the ps_initcanvas
    * routine. When the baseframe is mapped, the canvas will be mapped
    * and painted as well.
    */

    TheView = JotView_New(text);
    ps_initcanvas(JotView_Canvas(TheView));

    /* the keyboard callback is called whenever a character is typed in the JotCanvas */

    JotView_SetControllers(TheView, keyboard_callback, NULL, NULL, NULL);

    /* "Off on your way, hit the open road, there is magic at your
         fingers. For the spirit ever lingers, undemanding contact
         in your happy solitude." */

    while (wire_Notify()) {
     if (wire_WouldNotify(wire_ALLWIRES))
         continue;
     JotView_UpdateViews();
 }
}
```

```
% Define a routine and a menu.
%
% init_canvas accepts a JOT canvas parameter. It creates a base window
% and a scrollbar, and inserts the scrollbar and the JOT canvas
% into the base window. Then it attaches the scrollbar to the JOT
% canvas. When it's done it creates an event manager and activates
% the base window, which activates the JOT canvas and scrollbar.
%
% The menu is simply a list of well-known font families. When a
% client selects an item from the menu, the selected family name
% is sent up the wire to the client-side menu callback.
%

cdef ps_joe_initialize(menu_tag)
    /init_canvas { % JotCanvas => -
     5 dict begin
         /cv exch def
         /win cv framebuffer /new ClassBaseWindow send def
         /sbar framebuffer /new ClassVScrollbar send def

         /preferredsize { 500 600 } /installmethod cv send
         cv /settarget sbar send
         sbar /ScrollbarInitialize cv send
         /Scroll /setnotifier sbar send
         /Scroll /setpreviewer sbar send

         /East sbar /addclient win send

         0 2 0 2 /setgaps win send

         0 0 /preferredsize win send /reshape win send
```

```
        /place win send

        (JOE - Jonathan's Own Editor) /setlabel win send
        /new ClassEventMgr send /activate win send
        /map win send
        win                                                          % window
    end
   /JoeWindow exch def
} def

/FontMenu /Grid framebuffer /new ClassMenu send def
[ (Courier) (Helvetica) (Times-Roman) (LucidaSans) ]
    /setitemlist FontMenu send
{                                                                  % index menu
      /item exch send 0 get                                         % item-string
      menu_tag tagprint                                             % item-string
      typedprint                                                    % -
} /setnotifier FontMenu send

% Call init_canvas with a JotCanvas. This initializes the canvas
% and then sets its menu.

cdef ps_initcanvas(token cv)
 cv init_canvas
 FontMenu /setmenu cv send
 true /setmenuable cv send
```

```
# Makefile for JOE. TNTHOME should point at the top level of a
# TNT release directory. Just say "make" and JOE will be built.
# To run JOE just say "joe filename" Joe is a simple file viewer
# program based on JOT.

SRC = joe.c
OBJ = joe.o
CPSFILES = joe_cps.cps
INCLUDE = -I$(TNTHOME)/include
LIBPATH = -L$(TNTHOME)/lib -L$(OPENWINHOME)/lib

CFLAGS += $(INCLUDE) -g

joe:      $(OBJ)
          cc -o joe $(OBJ) $(LIBPATH) -ljot -lwire -lcps

joe.o:    joe_cps.h

.SUFFIXES: .cps
.cps.h:; cps $*.cps
SHAR_EOF
fi
exit 0
#         End of shell archive
```

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Class Index

ClassCanvas

ClassCheckBoxes

ClassControl

ClassLabel

ClassLayout

ClassMenu

ClassMenuButtons

ClassNotice

ClassNumericField

ClassObject

ClassPanel

ClassPopWindow

ClassRegion

Scrollbars

ClassScrollList

ClassSelection

ClassSettings

Sliders

ClassTextCanvas

ClassTextField

ClassWindow

WireService